计算机组织与体系结构实习报告 Lab4.2

姓名：张佳豪

学号：2200013093

大班教师：陆俊林

SIMD扩展指令设计

1. 该lab是在lab 4.1的基础上，进行一套SIMD指令系统的设计。需要确定指令操作类型、寻址方式、数据格式、编码格式等。（50分）
2. 为上述SIMD指令系统设计一套汇编助记符。（30分）

数据格式

数据格式支持u8,i8,u16,i16,u32,i32,u64,i64,f16,f32,f64. 在之后的描述中仅对部分举例(float均采用IEEE754标准) 对于整数溢出标准为饱和计算 对于浮点数遵从IEEE754标准(使用nan,inf)

编码格式

参考Risc-V指令格式如下. 使用64位指令.

63 - 48	47 - 32	31-16	15-11	10-7	6-0
rs2/rt	rs1	rd	func5	func4	opcode

和riscv最大的区别在于,当通过opcode和funct确定寄存器为向量寄存器时，rs和rd表示16个256位向量寄存器的掩码. 当确定为标量寄存器时，我们仅采用其低5位. 我们的拓展指令系统支持16个256位SIMD指令专用寄存器来加速计算

rt是rs2的别名，因为在某些指令中其用作目的(unpack)

我们方法的优点非常明显，就是事实上实现了虚拟上最大4096位simd寄存器的处理和运算并且相当灵活 举个例子，如果掩码设置了bit1和bit4，那就制定了由物理向量寄存器V1和V4拼接而成的虚拟向量寄存器，前者作为低位部分，长度为256*2=512位. 这样我们可以在不增加物理寄存器的情况下实现更大的向量寄存器

我们的方法没有处理不对齐的情况，所以正确性需要由编译器保证(多做的事不访问没有关系，只要有访问权限即可)

我们的程序是对齐的1920 = 64 * 30

编码举例(不去给opcode、funct7、funct3赋具体的值)

指令操作类型、寻址方式、汇编助记符可以从这一部分得到 指令操作类型包括: vload, vstore, vcvt, vpack, vunpack, vselect8, vselect16, vadd, vmul, ... vvadd, vvmul, ...

寻址方式包括: R[rs1]+R[rs2] (涵盖了单R[rs1]因为可以指定寄存器为x0)

杂项: 掩码为0默认行为不造成任何影响

vload rd, rs1, rs2

vload由opcode确定

func4全0

funct5全0

rs1为标量寄存器标号

rs2为标量寄存器标号

rd为16位向量寄存器掩码

描述: 从地址读数据到向量寄存器rd

根据掩码rd确定读取的数据长度，为置1bit的个数乘32字节. 地址为 R[rs1]+R[rs2] 如果 `rd` 掩码设置了 bit 0 和 bit 7，表示将数据加载到 V0 和 V7。数据长度是掩码置1位数 * 32字节 = 2 * 32 = 64字节。硬件需要从 `R[rs1]+R[rs2]` 地址开始读取 64 字节，并将前 32 字节存入 V0，后 32 字节存入 V7

延迟: 地址ALU和读取长度ALU计算可重叠，大致是一个ALU+一个访存的延迟

vstore rd, rs1, rs2

vstore由opcode确定

func4全0

funct5全0

rs1为标量寄存器标号

rs2为标量寄存器标号

rd为16位向量寄存器掩码

描述: 写向量寄存器数据到地址

根据掩码rd确定要写的数据长度，为置1bit的个数乘32字节. 地址为 R[rs1]+R[rs2]

延迟: 地址ALU和写长度ALU计算可重叠，大致是一个ALU+一个访存的延迟

vcvt.u8.f16 rd, rs1

vcvt由opcode确定

u8由funct4确定

f16由func5确定

rs1为16位向量寄存器掩码

rs2全0

rd为16位向量寄存器掩码

描述: 类型转换

根据rs1掩码和类型1确定元素数量1，根据rd掩码和类型2确定元素数量2，元素数量取两者最小值按照标准进行转换和移动，认为元素的排列是紧密的(也就是说没有间隙在向量寄存器中排列，顺序是固定的从低标号向量寄存器到高标号向量寄存器). 结果会填充到目标掩码指定的寄存器组

延迟: 元素数量可以查表一次LUT访问，最小值一次ALU，还需要一次转换单元

vpack.i16.u8 rd, rs1, rs2

vpack由opcode确定

i16由funct4确定

u8由func5确定

rs1为16位向量寄存器掩码

rs2为16位向量寄存器掩码

rd为16位向量寄存器掩码

描述: 打包

需要保证rs1，rs2掩码中1的个数相等

需要保证类型可以pack

否则为非法指令

首先根据掩码rs1和第一个类型计算元素数量1，(rs2保证1的个数相等),然后根据rd和第二个类型计算元素数量2，元素数量取最小值，然后进行正常的打包操作. 在这个例子中(如果两个算出来元素数目符合)是把rs1打包后的结果填充到rd前一半，rs2打包后的结果填充到rd后一半(可能跨寄存器)

比如掩码设置了bit0, bit1, bit2前一半是V0和V1的前一半，后一半是V1的后一半和V2

可能会发trunc等

延迟: LUT + ALU + move

vunpack.u8.i16 rd, rt, rs1

vunpack由opcode确定

u8由funct4确定

i16由func5确定

rs1为16位向量寄存器掩码

rt为16位向量寄存器掩码

rd为16位向量寄存器掩码

描述: 解包

需要保证rd，rt掩码中1的个数相等

首先根据掩码rs1和第一个类型计算元素数量1, 然后根据rd和第二个类型计算元素数量2(rt保证1的个数相等),元素数量取最小值,然后进行正常的解包操作. 在这个例子中(如果两个算出来元素数目符合)是把rs1前一半解包的结果(sign extension)放到rd，rs1后一半解包的结果(sign extension)放到rt

需要保证类型可以unpack

否则为非法指令

可能发生extension等情况

vselect8.u8.u8 rd, rs1, rs2

vselect8由opcode确定

第一个u8由func4确定

第二个u8由func5确定

rs1为16位向量寄存器掩码

rs2为16位向量寄存器掩码

rd为16位向量寄存器掩码

描述: 选择转换指令，根据rs2寄存器组中的下标从rs1中选择读取数据,并转换类型,填充到rd掩码指定的寄存器组

依然是根据掩码从rs1和rd元素数量取最小值，然后根据在rs2中的下标(每个长8bit为)选择，下标长度不够为非法指令. 如果下标是(0, 0, 1, 1), rd和rs1相同就可以把(u8_0, u8_1, u8_2, u_3)转成(u8_0, u8_0, u8_1, u8_1)

这个指令覆盖了vcvt指令的能力，但是需要额外提供下标寄存器，延迟也稍长故vcvt单列一条指令

> vselect16版本下标每个长为16bit，经计算可以覆盖最大下标，不再列出

延迟: LUT + ALU + convert + move

vadd.u8 rd, rs1, rs2

vadd由opcode确定

u8由func4确定

funct5为0

rs1为16位向量寄存器掩码

rs2为标量寄存器标号

rd为16位向量寄存器掩码

描述: 向量标量加

需要保证rd, rs1掩码1数量相等，否则不合法

计算为每个元素+rs2对应标量，运算方式通过opcode和funct4确定

延迟: LUT与broadcast并行+ALU

> 其它向量标量运算指令不再列出

vvadd.u8 rd, rs1, rs2

vadd由opcode确定

u8由func4确定

funct5为0

rs1为16位向量寄存器掩码

rs2为16位向量寄存器掩码

rd为16位向量寄存器掩码

描述: 向量向量加

需要保证rd, rs1, rs2掩码1数量相等，否则不合法 计算为elemetwise运算，运算方式通过opcode和funct4确定
延迟: LUT+ALU

> 其它向量向量运算指令不再列出

3. 采用这套SIMD扩展指令，重新编写Lab4.1中的图像计算核心函数，并理论分析使用前后可以获得的最大指令数减少，以及可以获得的潜在性能提升。（20分）

核心函数

选取alpha混合+RGB2YUV作为核心函数(YUV2RGB特点和RGB2YUV相同)

原simd(avx2)

```
void process2(uint8_t alpha, uint8_t *rgb_buffer)

{

    char file_name[32]; // Increased buffer size for filename

    sprintf(file_name, "%d.yuv", static_cast<int>(alpha)); // Cast alpha for
sprintf

    // Memory map the YUV file

    void *yuv_mmap_addr = malloc(width * height * 3 / 2);
```

```
    clk.start();
    // Pointers to Y, U, V planes in the memory-mapped file
    uint8_t *y_plane_out = static_cast<uint8_t *>(yuv_mmap_addr);
    uint8_t *u_plane_out = y_plane_out + (static_cast<size_t>(width) * height);
    uint8_t *v_plane_out = u_plane_out + (static_cast<size_t>(width) * height /
4);

    // Pointers to R, G, B planes in the input RGB buffer
    uint8_t *r_plane_in = rgb_buffer;
    uint8_t *g_plane_in = r_plane_in + (static_cast<size_t>(width) * height);
    uint8_t *b_plane_in = g_plane_in + (static_cast<size_t>(width) * height);

    // --- AVX2 Setup ---
    __m256 alpha_factor_ps = _mm256_set1_ps(static_cast<float>(alpha) / 255.0f);
    __m256 const_zero_ps   = _mm256_setzero_ps();
    __m256 const_max255_ps = _mm256_set1_ps(255.0f);

    __m128 const_zero_ps128   = _mm_setzero_ps(); // For U/V processing (4 floats)
    __m128 const_max255_ps128 = _mm_set1_ps(255.0f);

    // Y coefficients (for 8 pixels)
    const auto& Y_coeffs = BT601::RGB2YUV[0];
    __m256 y_offset_ps = _mm256_set1_ps(Y_coeffs.offset);
    __m256 y_s1_r_ps   = _mm256_set1_ps(Y_coeffs.scale1); // Coeff for R'
    __m256 y_s2_g_ps   = _mm256_set1_ps(Y_coeffs.scale2); // Coeff for G'
    __m256 y_s3_b_ps   = _mm256_set1_ps(Y_coeffs.scale3); // Coeff for B'
```

```
    // U coefficients (for 4 pixels)
    const auto& U_coeffs = BT601::RGB2YUV[1];
    __m128 u_offset_ps128 = _mm_set1_ps(U_coeffs.offset);
    __m128 u_s1_r_ps128   = _mm_set1_ps(U_coeffs.scale1);
    __m128 u_s2_g_ps128   = _mm_set1_ps(U_coeffs.scale2);
    __m128 u_s3_b_ps128   = _mm_set1_ps(U_coeffs.scale3);

    // V coefficients (for 4 pixels)
    const auto& V_coeffs = BT601::RGB2YUV[2];
    __m128 v_offset_ps128 = _mm_set1_ps(V_coeffs.offset);
    __m128 v_s1_r_ps128   = _mm_set1_ps(V_coeffs.scale1);
    __m128 v_s2_g_ps128   = _mm_set1_ps(V_coeffs.scale2);
    __m128 v_s3_b_ps128   = _mm_set1_ps(V_coeffs.scale3);
    // Permutation indices for selecting R/G/B values for U/V calculation
    // Input: [px0, px1, px2, px3, px4, px5, px6, px7]
    // Output (lower 128 bits): [px0, px2, px4, px6]
    __m256i uv_select_indices = _mm256_set_epi32(0, 0, 0, 0, 6, 4, 2, 0);

    for (int i = 0; i < height; ++i) {
        for (int j = 0; j < width; j += 8) { // Process 8 pixels horizontally
            int y_plane_idx = i * width + j;
            int uv_plane_idx = (i / 2) * (width / 2) + (j / 2);

            // 1. Load 8 R, G, B uint8_t values
            __m128i r_vals_epi8 = _mm_loadu_si64(reinterpret_cast<const void*>
(r_plane_in + y_plane_idx));
            __m128i g_vals_epi8 = _mm_loadu_si64(reinterpret_cast<const void*>
```

```
(g_plane_in + y_plane_idx));

            __m128i b_vals_epi8 = _mm_loadu_si64(reinterpret_cast<const void*>
(b_plane_in + y_plane_idx));



            // 2. Convert uint8_t to float32 (8 values each)

            __m256i r_vals_epi32 = _mm256_cvtepu8_epi32(r_vals_epi8);

            __m256i g_vals_epi32 = _mm256_cvtepu8_epi32(g_vals_epi8);

            __m256i b_vals_epi32 = _mm256_cvtepu8_epi32(b_vals_epi8);



            __m256 r_vals_ps = _mm256_cvtepi32_ps(r_vals_epi32);

            __m256 g_vals_ps = _mm256_cvtepi32_ps(g_vals_epi32);

            __m256 b_vals_ps = _mm256_cvtepi32_ps(b_vals_epi32);



            // 3. Apply alpha scaling: scaled_color = color * (alpha / 255.0f)

            __m256 r_scaled_ps = _mm256_mul_ps(r_vals_ps, alpha_factor_ps);

            __m256 g_scaled_ps = _mm256_mul_ps(g_vals_ps, alpha_factor_ps);

            __m256 b_scaled_ps = _mm256_mul_ps(b_vals_ps, alpha_factor_ps);



            // 4. Y Calculation (for 8 pixels)

            __m256 y_calc_ps = y_offset_ps;

            y_calc_ps = _mm256_fmadd_ps(r_scaled_ps, y_s1_r_ps, y_calc_ps);

            y_calc_ps = _mm256_fmadd_ps(g_scaled_ps, y_s2_g_ps, y_calc_ps);

            y_calc_ps = _mm256_fmadd_ps(b_scaled_ps, y_s3_b_ps, y_calc_ps);



            // 5. Clamp Y, convert to uint8_t, and store

            y_calc_ps = _mm256_max_ps(const_zero_ps, y_calc_ps);    // Clamp min 0

            y_calc_ps = _mm256_min_ps(const_max255_ps, y_calc_ps);  // Clamp max
255
```

```
            __m256i y_clamped_epi32 = _mm256_cvtps_epi32(y_calc_ps); // Convert
float to int32 (truncates)



            // Pack 8x int32 to 8x uint8_t

            __m128i y_out_epi16_lane0 = _mm256_castsi256_si128(y_clamped_epi32);
     // Lower 4 int32s

            __m128i y_out_epi16_lane1 = _mm256_extracti128_si256(y_clamped_epi32,
1); // Upper 4 int32s

            __m128i y_packed_epi16 = _mm_packus_epi32(y_out_epi16_lane0,
y_out_epi16_lane1); // 8x signed int32 -> 8x unsigned int16

            __m128i y_packed_epi8  = _mm_packus_epi16(y_packed_epi16,
_mm_setzero_si128());   // 8x signed int16 -> 8x unsigned int8 (lower 8 bytes of
result)

            _mm_storeu_si64(y_plane_out + y_plane_idx, y_packed_epi8);



            // 6. U and V Calculation and Storage (conditional: only for even rows
'i')

            if (i % 2 == 0) {

                // Select scaled R', G', B' for pixels 0, 2, 4, 6 from the current
block of 8

                // These will be used to calculate 4 U and 4 V values.

                __m128 r_for_uv_ps =
_mm256_castps256_ps128(_mm256_permutevar8x32_ps(r_scaled_ps, uv_select_indices));

                __m128 g_for_uv_ps =
_mm256_castps256_ps128(_mm256_permutevar8x32_ps(g_scaled_ps, uv_select_indices));

                __m128 b_for_uv_ps =
_mm256_castps256_ps128(_mm256_permutevar8x32_ps(b_scaled_ps, uv_select_indices));



                // U calculation (for 4 pixels)

                __m128 u_calc_ps128 = u_offset_ps128;

                u_calc_ps128 = _mm_fmadd_ps(r_for_uv_ps, u_s1_r_ps128,
u_calc_ps128);

                u_calc_ps128 = _mm_fmadd_ps(g_for_uv_ps, u_s2_g_ps128,
u_calc_ps128);
```

```
                u_calc_ps128 = _mm_fmadd_ps(b_for_uv_ps, u_s3_b_ps128,
u_calc_ps128);



                // V calculation (for 4 pixels)

                __m128 v_calc_ps128 = v_offset_ps128;

                v_calc_ps128 = _mm_fmadd_ps(r_for_uv_ps, v_s1_r_ps128,
v_calc_ps128);

                v_calc_ps128 = _mm_fmadd_ps(g_for_uv_ps, v_s2_g_ps128,
v_calc_ps128);

                v_calc_ps128 = _mm_fmadd_ps(b_for_uv_ps, v_s3_b_ps128,
v_calc_ps128);

                // Clamp U and V values

                u_calc_ps128 = _mm_max_ps(const_zero_ps128, u_calc_ps128);

                u_calc_ps128 = _mm_min_ps(const_max255_ps128, u_calc_ps128);

                v_calc_ps128 = _mm_max_ps(const_zero_ps128, v_calc_ps128);

                v_calc_ps128 = _mm_min_ps(const_max255_ps128, v_calc_ps128);



                // Convert U, V floats to int32

                __m128i u_clamped_epi32 = _mm_cvtps_epi32(u_calc_ps128);

                __m128i v_clamped_epi32 = _mm_cvtps_epi32(v_calc_ps128);



                // Pack 4x int32 to 4x uint8_t for U

                __m128i u_packed_epi16 = _mm_packus_epi32(u_clamped_epi32,
_mm_setzero_si128()); // 4 int32 -> 4 uint16 (in lower half of 8 uint16s)

                __m128i u_packed_epi8  = _mm_packus_epi16(u_packed_epi16,
_mm_setzero_si128());   // 4 uint16 -> 4 uint8 (in lower half of 8 uint8s)

                // Store the 4 uint8_t U values using _mm_cvtsi128_si32 (gets the
lowest 32 bits as int)

                *reinterpret_cast<uint32_t*>(u_plane_out + uv_plane_idx) =
_mm_cvtsi128_si32(u_packed_epi8);
```

```
                // Pack 4x int32 to 4x uint8_t for V

                __m128i v_packed_epi16 = _mm_packus_epi32(v_clamped_epi32,
_mm_setzero_si128());

                __m128i v_packed_epi8  = _mm_packus_epi16(v_packed_epi16,
_mm_setzero_si128());

                // Store the 4 uint8_t V values

                *reinterpret_cast<uint32_t*>(v_plane_out + uv_plane_idx) =
_mm_cvtsi128_si32(v_packed_epi8);

            }

        }

    }

    clk.stop();

    FILE *file = fopen(file_name, "wb");

    fwrite(yuv_mmap_addr, 1, width * height * 3 / 2, file);

    fclose(file);

    free(yuv_mmap_addr);

}
```

计算主体simd指令数~1080 * 1920 / 8 * (23 + 24/2) = 9072000 (已忽略获取0向量寄存器)

我们的simd实现

> 采用伪代码，人工分析需要的simd指令数(因为比如寄存器分配和掩码编译器可以自动生成，人工书写太过麻烦，只需要分析确定我们的指令系统支持就行了) offset不表示程序变量，只是一个记号表示对应偏移 C不表示程序变量，只是一个记号表示对应常量 会直接使用operator表示向量运算，可以理解为重载过了(写的方便) 下面软件编程模型中的向量变量不是对应单个物理向量寄存器，而是对应虚拟的向量寄存器，可能编译器会是现成几个物理向量寄存器拼在一起

```
I/O Part
Constant Definitions
for (int i = 0; i < 1080; i++)
    for (int j = 0; j < 1920; j+= 64)
    {
        Calulate Offsets
        vr = vcvt_u8_f16(vload_u8(rplane + offset, 64)) * alpha_factor;
        vg = vcvt_u8_f16(vload_u8(gplane + offset, 64)) * alpha_factor;
        vb = vcvt_u8_f16(vload_u8(bplane + offset, 64)) * alpha_factor;
```

```
        vy = C + C * vr + C * vg + C * vb;

        vstore(yplane + offset, vcvt_f16_u8(vy));
        if (i % 2 == 0)
        {
            vu = C + C * vr + C * vg + C * vb;
            vv = C + C * vr + C * vg + C * vb;
            vstore(uplane + offset, vselect_f16_u8(vu, ...));
            vstore(vplane + offset, vselect_f16_u8(vv, ...));
        }
    }
I/O Part
```

分析: 我们一次处理64个元素，使用f16，共1024位，需要4个物理向量寄存器. 常驻vr, vg, vb三个虚拟向量寄存器共占用16个物理向量寄存器 临时向量占用4个（算完vy后就存vy，然后vy就不用了...所以4个够用） 所以编译器有能力把上面程序翻译成无需多步处理的汇编，源程序和汇编基本对应

总simd指令数:

- r,g,b向量加载混合: 9条
- vy计算存储: 8条
- vu,vv计算存储: 16条

总simd指令数~ 1080 * 1920 / 64 * (8 + 16/2) = 518400

我们指令数为原先的5.7% 潜在性能变为17.5x (假设CPI不变)

这主要是由:

1. 并行度8x

我们的f16贡献了额外的2倍并行度

我们的掩码处理器拼接设计实际上在这个例子中支持了1024位的虚拟向量，相比于先前256位有4倍提升

2. 精简的指令2.19x

先前每个循环体内有23 + 24/2 = 35条 而我们只需要8 + 16/2 = 16条

这是因为我们的select融合了shuffle和convert

可能的问题:

1. 复杂指令导致CPI降低
2. 硬件上支持掩码寄存器访问的复杂性
3. 内存带宽成为瓶颈