ffmpeg的riscv优化介绍

plct实验室
孙越池 sunyuechi@iscas.ac.cn
2023-12-15

主要内容:

- 简介
- ffmpeg riscv部分介绍
- 添加一个rvv优化和一个测试
- 未来计划

简介

FFmpeg是一个使用非常广泛的开源多媒体框架，支持编解码、转码、复用/解复用, 过滤等多种功能。

它几乎支持所有音视频格式（来自不同的委员会，社区，公司..), 支持各种构建环境 (Linux、Mac OS X、Microsoft Windows、BSD、Solaris ..)

在RISC-V系统中，音视频处理仍处于早期发展阶段。

优化FFmpeg会是一个不错的选择，可以期待随着ffmpeg的性能改善，RISC-V的用户会得到更流畅、高效的音视频处理体验。

这样的优化可以包括针对RISC-V特性的算法优化、利用硬件特性进行编解码加速等。

ffmpeg riscv 历史

ffmpeg在2021 10月添加了对riscv的支持, 可以在riscv构建了

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

ffmpeg在上个月，11月10号发布的6.0.1 release中，正式包含了riscv的优化

代码观察

在ffmpeg 8个涉及汇编优化的架构中，有三个是使用了intrinsics, 五个是直接写汇编来优化的,包括x86, arm, riscv

- intrinsics: ppc mips loongarch
- asm: x86 riscv arm alpha aarch64

在ffmpeg 6.0.1 release中 主要添加了下面的几个riscv目录

- libavutil/riscv/
- libavcodec/riscv/
- libswscale/riscv/
- tests/checkasm/riscv/

- libavutil/riscv/

  - timer.h : riscv的timer, 通过rdcycle来计时，benchmark时会用到
  - cpu.h cpu.c : 包括 compilation/boot/run-time的cpu检测 用于后续设置正确的flag，在合适的情况使用对应的优化
  - asm.S : 包括一些常用的宏
  - intmath.h : 一些基本的位操作如clipping的优化，利用zbb扩展的优化等

- libavcodec/riscv/ : 对编解码dsp的具体优化，每个dsp优化要包含一个init.c文件，根据汇编文件使用到的扩展，包含rvv.S,rvi.S,rvf.S 等汇编文件 主要是通过v扩展(1.0)，其次是b扩展

- libswscale/riscv/ :是对颜色转换，缩放库的具体优化，和前面libavcodec中的格式相同

- tests/checkasm/riscv/
  - checkasm.S 这个目录只有一个文件，验证当前环境是否能够正确的修改寄存器，能否处理扩展等基本功能的正确性

工作流程

riscv部分的基本结构已经搭建好了，所以现在优化ffmpeg，要做的主要是优化dsp函数

因为rvv1.0的设备还未普及，而提交到上游又需要有真机的benchmark结果，这里在rvv0.7的设备，lpi4a上进行工作

首先要把ffmpeg测试用的benchmark跑起来，benchmark会分别输出c语言实现的函数与汇编函数的运行时间，可以用来观察性能改进

执行步骤

首先这里按照lpi4a文档配置了revyos的环境
https://wiki.sipeed.com/hardware/zh/lichee/th1520/lpi4a/1_intro.html
然后按照下面ffmpeg标准的对某个功能benchmark的流程，会在lpi4a或者其他rvv0.7的设备会遇到一些问题，下面是一些解决方法

```
git clone https://git.ffmpeg.org/ffmpeg.git ffmpeg
./configure
make fate-checkasm-aacpsdsp
tests/checkasm/checkasm --bench --test=aacpsdsp
```

configure问题

首先会发现make没有使用到rvv相关的文件，使用 `--enable-rvv` 也没有作用

原因是如果enable了rvv, configure里面会检查rvv1.0的vset指令，检查失败后会不提示的改为disable

```
enabled rvv && check_inline_asm rvv '".option arch,
+v\nvsetivli zero, 0, e8, m1, ta, ma"'
```

所以为了使用这些rvv可以直接把configure中检查vset部分，&& check_inline_asm … 去掉或者改写为0.7

另外优化的gcc可能对于rvv1.0或者rvv0.7的代码都可以编译通过，但这里其实希望rvv1.0的都编译报错， 所以还需要指定目标架构为0.7 `./configure --extra-cflags='-march=rv64gcv0p7'`

1.0转0.7

接下来要把汇编部分转为0.7来运行

主要分为3个部分

1. 大部分指令可以直接用下面的这个脚本替换或者注释，这个我写了一些sed

```
#!/usr/bin/env bash

declare -A replacements=(
    [", zve64f"]=""
    [", zve64d"]=""
    [", zve32f"]=""
    [", zve32x"]=""
    [", zve64x"]=""
    [", ta, ma"]=""
    [", ta, mu"]=""
```

```
    [", tu, ma"]=""

    # vle32, vlse64 vse32 vsse32
    ["\b\(vl\?s\{0,2\}\)e8\b"]="\1b"
    ["\b\(vl\?s\{0,2\}\)e16\b"]="\1h"
    ["\b\(vl\?s\{0,2\}\)e32\b"]="\1w"
    ["\b\(vl\?s\{0,2\}\)e64\b"]="\1e"

    # vlseg, vsseg,vssseg, vlsseg
    ["\b\(v[ls]s\{1,2\}eg[248]\)e8\b"]="\1b"
    ["\b\(v[ls]s\{1,2\}eg[248]\)e16\b"]="\1h"
    ["\b\(v[ls]s\{1,2\}eg[248]\)e32\b"]="\1w"

    ["6e16"]="6h"
    ["6e32"]="6w"

    ["vnsrl.wv"]="vnsrl.vv"
    ["vnsrl.wx"]="vnsrl.vx"
    ["vnsrl.wi"]="vnsrl.vi"

    ["vnclipu.wi"]="vnclipu.vi"
    ["vnclip.wi"]="vnclip.vi"
    ["vnclip.wx"]="vnclip.vx"

    ["vfredusum.vs"]="vfredsum.vs"

    ["vaaddu.vv"]="vaadd.vv"
)
```

```
sed_expr=""
for key in "${!replacements[@]}"; do
        sed_expr+="s/${key}/${replacements[$key]}/g;"
done

declare -A replacements_comment=(
    ["mf2"]=""
    ["vncvt"]=""
    ["\bmin\b"]=""
    ["vfslide1up"]=""
    ["vfslide1down"]=""
    ["vfneg"]=""
    ["vsext"]=""
    ["vrgatherei16"]=""
    ["8e64"]=""
    ["ltypei"]=""
)
for key in "${!replacements_comment[@]}"; do
        sed_expr+="/^[^\/][^\/]*${key}/s/^/\/\//;"
done

for file in $(find . -type f -name '*.S' -not -name 'asm.S' | grep riscv); do
        # echo ">>> $file"
        sed -i "${sed_expr}" "$file"
done
```

2. vsetivli部分手动去替换

因为rvv1.0使用了vsetivli在rvv0.7不存在，替换为0.7的vsetvli要多用一个寄存器, 所以替换，选择一个不冲突的寄存器

3. 宏的实现(libavutil/riscv/asm.S) 里面shnadd宏 需要重新实现，否则在lpi4a会出现不合法指令的报错

```
.macro shnadd n, rd, rs1, rs2
addi sp, sp, -4
sw t0, 0(sp)

slli t0, \rs1, \n
add \rd, t0, \rs2

lw t0, 0(sp)
addi sp, sp, 4
.endm

.macro sh1add rd, rs1, rs2
shnadd 1, \rd, \rs1, \rs2
.endm

.macro sh2add rd, rs1, rs2
shnadd 2, \rd, \rs1, \rs2
.endm

.macro sh3add rd, rs1, rs2
shnadd 3, \rd, \rs1, \rs2
.endm
```

这样修改后可以benchmark, 然后就可以开始修改ffmpeg代码了.

这个修改方法在下一部分会说

修改完成后，在lpi4a上通过benchmark的结果验证后性能改进后，为了提交到上游，需要把v0.7的代码改写为v1.0的代码，应用到master分支。

这次用默认的方式编译构建后，生成 `tests/checkasm/checkasm` 的二进制 然后在qemu上验证rvv1.0版本功能的正确性，然后就可以提交了。

```
export QEMU_LD_PREFIX="/opt/riscv64-lp64d-glibc-
bleeding-edge/riscv64-buildroot-linux-gnu/sysroot"
qemu-riscv64 -cpu
rv64,v=true,g=true,c=true,zba=true,vlen=128 checkasm --
test="$1" --bench
```

如果获得了支持rvv1.0的设备？

在上周12月8号 我得到了支持rvv1.0的k230, 这样能够直接benchmark, 可以省去上面的转换过程。

按照这里的repo安装debian https://code.videolan.org/Courmisch/k230-boot/

有一个问题是k230的内存只有500m, 在上面编译会出现out of memory的错误，

所以还是使用lpi4a来编译，只需在configure指定目标v1p0 `./configure --extra-cflags='-march=rv64gcv1p0'` (k230只用于benchmark.)

工作进展

有8个patch merge到上游master了

- af_afir: RISC-V V fcmul_add
- checkasm: test for dcmul_add
- checkasm/ac3dsp: add float_to_fixed24 test
- lavc/ac3dsp: R-V V float_to_fixed24
- lavc/vc1dsp: R-V V inv_trans
- lvac/aacenc: add ff_aac_dsp_init
- checkasm: test for abs_pow34
- lavc/aacencdsp: R-V V abs_pow34

其中有4个是rvv的优化，有3个是架构无关的测试，一个是为了测试的小的重构

https://twitter.com/FFmpeg/status/1729982097652158975

这里ffmpeg引用我的提交发了twitter, 对riscv工作进行推荐

添加一个rvv优化和一个测试

下面是添加一个rvv优化和一个测试的过程，具体解释一下修改的内容

lavc/vc1dsp: R-V V inv_trans:

这个patch是优化vc1 decoder中，数据块逆变换，恢复数据的过程, 涉及到四个处理不同大小的块数据的函数

因为过程比较简洁，在下面详细的说明一下

```
libavcodec/riscv/Makefile       |   2 ++
libavcodec/riscv/vc1dsp_init.c |  49 ++
libavcodec/riscv/vc1dsp_rvv.S  | 113 ++
libavcodec/vc1dsp.c             |   2 ++
libavcodec/vc1dsp.h             |   1 +
```

首先在vc1dsp.h中 声明riscv的init函数

libavcodec/vc1dsp.h

```
void ff_vc1dsp_init_riscv(VC1DSPContext *c);
```

然后在vc1dsp.c的新增riscv分支调用这个init

libavcodec/vc1dsp.c

```
#elif ARCH_RISCV
    ff_vc1dsp_init_riscv(dsp);
```

在Makefile中添加新增的init和汇编文件编译部分

libavcodec/riscv/Makefile

```
OBJS-$(CONFIG_VC1DSP) += riscv/vc1dsp_init.o
RVV-OBJS-$(CONFIG_VC1DSP) += riscv/vc1dsp_rvv.o
```

riscv init.c中，在合适的cpu flag处，为context设置使用新增的汇编函数(如果设置条件错了或者不使用条件 benchmark会没有结果)

在判断flag时，可能需要判断一下向量寄存器的长度，也可能需要判断汇编函数可以使用的条件 具体实现列在这里

```
#include <stdint.h>

#include "libavutil/attributes.h"
#include "libavutil/cpu.h"
#include "libavutil/riscv/cpu.h"
#include "libavcodec/vc1.h"

void ff_vc1_inv_trans_8x8_dc_rvv(uint8_t *dest, ptrdiff_t stride, int16_t *block);
void ff_vc1_inv_trans_4x8_dc_rvv(uint8_t *dest, ptrdiff_t stride, int16_t *block);
void ff_vc1_inv_trans_8x4_dc_rvv(uint8_t *dest, ptrdiff_t stride, int16_t *block);
void ff_vc1_inv_trans_4x4_dc_rvv(uint8_t *dest, ptrdiff_t stride, int16_t *block);

av_cold void ff_vc1dsp_init_riscv(VC1DSPContext *dsp)
{
#if HAVE_RVV
    int flags = av_get_cpu_flags();

    if ((flags & AV_CPU_FLAG_RVV_I64) && ff_get_rv_vlenb() >= 16) {
        dsp->vc1_inv_trans_8x8_dc = ff_vc1_inv_trans_8x8_dc_rvv;
        dsp->vc1_inv_trans_8x4_dc = ff_vc1_inv_trans_8x4_dc_rvv;
    }
    if ((flags & AV_CPU_FLAG_RVV_I32) && ff_get_rv_vlenb() >= 16) {
        dsp->vc1_inv_trans_4x8_dc = ff_vc1_inv_trans_4x8_dc_rvv;
        dsp->vc1_inv_trans_4x4_dc = ff_vc1_inv_trans_4x4_dc_rvv;
    }
#endif
}
```

最后是新增一个汇编文件，实现这些函数，这里四个函数分别处理8*8到4*4的四个不同大小的数据块
主要是关于，循环展开和int16及uint8数据的互操作的问题

```
#include "libavutil/riscv/asm.S"

func ff_vc1_inv_trans_8x8_dc_rvv, zve64x
        lh              t2, (a2)
        sh1add          t2, t2, t2
        addi            t2, t2, 1
        srai            t2, t2, 1
        sh1add          t2, t2, t2
        addi            t2, t2, 16
        srai            t2, t2, 5
        vsetivli        zero, 8, e8, mf2, ta, ma
        vlse64.v        v0, (a0), a1
        li              t0, 8*8
        vsetvli         zero, t0, e16, m8, ta, ma
        vzext.vf2       v8, v0
        vadd.vx         v8, v8, t2
        vmax.vx         v8, v8, zero
        vsetvli         zero, t0, e8, m4, ta, ma
        vnclipu.wi      v0, v8, 0
        vsetivli        zero, 8, e8, mf2, ta, ma
        vsse64.v        v0, (a0), a1
        ret
endfunc
```

```
func ff_vc1_inv_trans_4x8_dc_rvv, zve32x
        lh              t2, (a2)
        slli            t1, t2, 4
        add             t2, t2, t1
        addi            t2, t2, 4
        srai            t2, t2, 3
        sh1add          t2, t2, t2
        slli            t2, t2, 2
        addi            t2, t2, 64
        srai            t2, t2, 7
        vsetivli        zero, 8, e8, mf2, ta, ma
        vlse32.v        v0, (a0), a1
        li              t0, 4*8
        vsetvli         zero, t0, e16, m4, ta, ma
        vzext.vf2       v4, v0
        vadd.vx         v4, v4, t2
        vmax.vx         v4, v4, zero
        vsetvli         zero, t0, e8, m2, ta, ma
        vnclipu.wi      v0, v4, 0
        vsetivli        zero, 8, e8, mf2, ta, ma
        vsse32.v        v0, (a0), a1
        ret
endfunc
```

```
func ff_vc1_inv_trans_8x4_dc_rvv, zve64x
        lh              t2, (a2)
        sh1add          t2, t2, t2
        addi            t2, t2, 1
        srai            t2, t2, 1
        slli            t1, t2, 4
        add             t2, t2, t1
        addi            t2, t2, 64
        srai            t2, t2, 7
        vsetivli        zero, 8, e8, mf2, ta, ma
        vlse64.v        v0, (a0), a1
        li              t0, 8*4
        vsetvli         zero, t0, e16, m4, ta, ma
        vzext.vf2       v4, v0
        vadd.vx         v4, v4, t2
        vmax.vx         v4, v4, zero
        vsetvli         zero, t0, e8, m2, ta, ma
        vnclipu.wi      v0, v4, 0
        vsetivli        zero, 8, e8, mf2, ta, ma
        vsse64.v        v0, (a0), a1
        ret
endfunc
```

```
func ff_vc1_inv_trans_4x4_dc_rvv, zve32x
        lh              t2, (a2)
        slli            t1, t2, 4
        add             t2, t2, t1
        addi            t2, t2, 4
        srai            t2, t2, 3
        slli            t1, t2, 4
        add             t2, t2, t1
        addi            t2, t2, 64
        srai            t2, t2, 7
        vsetivli        zero, 4, e8, mf2, ta, ma
        vlse32.v        v0, (a0), a1
        li              t0, 4*4
        vsetvli         zero, t0, e16, m2, ta, ma
        vzext.vf2       v2, v0
        vadd.vx         v2, v2, t2
        vmax.vx         v2, v2, zero
        vsetvli         zero, t0, e8, m1, ta, ma
        vnclipu.wi      v0, v2, 0
        vsetivli        zero, 4, e8, mf2, ta, ma
        vsse32.v        v0, (a0), a1
        ret
endfunc
```

[FFmpeg-devel] checkasm/ac3dsp: add float_to_fixed24 test

前面的patch在函数优化时，已经有了对应的测试，但是还有很多函数缺乏测试，如果是这种情况，就需要自己添加一个

下面这个patch是添加一个架构无关的测试

首先在下面的几个文件添加配置和函数声明

tests/checkasm/Makefile | 1 +

```
AVCODECOBJS-$(CONFIG_AC3DSP)          += ac3dsp.o
```

tests/checkasm/checkasm.c | 3 ++

```
#if CONFIG_AC3DSP
    { "ac3dsp", checkasm_check_ac3dsp },
#endif
```

tests/checkasm/checkasm.h | 1 +

```
void checkasm_check_ac3dsp(void);
```

tests/fate/checkasm.mak | 1 +

```
fate-checkasm-ac3dsp
```

```
tests/checkasm/ac3dsp.c    | 70
++++++++++++++++++++++++++++++++++++++++++++++++++
```

然后主要是在test目录下对应的dsp文件中，设置合适的数据源，正确的调用call_ref, call_new, bench函数等进行测试

这个文件中主要需要考虑编解码器的context,来随机生成合适的数据，保证测试充分。

```
#include <string.h>

#include "libavutil/mem.h"
#include "libavutil/mem_internal.h"

#include "libavcodec/ac3dsp.h"

#include "checkasm.h"

#define randomize_float(buf, len)                                   \
    do {                                                            \
        int i;                                                      \
        for (i = 0; i < len; i++) {                                 \
            float f = (float)rnd() / (UINT_MAX >> 5) - 16.0f;       \
            buf[i] = f;                                             \
        }                                                           \
    } while (0)
```

```
static void check_float_to_fixed24(AC3DSPContext *c) {
#define BUF_SIZE 1024
    LOCAL_ALIGNED_32(float, src, [BUF_SIZE]);

    declare_func(void, int32_t *, const float *, unsigned int);

    randomize_float(src, BUF_SIZE);

    if (check_func(c->float_to_fixed24, "float_to_fixed24")) {
        LOCAL_ALIGNED_32(int32_t, dst, [BUF_SIZE]);
        LOCAL_ALIGNED_32(int32_t, dst2, [BUF_SIZE]);

        call_ref(dst, src, BUF_SIZE);
        call_new(dst2, src, BUF_SIZE);
```

```
if (memcmp(dst, dst2, BUF_SIZE) != 0)
            fail();

        bench_new(dst, src, BUF_SIZE);
    }


    report("float_to_fixed24");
}

void checkasm_check_ac3dsp(void)
{
    AC3DSPContext c;
    ff_ac3dsp_init(&c);

    check_float_to_fixed24(&c);
}
```

未来计划

- 可以进一步向上游提交更多的dsp的汇编优化，让riscv向成熟的架构的优化状态，比如向x86 靠近

- 对某些关键+困难的dsp进行优化，比如h264

感谢观看