SIMD 扩展指令设计

姓名： 张煌昭
学号： 1400017707
院系： 元培学院
邮箱： zhang_hz@pku.edu.cn
手机： 17888838127

一．实验要求

1. 设计 SIMD 指令系统

在 LAB 4.1 的基础上，可以参考其他 ISA，比如 x86 或 RISC-V 设计一套 SIMD 指令系统。需要确定指令操作类型，寻址方式，数据格式，编码格式等，具体的指令要求如下：

支持 256 位 SIMD 指令；支持 8 个 256 位 SIMD 指令专用寄存器；支持 8/16/32 位 pack/unpack 计算；支持加/减/乘法；支持饱和运算；支持必要的数据传输。

为该 SIMD 指令系统设计一套汇编注记符，以便编写汇编程序。

2. 使用该 SIMD 扩展指令重写 LAB 4.1 核心函数并分析可能的性能提升

二．任务分析

对 LAB 4.1 中的任务进行分析如下。以 YUV2RGB 转换为例，其他任务均有类似特性。

在 LAB 4.1 中，将 YUV420 格式图片转为 RGB888（ARGB8888）格式图片的算法描述如下。

```
输入：unsigned char 类型数组 YUV[]，其长度为 W*H*1.5，存放的是 YUV420 格式图片
输出：unsigned char 类型数组 RGB[]，其长度为 W*H*3，存放的是 RGB888 格式图片
算法：YUV2RGB(W, H, YUV[])
YUV2RGB(W, H, YUV[]):
1.  for i from 0 to W * H - 1, do
2.      y = YUV[i]
3.      u = YUV[cal_u_idx(i)]
4.      v = YUV[cal_v_idx(i)]
5.      r = cal_r(y, u, v)
6.      g = cal_g(y, u, v)
7.      b = cal_b(y, u, v)
8.      RGB[3 * i] = r
9.      RGB[3 * i + 1] = g
10.     RGB[3 * I + 2] = b
11. return RGB[]
```

使用 SIMD 指令，并对其进行循环展开的优化，算法描述如下。

输入：unsigned char 类型数组 YUV[]，其长度为 W*H*1.5，存放的是 YUV420 格式图片；int 类型 K 为循环展开的层数
输出：unsigned char 类型数组 RGB[]，其长度为 W*H*3，存放的是 RGB888 格式图片
算法：YUV2RGB_SIMD(W, H, YUV[], K)

```
YUV2RGB_SIMD(W, H, YUV[]):
1.   for i from 0 to W * H - 1, do
2.      for k from 0 to K, do
3.         y[k] = YUV[i+k]
4.         u[k] = YUV[cal_u_idx(i+k)]
5.         v[k] = YUV[cal_v_idx(i+k)]
6.      r = cal_r_SIMD(y, u, v)      // 使用 SIMD 指令并行计算多个值
7.      g = cal_g_SIMD(y, u, v)
8.      b = cal_b_SIMD(y, u, v)
9.      for k from 0 to K, do
10.        RGB[3 * (i + k)] = r
11.        RGB[3 * (i + k) + 1] = g
12.        RGB[3 * (i + k) + 2] = b
13. return RGB[]
```

发现在上述代码中有很长一部分都在进行数据传送的工作，因为 YUV420 格式中只有 Y 可以直接从内存装入 XMM 寄存器中，而 U 和 V 由于采样的原因，都必须经过转换才可以放入，此处可以针对设计 SIMD 指令，设计专用的数据传输指令。

此外，由于 AVX 以前的 SIMD 指令并没有 reduce 操作，即将一个向量里的所有元素进行运算，结果放入第一个元素之中，在进行运算时很不方便，因而考虑在设计的 SIMD 指令中也需要有 reduce 操作。

三．mySIMD 指令系统设计

1. mySIMD 指令系统概述

在 mySIMD 指令系统中，将内存视为一维线性的数组，按字节编号随机寻址。指令寻址仿照 RISC-V ISA，采用间接寻址方式，offset(%r)表示的地址为 regfile[%r] + offset，注意此处寄存器为寄存器堆中的整点寄存器，下面将要提到的浮点寄存器不能用于寻址。

整点指令与 RISC-V ISA 一致，使用 32 个 64 位寄存器等。

mySIMD 指令专用于 8 个 256 位的 SIMD 寄存器，记为 rs0 到 rs7，各 SIMD 寄存器地位对等，都是通用寄存器。每个 SIMD 寄存器可以在低位装载或运算 8 位、16 位、32 位、64 位的整形或浮点数，也可以按向量形式装载或运算 32 个 8 位或 16 个 16 位整形数，或 8 个 32 位或 4 个 64 位整形数或浮点数。

仿照 AVX 功能，mySIMD 指令提供向量的（packed）算术逻辑运算和低位标量的（unpacked）算术逻辑运算，此外也提供向量的 reduce 运算，数据传输方面除了提供 AVX 的标准的数据传输方式，还提供 YUV420, RGB888 和 ARGB8888 格式的像素级的数据传输。

2. mySIMD 各指令功能

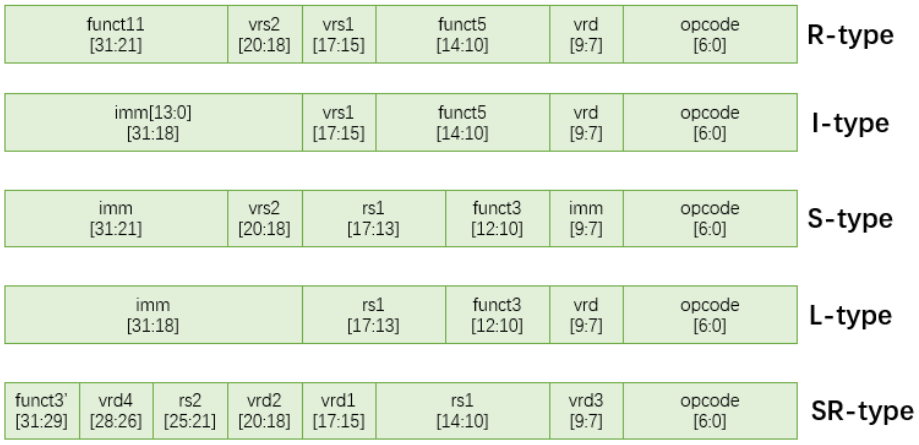
将 mySIMD 指令按上述功能归类，指令系统中需要有如下表功能所示的指令。

表. MySIMD 各类指令的功能				
Type	**Mnemonic**	**Description**	**Data type**	**Data Length**
Data Transfer (Reg2Mem / Mem2Reg / Reg2Reg)	VULD	Unpacked load (load to lower n-bit)	Int / FP	B / H / W / D
	VUST	Unpacked store (store lower n-bit)	Int / FP	B / H / W / D
	VPLD	Packed load (load to every element)	Int / FP	B / H / W / D
	VPST	Packed store (store every element)	Int / FP	B / H / W / D
	VMV	Move data from one register to another	Int / FP	B / H / W / D
	LDYUVPIX	Load YUV420 pixels and pack them into 3 mySIMD registers	Int	B
	SDYUVPIX	Store YUV420 pixels and pack them into 3 mySIMD registers	Int	B
	LDRGBPIX	Load RGB888 pixels and pack them into 3 mySIMD registers	Int	B
	SDRGBPIX	Store RGB888 pixels and pack them into 3 mySIMD registers	Int	B
	LDARGBPIX	Load ARGB8888 pixels and pack them into 4 mySIMD registers	Int	B
	SDARGBPIX	Store ARGB8888 pixels and pack them into 4 mySIMD registers	Int	B
Arithmetic / Logical Operation (Packed / Unpacked)	VUAO	Unpacked arithmetic operations. Use RISC-V arithmetic operation mnemonic to substitute “AO” to generate a “real” mnemonic. Note that shifts are in this sector.	Int / FP	B / H / W / D
	VPAO	Packed arithmetic operations, which are similar to “VUAO”, but are all packed version. Note that shifts are in this sector.	Int / FP	B / H / W / D
Bit-wise Logical Operation	VLO	Logical operations. Use “AND”, “OR”, “NOT” etc to substitute “LO” to generate a “real” mnemonic. Note that there is no need to distinguish bit-wise logical operation.	Int	B / H / W / D
Reduced Operation	VREO	Reduce operations. Use “RED” + RISC-V arithmetic / logical operation mnemonic to substitute “REO” to generate a “real” mnemonic.	Int / FP	B / H / W / D

3. 指令编码

仿照 RISC-V 指令的编码格式进行编码，mySIMD 指令包含 R 型，I 型，S 型，以及一个新添加的指令类型，记作 SR 型（super-R-type）。

由于 mySIMD 使用 8 个 256 位的 vector 寄存器，和 32 个通用整点寄存器，因而每个 vector 寄存器需要 3 位进行编码，通用寄存器仍需要 5 位进行编码，各个类型的域如下图所示。



R 型指令包括 SIMD 寄存器间的算数/逻辑运算指令。R 型指令使用相同的 opcode；指令接受两个 SIMD 寄存器作为源，将运算结果放回一个 SIMD 寄存器；通过 funct5 和 funct11 来判断该指令是 R 型中的具体的哪一个指令，具体地，可以通过 funct11 确定运算类型（如确定是加法），通过 funct5 确定运算的具体设置（如确定运算是向量按字节无符号饱和加法），每一个设置可以通过 1 位的 01 来判断，由于运算类型有限，因而 funct11 中实际会有很多位为没有意义的“空位”，可以考虑扩展时使用。

I 型指令包括 SIMD 寄存器和立即数间的算数/逻辑运算指令。I 型指令 opcode 各不相同；其具有一个源 SIMD 寄存器和一个目标 SIMD 寄存器，另一个数据源来自于 imm 域，imm 域由 R 型中的 vrs2 和 funct11 两个域合并而成；通过 opcode 来判断该指令的运算类型，通过 funct5 来确定运算的具体设置，其中 opcode 的运算类型编码与 R 型指令的 funct11 域运算类型编码一致，以便译码。

S 型指令为将 SIMD 寄存器的内容按向量或按标量存放在通用寄存器基地址和偏移量表示的内存位置的 ST 指令。S 型指令 opcode 固定，通过 funct3 确定存放的类型和按向量存还是按标量存；指令接受一个 SIMD 寄存器作为数据源，一个通用寄存器作为地址基址，将通用寄存器与立即数偏移量有符号加后得到内存地址进行存放。

L 型指令与 S 型指令对应，为 LD 指令，按向量或按标量读取在通用寄存器基地址和偏移量表示的内存位置的内容放入 SIMD 寄存器中。L 型指令 opcode 固定，通过 funct3 确定读取类型和按向量读还是按标量读，编码格式与 S 型 funct3 相同；指令接受一个 SIMD 寄存器作为数据目的寄存器，一个通用寄存器作为地址基址，将通用寄存器与立即数偏移量有符号加后得到内存地址进行读取。

SR 型指令为一类特殊指令，该类指令要求根据两个通用寄存器计算内存地址，之后写至多 4 个 SIMD 寄存器，由于寄存器堆同时只能写一个寄存器，因而 SR 型指令实际上是分别写四个寄存器，可以通过一系列 S 和 L 指令完成，可以视为伪指令。

表. MySIMD 指令实例	
Instruction	**Description**
VULDB vrd, offset(rs1) VUSTB vrs2, offset(rs1)	Load / store one byte scalar data from / to memory. Other similar instructions are VULDH, VUSTH, etc.
VPLDB vrd, offset(rs1) VPSTB vrs2, offset(rs1)	Load / store 32 bytes vector data from / to memory. Other similar instructions are VPLDH, VPSTH, etc.
VMVS2R rs1, vrd VMVR2S vrs2, rs1	Move data from lower bits of SIMD register to integer register or from integer register to lower bits of SIMD register.
VPADDUBS vrd, vrs1, vrs2 VPADDSBS vrd, vrs1, vrs2 VPADDUB vrd, vrs1, vrs2 VPADDSB vrd, vrs1, vrs2	Different types of vector addition. “P” means vector operation. “U/S” for unsigned / signed, “B” for byte, and “S/(none)” for saturated / unsaturated. Also, FP addition is written as VPADDF, VPADDD, etc.
VPMULHUB vrd, vrs1, vrs2 VPMULLSB vrd, vrs1, vrs2	Different types of vector multiplification. “H/L” is for get-high-bits / get-low-bits.
VPSRLB vrd, vrs1, vrs2	Arithmetic right shift.
VPREDADDUBS vrd VPREDADDSBS vrd VPREDADDUB vrd VPREDADDSB vrd	Different types of vector reduced sum. “RED” is for reduced. “ADD” can be substitute with other reasonable operations, such as “MAX”, “MIN”, etc.
LDYUVPIX vrd1, vrd2, vrd3, rs1, rs2 SDYUVPIX rs1, rs2, vrs1, vrs2, vrs3	SR YUV420 pixel vector load / store. Each SIMD register loads / stores 16 pixel y/u/v. Function equivalent to loading / storing Y first, and up-sampled U and V then.
LDRGBPIX vrd1, vrd2, vrd3, rs1, rs2 SDRGBPIX rs1, rs2, vrs1, vrs2, vrs3	SR RGB888 pixel vector load / store. Each SIMD register loads / stores 16 pixel r/g/b. Function equivalent to loading storing R first, and up-sampled G and B then.
LDARGBPIX vrd1, vrd2, vrd3, vrd4, rs1, rs2 SDARGBPIX rs1, rs2, vrs1, vrs2, vrs3, vrs4	SR ARGB8888 pixel vector load / store. Each SIMD register loads / stores 16 a/r/g/b.

4. mySIMD 指令表

本次 LAB 4.2 可能会使用到的所有 mySIMD 指令如上表所示。上表仅为实例指令，其余指令可以根据功能以及如下描述进行添加。

四．使用 mySIMD 实现 LAB 4.1 核心程序

1. 实现 YUV2RGB

以 YUV2RGB 为例，其余的 YUV2ARGB，ARGB2RGB，RGB2YUV 等均为类似形式。

在 LAB 4.1 中使用的 MMX，SSE2，AVX2 的程序均为第一节中算法的实现，额外的，为了使用定点运算，在实现中将 0-1 浮点类型的参数乘 64 后装载入 XMM/YMM 寄存器，之后按照 16 位（H）进行向量操作，最后使用在向内存的 RGB 区域写入时，再将值右移 6 位回复原值。

使用 mySIMD 实现 YUV2RGB 的核心部分如下

```
Loops and other codes …
LDYUVPIX vr0, vr1, vr2, r23, r24 #  以 r23 为基地址，取下标 r24 开始的 YUV 像素分量
VPLDH vr3, 0x0(r27) #  将 Y 在 R 计算中的权值装入 vr3
VPLDH vr4, 0x20(r27) #  将 U 在 R 计算中的权值装入 vr4
VPLDH vr5, 0x40(r27) #  将 V 在 R 计算中的权值装入 vr5
VPMULLUH vr0, vr0, vr3   #  计算 R 分量
VPMULLUH vr1, vr1, vr4
VPMULLUH vr2, vr2, vr5
VPADDUHS vr6, vr0, vr1 #  放于 vr6 中
VPADDUHS vr6, vr6, vr2
VPLDH vr3, 0x0(r28) #  将 Y 在 G 计算中的权值装入 vr3
VPLDH vr4, 0x20(r28) #  将 U 在 G 计算中的权值装入 vr4
VPLDH vr5, 0x40(r28) #  将 V 在 G 计算中的权值装入 vr5
VPMULLUH vr0, vr0, vr3   #  计算 G 分量
VPMULLUH vr1, vr1, vr4
VPMULLUH vr2, vr2, vr5
VPADDUHS vr7, vr0, vr1 #  放于 vr7 中
VPADDUHS vr7, vr7, vr2
VPLDH vr3, 0x0(r29) #  将 Y 在 B 计算中的权值装入 vr3
VPLDH vr4, 0x20(r29) #  将 U 在 B 计算中的权值装入 vr4
VPLDH vr5, 0x40(r29) #  将 V 在 B 计算中的权值装入 vr5
VPMULLUH vr0, vr0, vr3   #  计算 B 分量
VPMULLUH vr1, vr1, vr4
VPMULLUH vr2, vr2, vr5
VPADDUHS vr0, vr0, vr1 #  放于 vr0 中
VPADDUHS vr0, vr0, vr2
SDRGBPIX r25, r24, vr6, vr7, vr0 # #  以 r25 为基地址，存下标 r24 开始的 RGB 像素分量
Loops and other codes …
```

整段程序中间计算部分与 LAB 4.1 基本相同，区别在于读取和写会内存的部分。也可以

将以上对 RGB 分量的计算写为内层循环的形式，但为了编写方便，没有将其写为循环。

2. 分析

考虑到程序运行时间大部分在于内存读写，因而根据 Amdahl's law，访存为主要部分，在 mySIMD 设计中针对这一部分进行优化；而对于运算部分，基本仿照 AVX2 的设计思路进行设计，没有作很多改动。

由于上述原因，mySIMD 并不会引起运算部分代码动态执行指令数减少，也基本不会使得这一部分性能提升，但对于读取 YUV420 格式的像素的 YUV 分量和写入 RGB888 格式的像素的 RGB 分量，mySIMD 各只需要 1 条指令便可完成。

最坏情况下，将 mySIMD 中的 SR 型指令视为伪指令，其实现方法为组合三/四个分量的 LD 指令，这与 LAB 4.1 的实现完全相同，指令数目和性能不会下降。

而较好情况下，将 SR 指令视为需要执行多个周期的指令，由于不存在相关，因而可以充分利用流水线，完成快速装入/写回多个像素的 YUV/RGB 分量的操作。这种情况下，指令数由 LAB 4.1 的约 20 条降至 1 条，而运行时间为该流水线下的最短时间。假设 LAB 4.1 中的 YUV 装载是最坏的情况，即处处存在冒险，那么 mySIMD 的 LDYUVPIX 指令的执行时间将是 LAB 4.1 的约 1/3。

当然，设计这样的 LD 和 ST 指令是代价极高，也极不自然的。但在这一任务下，若不考虑成本和实现问题，这样的做法是可以最大可能最大限度提升性能的。