Technical Document

Technical Document 补充

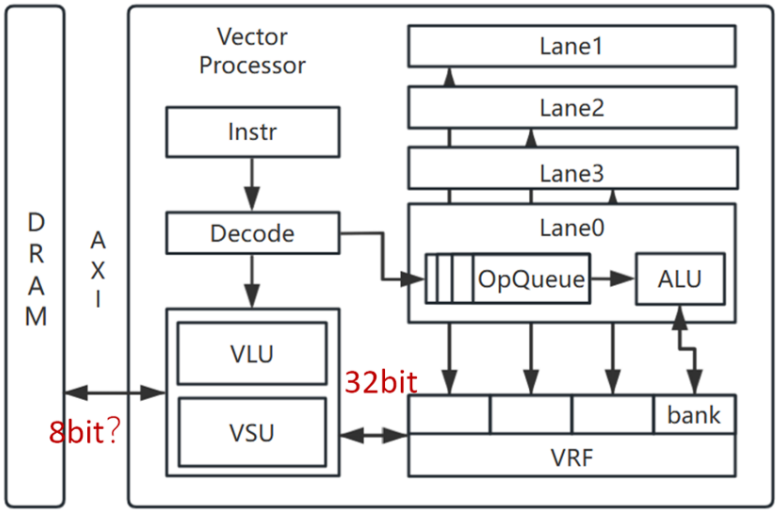
Dataflow 各阶段时序&耗时分析

Technical Document

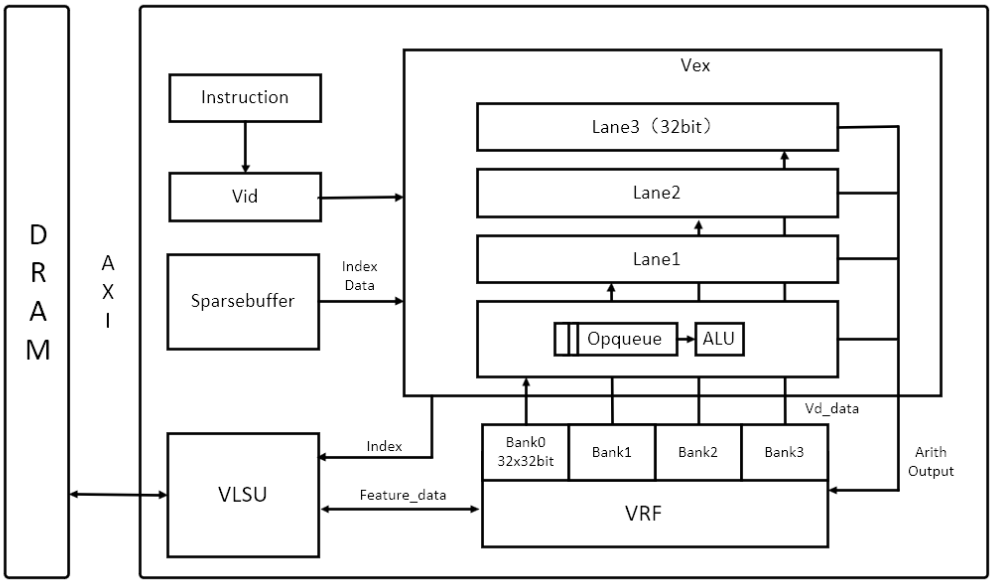
架构更新

架构方面，新增了一个 sparse buffer，用于存储 csr 格式的矩阵，也就是稀疏矩阵（左矩阵）。其他变化不大，主要是内部执行逻辑的变化。

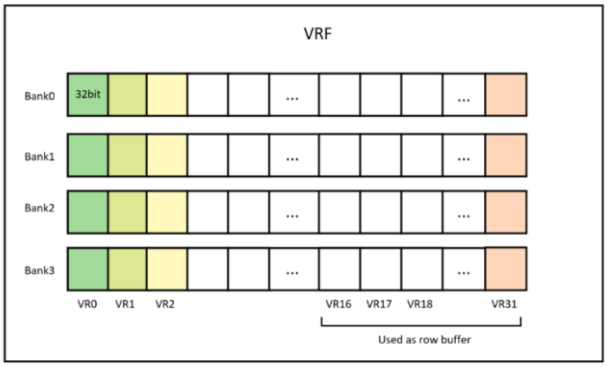
原先架构:



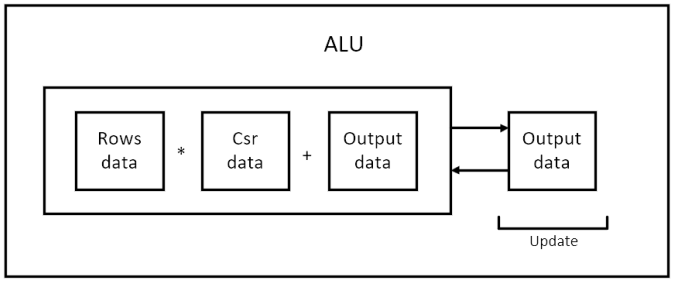
更新后架构：



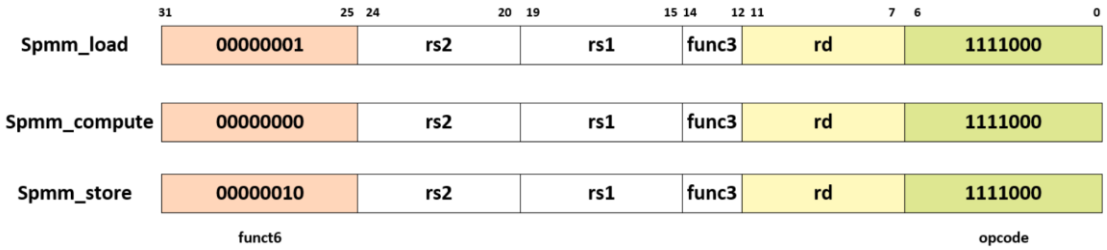
VRF（规定 16~31 寄存器用来存计算所用行，从 vlsu 取出）：



ALU（inside VEX）（本质是乘加计算）：



新增三条指令：




```
// 自定义向量指令的操作码定义和不同功能
// 以下所有判断条件要加opcode和funct6放在一起以便于后续增加指令
localparam logic [6:0] V_OPCODE_SPMM      = 7'b1111000;   // 新增，用于执行spmm计算
localparam logic [5:0] V_FUNCT6_COMPUTE   = 6'b000000;    // opcode下的compute功能
localparam logic [6:0] V_FUNCT6_LOAD      = 6'b000001;    // opcode下的Load功能
localparam logic [6:0] V_FUNCT6_STORE     = 6'b000010;    // opcode下的store功能
```

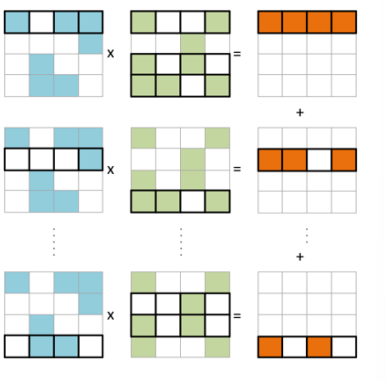
Opcode 相同，根据 funct6 区分三条指令。

介绍 spmm 数据格式和位置：

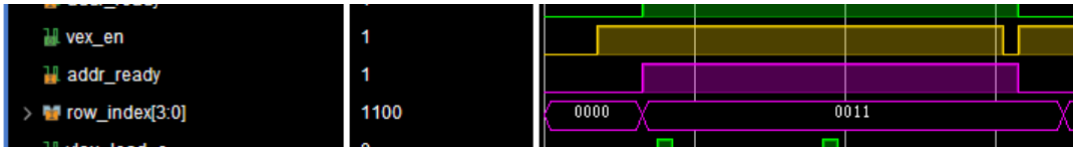
实现的是两个 4x4 矩阵的相乘，元素位宽是 32bit。其中一个矩阵用 csr 格式存储在 core 内的 buffer，另外一个矩阵按照从左到右，从上到下的顺序以元素级别存储在 dram 当中。

介绍 spmm 数据流（以算一行结果为例）：

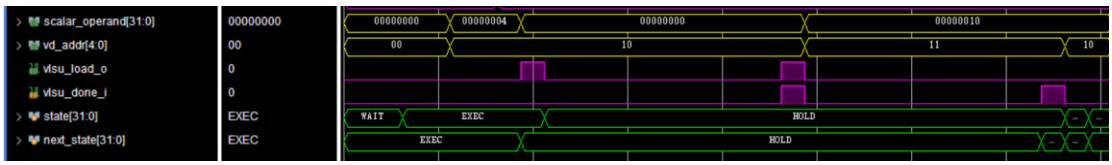
Row_wise 矩阵计算：



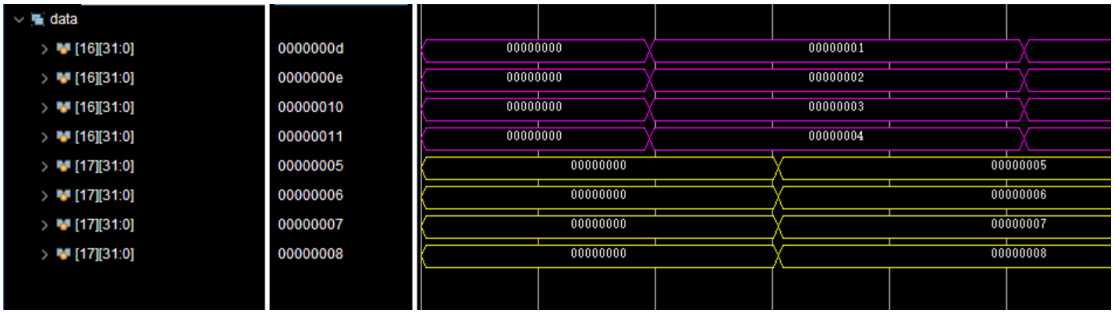
首先，执行 Spmm_load 指令。vex 取更新一行需要用的 csr 格式数据，vlsu 使能 vex，拉高 vex_en 信号，vex 经过一个周期算出所需要用的行，然后输出回 vid，具体信号是 row_index,addr_ready 这两个信号。（比如需要第 1，2 行，则 row_index 的值为 0011）。



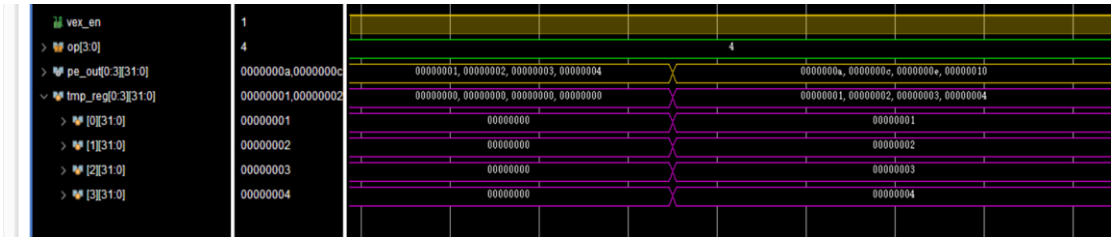
接着，vid 根据输入的两个数据，更新要取的数据地址。数据在内存当中的存储地址是已知的，一个地址存 32bit 数据，所以所有的右边矩阵一共占了 16 个内存地址。由于数据是一行一行的取，vid 根据第一个要取的数据是第一行，更新第一行第一个数据的基地址（在 vid 当中是 scalar_operand 信号），然后传给 vlsu，使能 vlsu，vlsu 根据该基地址，四个周期，取四个数，即一行，存到 vrf 的第 16 个向量寄存器当中。（我们规定第 16-31 向量寄存器用来存取的行，信号为 vd_addr，可变）。vlsu 存完第一个行之后，向 vid 发出 vlsu_done 的信号，接着，vid 根据第二个要取的数据也就是第二行，将 scalar_operand 更新为第二行第一个数据的基地址，传给并使能 vlsu，vlsu 取出行数据放入第 17 个向量寄存器当中。存完所有的数据之后，vid 根据最后一个 vlsu_done 的信号拉高 spmm_load_done 信号，表示 load 已完成。



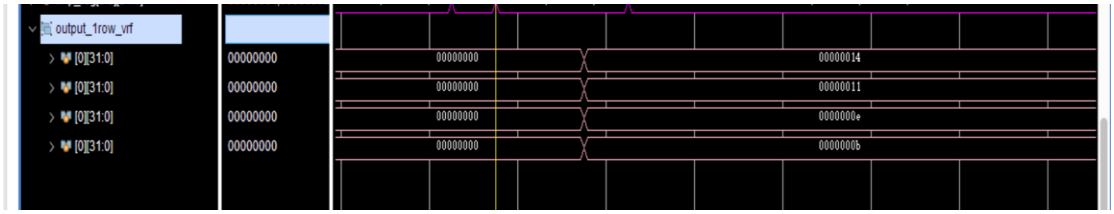
根据下图可以看出，vrf 当中已经存好了相关数据。



在这之后，执行 Spmm_compute 指令。Compute 阶段分为四部分，第一部分是从 vrf 当中取存好的行数据。第二部分是将系数矩阵的 data 和行进行相乘。第三部分是将第二部分乘法计算出的结果和先前的计算结果累加在一起，第四部分则将计算结果存到 vrf 的第 1 个向量寄存器当中，完成一行的计算。具体来说，指令会首先使能 vex，vex 根据 spmm_indices 获取当前非零元素的序号，选择对应的稠密矩阵行数据，每一个 lane 负责一个元素的计算，lane 执行乘法计算，结果（pe_out）和 tmp_reg 相加并存回 tmp_reg。（本质上是乘加指令）。接着 spmm_compute_indice_base 指向下一行，进行下一行的乘加计算。



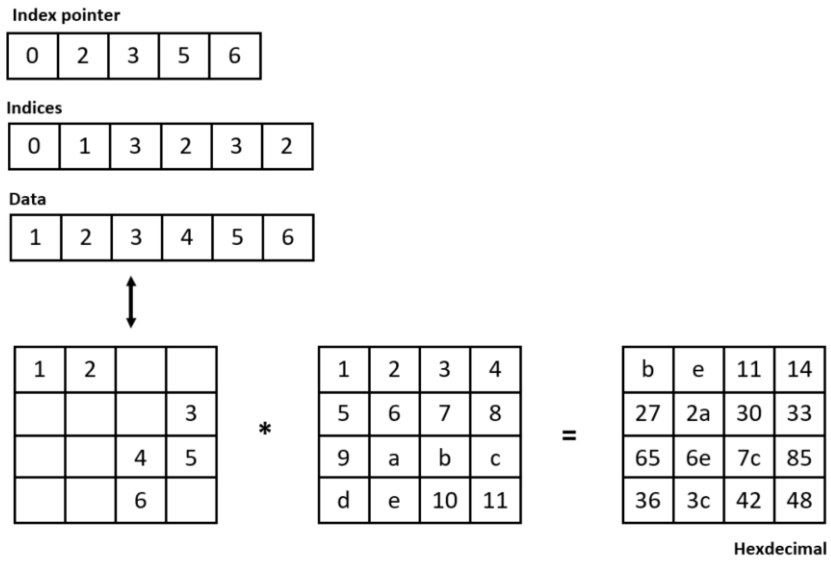
若所有行计算完成，将 tmp_reg 拼起来成 128bit，存到 vrf 的向量寄存器当中，拉高 spmm_compute_done。表示计算完成



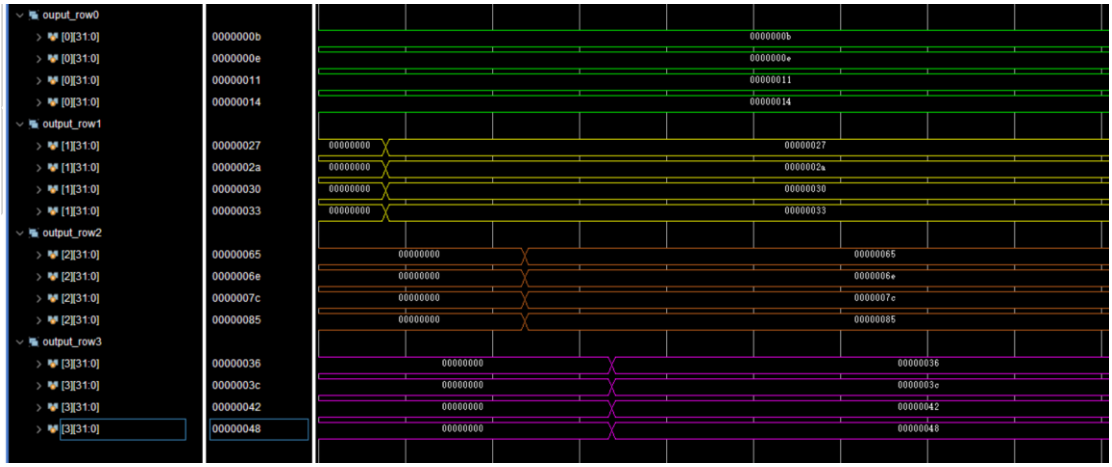
最后，执行 Spmm_store 指令。最后将存到 vrf 当中的数据存回外部内存当中。在指令的 rd 当中选择向量寄存器（数据），store 指令的标量操作数是存回外部内存的地址。值得注意的是，我们没有写如果计算结果是全零就将一段存储空间置零的 code。从图数据的角度出发，左边矩阵不存在全为零的行，即图大概率不会出现完全没有边的点，对于大型图来说，孤立点的存在对整体 gnn 的计算影响很小。

矩阵计算举例

以下是目前仿真所用的数据，和理论输出结果。



在波形图当中，将 vrf 四个 bank 的 0,1,2,3 向量寄存器分组，以下是计算结果。

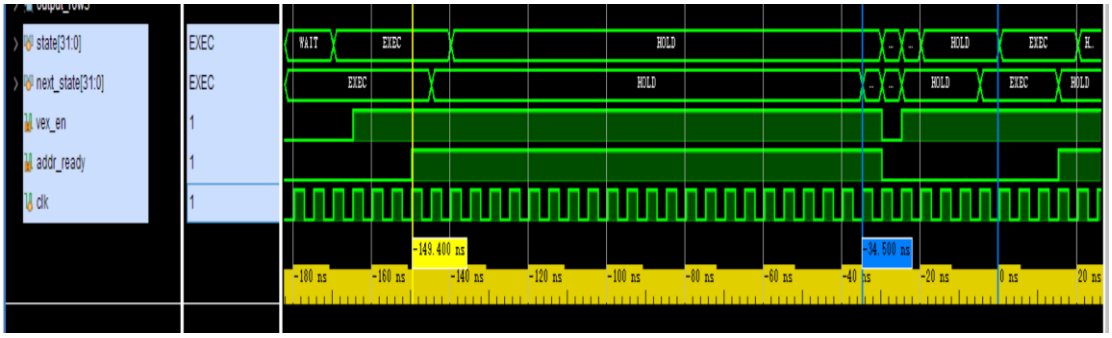


未来优化方向

1.增加 HDNbuffer，可以划分一部分向量寄存器作为存储高度节点对应的行，vid 每

次先从该 buffer 当中找行，没有找到再去通过 vlsu 从外部内存当中找对应的行。由于从 buffer 当中取数据相对从外部快很多，而且要处理的 dataset 是符合 power-law 模型的（少数节点占多数边），意味着每次计算要取的行重复度很高，所以该方法可以显著提高计算效率

2.增加一个 ctrl 逻辑，实现边取边算。现有的处理器是将所有相关的行全部取出来再进行计算，实际可以做到边算边取。由下图可以，取（2 个数据）需要 23 个时钟周期，算并存到 vrf 当中需要 7 个时钟周期。



备注：如果现有代码跑不通或者跑通但少了一行输出的结果，可以考虑在指令最前加一行 vset 指令，注意该[]当中的数字，再重新跑仿真。

```
//spmm compute 32'b000000_10000_00000_000_00000_1111000;
//                                  funct6   rs2   rs1  funct3  rd  opcode
external_mem[32'h0000_0400 >> 2]  =  32'b000000_00010_00000_111_00001_1010111; //vset 跑不出来用该指令
external_mem[32'h0000_0404 >> 2]  =  32'b000001_00000_00000_000_10000_1111000; //spmm load
external_mem[32'h0000_0408 >> 2]  =  32'b000000_10000_00000_000_00000_1111000; //spmm compute 存v0
external_mem[32'h0000_040C >> 2]  =  32'b000001_00000_00000_000_10000_1111000; //spmm load
external_mem[32'h0000_0410 >> 2]  =  32'b000000_10000_00000_000_00001_1111000; //spmm compute 存v1
external_mem[32'h0000_0414 >> 2]  =  32'b000001_00000_00000_000_10000_1111000; //spmm load
external_mem[32'h0000_0418 >> 2]  =  32'b000000_10000_00000_000_00010_1111000; //spmm compute 存v2
external_mem[32'h0000_041C >> 2]  =  32'b000001_00000_00000_000_10000_1111000; //spmm load
external_mem[32'h0000_0420 >> 2]  =  32'b000000_10000_00000_000_00011_1111000; //spmm compute 存v3
//external_mem[32'h0000_0420 >> 2]  =  32'b000001_00000_00000_110_00010_0100111; //load v2
// external_mem[32'h0000_0424 >> 2]  =  32'b000001_00000_00000_110_10100_0100111; // vse32.v v20, 地址 0x24
```

Technical Document 补充

vex 模块补充

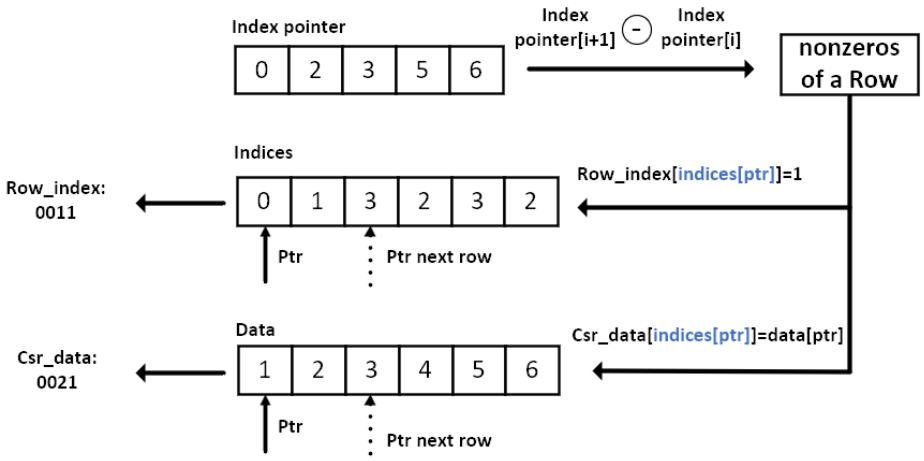
vex 接收输入的 csr 格式的 4x4 稀疏矩阵数据，在 vex 中分成三份，分别代表 data, indices 和 index。

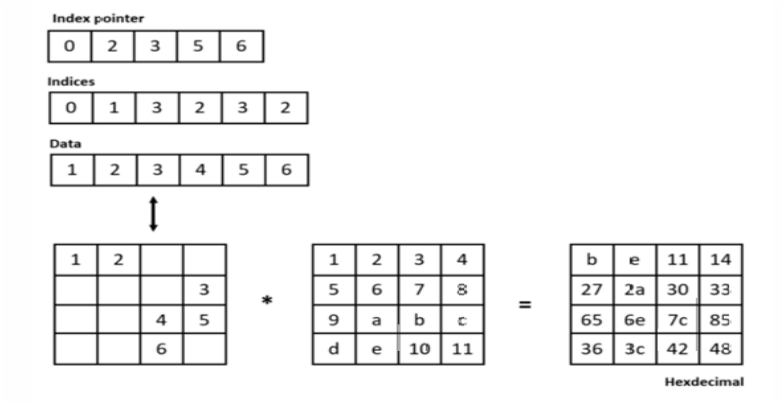
在 LOAD 指令中（确定一行数据长度以及解释 row_index 值）

由于输入的稀疏矩阵数据定为 4x4 的大小, 因此 index 中的有效数据数量固定为 5。vex 中利用一个小的状态机对 index 中的数据进行后项减去前项，判断出对应行中有没有有效数据，以及有效数据的个数是多少，保存在 spmm_row_data_num 寄存器当中。

在接下来的组合逻辑中，利用一个计数器 spmm_data_ptr 来从前面得到的稀疏矩阵的 data 和 indices 中找出这一行数据对应的数值，分别存进 spmm_row_data 和 spmm_row_index 中, 之后输出 addr_ready 信号说明本次 Load 需要的数据准备好了，同时更新 spmm_data_ptr，spmm_data_ptr = spmm_row_data_num + spmm_data_ptr，方便后续指向下一行的 indices。

由于不确定一行有多少数据，因此定义 spmm_row_data 的大小为 128bit,刚好为 4 个元素的大小，通过 spmm_indices 中记录的 indice 值将稀疏矩阵的一行还原出来，之后分成 4 块，存进 csr_data 这个数组里面。在后面使用的时候同样利用 indice 中记录的值取出对应元素进行计算。spmm_row_index 定义 4bit 的宽度, 4 个位分别代表 4 行，例如 index[0]拉高，说明需要第 1 行的数据。以下是画图说明：





以上两张图中的稀疏矩阵为例子，第一行稀疏矩阵一共有 2 个元素，则计算出来的结果为：

spmm_row_data_num 为 2，代表当前行共有 2 个非零数，需要取两行。

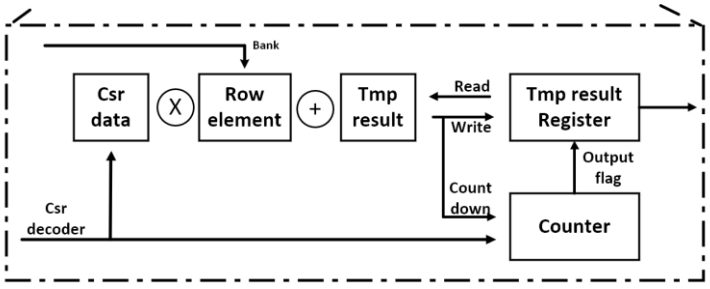
row_index = 4‘b0011,代表需要取第一行第二行的数据。

csr_data[3] = 32'h0　csr_data[2] = 32'h0　csr_data[1] = 2　csr_data[0] = 32'h1，后续 vex 计算用。

另外设置了一个 load_matrix_num 用来对这个矩阵进行计数，每 load 完一行就-1，等清零之后重置上面的所有 ptr 和数据。

在 COMPUTE 指令中（计算细节，乘加过程）

vex 收到由 vrf 输入的在 load 指令中取出来的数据，每个周期一行，将这一行的 4 个元素分别放进 4 个 lane，同时在 vex 内部利用一个时序逻辑每个周期更新 spmm_compute_indice_ptr，利用这个指针以及基地址从前面得到的 csr_data 数组中取出对应的元素复制 4 份放进 4 个 lane 中进行计算，之后将计算的结果与 tmp_reg 中的数值进行累加并存入 tmp_reg。经过 4 个周期，4 个 tmp_reg 中的结果就是需要输出的行中的 4 个元素，之后更新基地址 spmm_compute_indice_base <= spmm_compute_indice_base + spmm_row_data_num。下图是单个 lane 当中的计算过程



同样也设置了一个 compute_matrix_num 用来对矩阵进行计数，每计算完一行就-1，

清零之后重置指针和数据。

问题回答

1. csr 一行数据的 csr 还是 4x4 矩阵的 csr?

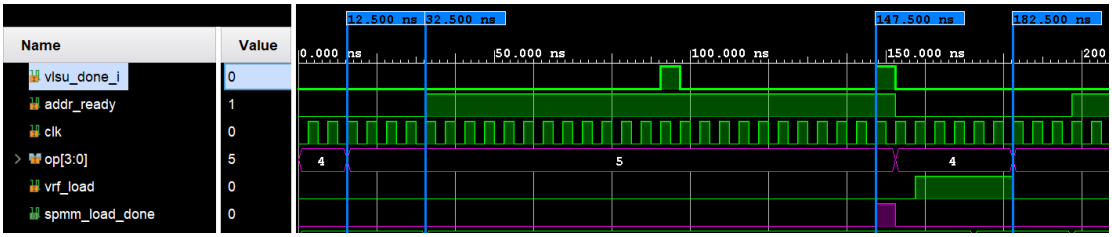
目前设计中 vex 一开始得到的 csr 数据输入就是一整个矩阵的数据，在后面计算过程中再根据 index 和 indices 分成每一行的数据。

2.有无 pipeline?

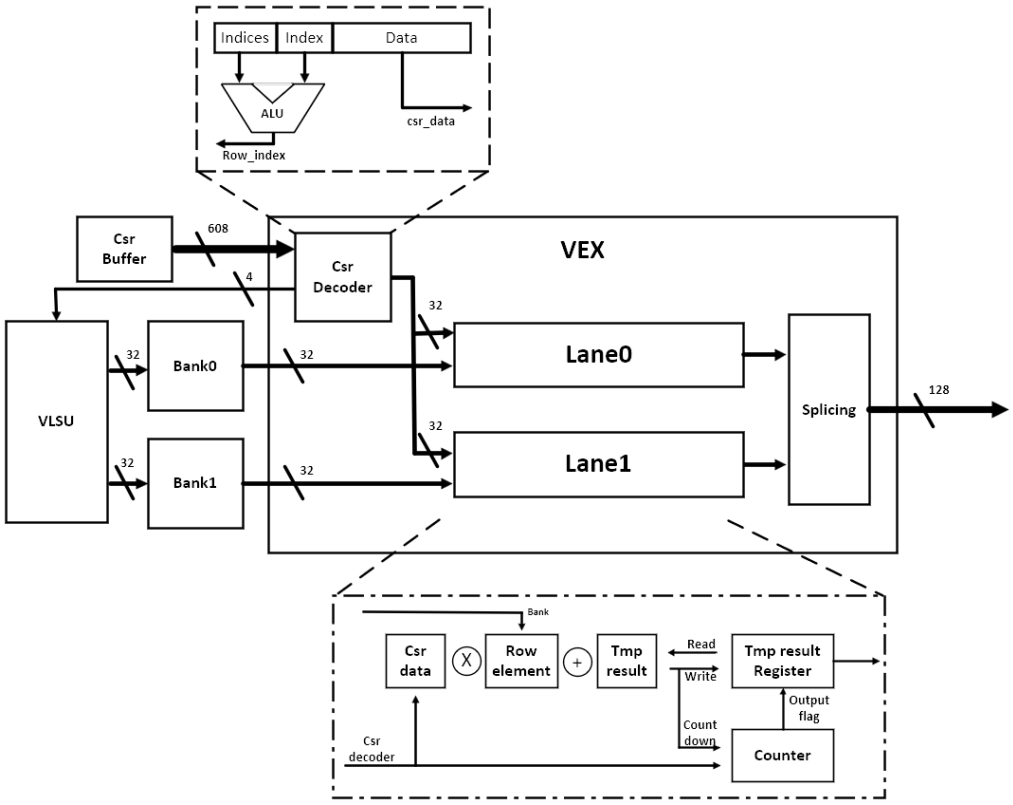
目前设计不能做出流水线，没有办法边算边存，只能实现先 load 后 comp，因为 comp 指令需要前面 load 指令中得到的 row_data_num，csr_data 等数据。

3. 多少周期完成 spmm_load，计算地址花了多少周期等等

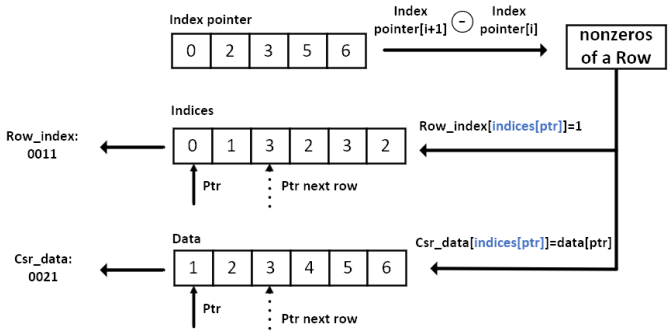
完成整个 load 指令（从取到指令到 load_done 信号拉高）大概需要 27 个周期，其中计算出 row_index 需要 4 个周期。完成 load+comp 一次需要大约 34 个周期。



4.VEX 细节图:



5.csr decoder 地址计算过程（画图表示）:



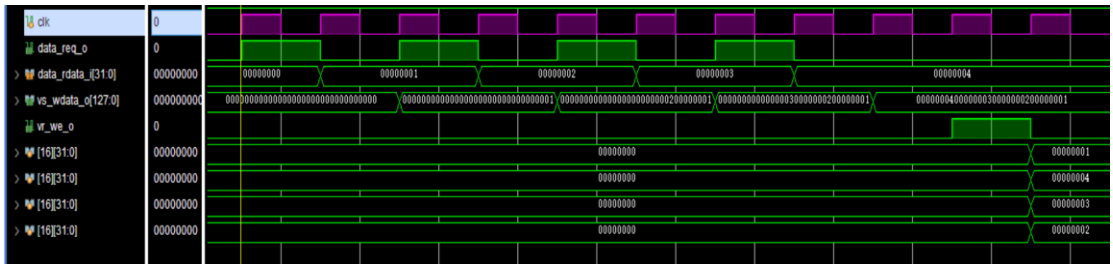
VLSU 模块补充

LOAD 指令

在 vid 收到 row_index 和 addr_ready 两个信号之后，处理信号，确定第一个目标行的第一个数据的外部内存的地址 scalar_operand 和要存放的向量寄存器位置 vd_addr。接着拉高 vlsu_load_o，传给 vlsu 地址和位置两个数据。Vlsu 内部有状态机，收到使能信号之后状态跳到 Load_cycle，该状态拉高 data_req_o,发起读数据请求，从

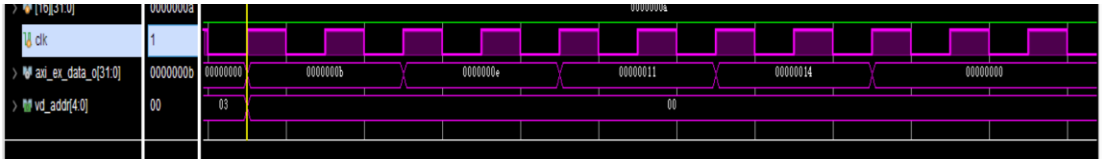
外部接收数据输入给内部 temporary_reg 模块。temporary_reg，vlsu 内部的接收数据并整理数据的模块，主要负责将四周期，一周期 32bit 的数据拼接起来成 128bit 然后输出。处理完数据后状态机跳到 Load_wait，更新地址，跳回 Load_cycle，开始加载下一个 32bit 数据，循环四次，直到加载一行所有数据完成，也就是 temporary_reg 拼成了 128bit 数据，跳到 Load_final，该状态拉高 vr_we_o，该信号为向量寄存器写使能，将 temporary_reg 的 128bit 输出连到 vs_wdata_o（128bit）上面，作为 vrf 的输入数据，还有 vid 来的 vd_addr 输入给 vrf，作为写入目标向量寄存器的地址。另外，该状态还拉高 vlsu_done_o 输出给 vid，表示 load 结束。

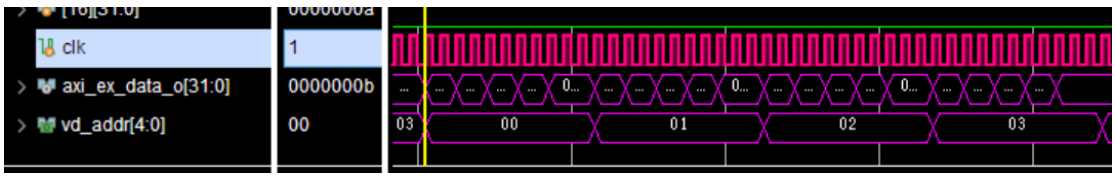
请求信号拉高之后，一个周期外部内存读取数据放到 data 总线上，一个周期 vlsu 内部处理输出信号。以两个周期为单位，一个单位读取一个 32bit 的数，四个单位读取完全部数据，共需要 8 个周期。接着下一个周期拉高寄存器写使能，一个周期后写入到对应向量寄存器当中。



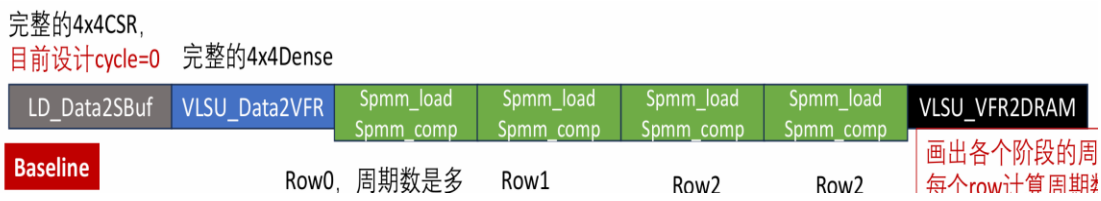
Store 指令

状态机跟 load 指令类似。从 vrf 接收 128bit 的数据作为 vlsu 的输入（向量寄存器的地址由指令的 rd 决定），将 128bit 数据拆分，根据 store_cycles_cnt 分次进行输出，一共四次，每次输出 32bit。外部数据地址我们设置是从 0 开始顺序写入（原先是根据 store 指令的标量操作数进行指定地址写入，由于 spmm 不需要指定写入地址，于是我们取消标量操作数，从地址 0 开始，顺序往下写）。以下表示从 0 寄存器读取数据，写出数据。写出一个元素需要两个周期，一行需要 8 个周期。



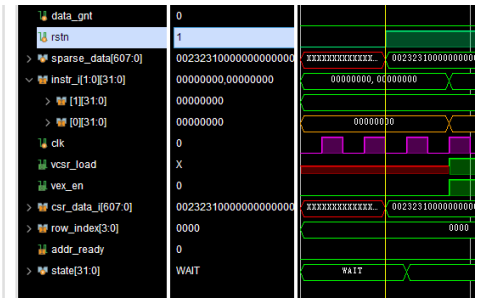


Dataflow 各阶段时序&耗时分析



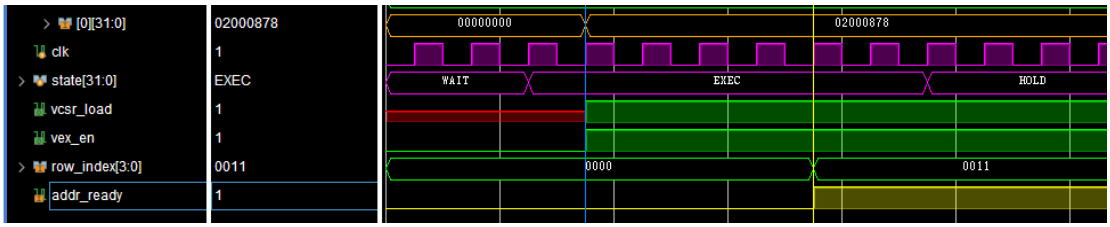
注：当前设计直接 LD_Data2SBuf 是不准的，由于未写完 dma（在写），我们直接将 csr 数据放在 tb 里面，load 拉高直接传入 vex 当中。实际应该是从 dram 通过 dma 传入 sparse buffer 再到 vex。这一部分先不分析，后续会补上。

LD_Date2SBuf (待更新)：

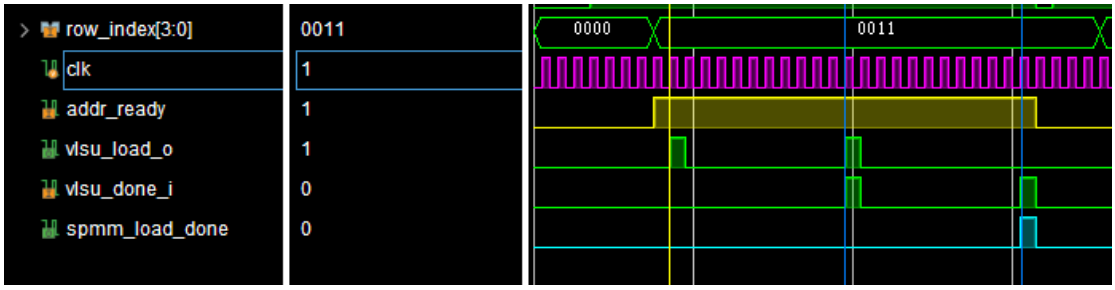


仿真当中的数据 sparse_data 在 rstn 当中准备好，rstn 拉高后传到 vex 当中的 csr_data_i。1.5 周期（跟仿真中 rstn 变化的时间有关）。

VLSU_Date2VRF (SPMM_LOAD NO.1)

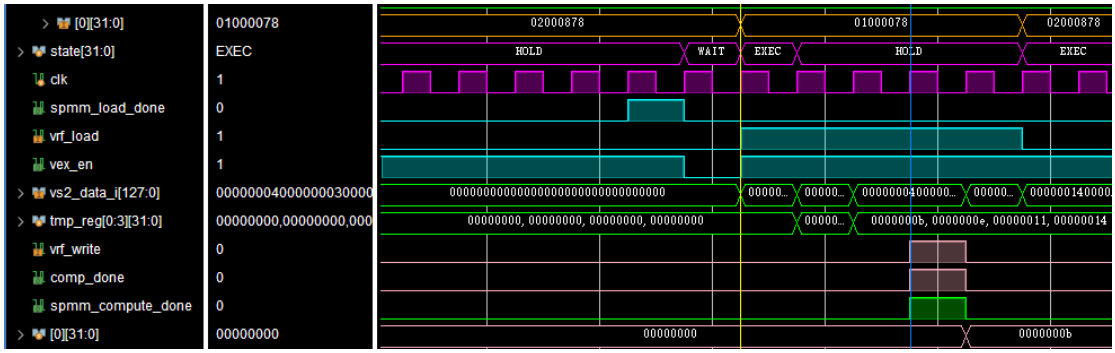


在 load 指令进来之后（第四周期开始），立刻使能 vex 开始计算要更新第一个行结果要用到的矩阵的行位置（row_index）。花费四周期计算完。



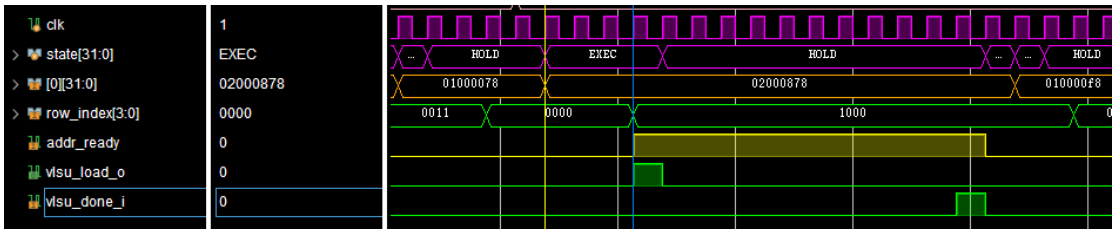
Vex 将 addr_ready 信号传给 vid，v 拉高 vlsu 取数。第一次拉高取第一行，花费 11 个周期。第二次拉高取第二行，花费 11 个周期(取行花费的时间是一样的)。取数一共花费 22 个周期。接着拉高 spmm_load_done。

SPMM_COMPUTE (NO.1)



接着读取下一个指令，EXEC 阶段 compute 指令进来，瞬间拉高 vex_en 和 vrf_load（组合逻辑），数据也在 vrf_load 拉高的瞬间进来到 vex，开始计算，经过一个周期将结果存到 tep_reg 当中。然后 load 下一行数据，经过一个周期（乘加计算）将结果存到 tem_reg 当中。然后在下一个周期拉高 comp_done 信号。在下一个周期存到 vrf_bank 当中。接着状态到 exec 指令开始读取下一个 load 指令计算第二行结果。在从 vrf 当中 load 数据，load 需要一个周期，乘加计算需要一个周期，write 到 vrf 当中需要一个周期。有个小流水线是计算该行可以 load 下一行。这样计算两行只需要 3 个周期（原本是 4）。这样 load+计算需要（2*n-1 周期）+write 一个周期。

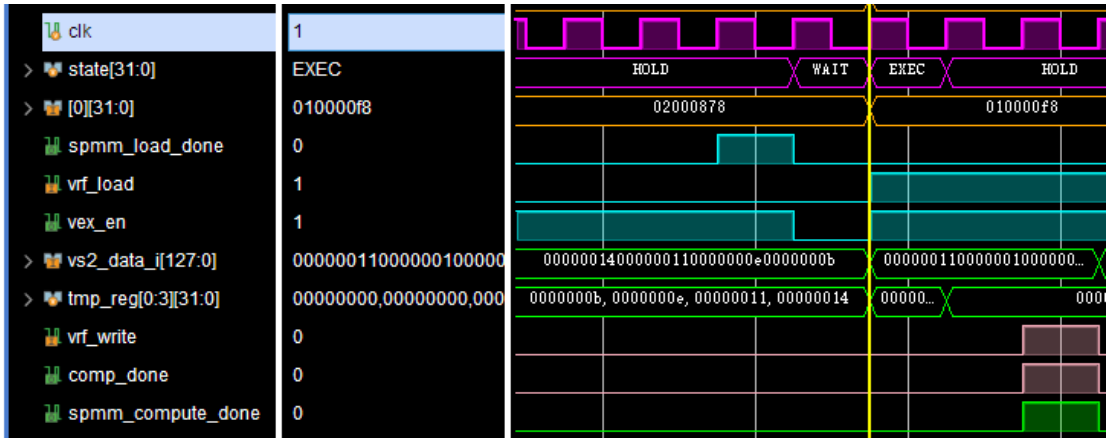
VLSU_Date2VRF (SPMM_LOAD NO.2)



Write 到 vrf 当中后一个周期接收 load 指令。拉高 vex 计算 row_index，花费 3 个周期。(后面计算都是用 3 个周期，因为比第一个 exec 少了个 wait 所以少了一个周期)。然后拉高 vlsu，取行数据，花费 11 个周期，

（稀疏矩阵第二行就一个数据，所以取一次数据）跟之前一样，最后拉高 spmm_load_done。然后开始计算。（取一行数据是 11 周期，n 行是 11n 周期）

SPMM_COMPUTE (NO.2)



接着读取下一个 compute 指令，从 vrf 加载数据花了一周期，然后计算花了一周期。计算完毕后拉高 vrf_write，存到 vrf_banks 也花一周期。

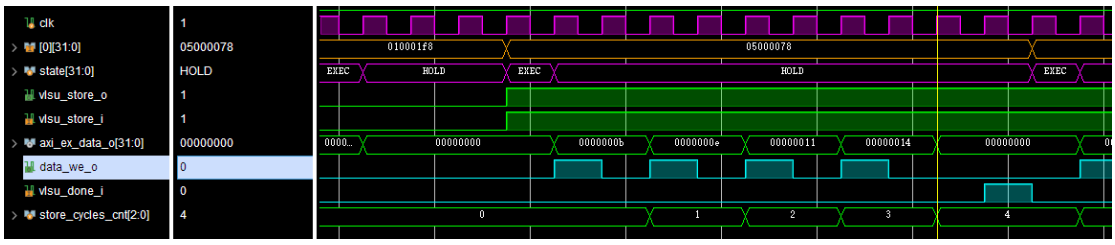
VLSU_Date2VRF (SPMM_LOAD NO.3)

SPMM_COMPUTE (NO.3)

VLSU_Date2VRF (SPMM_LOAD NO.4)

SPMM_COMPUTE (NO.4)

SPMM_STORE(NO.1)



在经过四次 load 和四次计算之后，数据已经全部存到 vrf 当中，四行数据在对应的 VR0~VR3。接着是四条 load 指令，一条指令会输出四个数，分四次输出（32*4）。在 EXEC 阶段读取到 store 的指令后，花费一个周期读到数据，拉高 dram 写使能，然后下一个周期写入到 dram 当中。在下一个周期读下一个数。四个数的读和写一共持续 8 周期。接着下一周期拉高 vlsu_done 信号。一行写入 dram 完成。接着读取下一个 store

指令。

SPMM_STORE(NO.2)

SPMM_STORE(NO.3)

SPMM_STORE(NO.4)

各个阶段周期数整理

Sp2buffer	load1	mac1	load2	mac2	load3	mac3	load4	mac4
0	26 (4+22)	4	14 (3+11)	2	25 (3+22)	4	14 (3+11)	2
Store1	Store2	Store3	Store4	总计				
11	11	11	11	135				

备注：每条指令结束状态机会跳回 wait，再读下一个指令到 exec，exec 阶段开始执行下一条。也就是指令之间 wait 会消耗一个周期，上表没有严格把读指令的周期算进去。