Fast Track Architecture Proposal

Vector zip (Zvzip) extension

Originally contributed by XXX YYY, please direct questions to XXX.YYY@rivosinc.com
This proposed extension is following the Fast Track Architecture Extension process.

Summary

The Zvzip extension defines a set of instructions used to combine lanes from multiple vector registers with important patterns that allow for efficient specialisation. They complement mechanisms for reshuffling data (such as vlseg, vrgather, vslideup/down) and remove the need for passing through memory, changing the LMUL, or setting up masks. This addresses use cases from application areas as varied as signal processing, high-throughput AI, and high-performance computing.

Motivation and use cases

The Zvzip instructions help in the reordering of structured data in vector registers. Examples of where this is useful include:

- Reading real / imaginary components from a stream of complex numbers (occurs frequently in communication systems)
- Unpacking of data structures of small powers of two (e.g. RGBA tuples)
- Transposing of small matrices, where different rows are held in vector registers. This in turn can be used iteratively to transpose larger matrices, and is of utility for the smallest and largest matrices we can have

The RVV (RISC-V Vector extension) v1.0 standard provides instructions for moving data in registers, but the instructions are general and can require reading data from memory, constructing additional input data in registers, or restricting instruction-level parallelism. Specific instructions allow for hardware to provide more optimal implementations. Experiments run on simulation show that (depending on the hardware and the algorithm) that realistic speedups when compared to using existing RVV v1.0 instructions are between 10% and 110%. In particular, unpacking of structure, such as RGBA data gets the most advantage. An assumption is that the throughput is higher, and the latency is lower, compared to vrgather LMUL > 1. The advantage of these instructions over vrgather or segmented load and store increases as the ratio between CPU clock speed and memory frequency increases.

Suitability for Fast Track Extension Process

This proposed extension meets the Fast Track criteria: it consists of a small set of instructions which can improve the throughput of operations from a wide range of applications. The structured reshuffling of data held in registers is a ubiquitous requirement from large-scale linear algebra, to small-scale linear algebra, from high-performance computing to communication systems, deep learning and more.

Proposed Specification

All instructions have a vector destination register, and two vector source registers. The instruction encodings have the following form

31 26	25	24 20	19 15	14 12	11 7	6 0
funct6	m	rs2	rs1	funct3	rd	opcode
6	1	5	5	3	5	7

Where opcode = 1011011 and funct3 = 000. Instructions with the following mnemonics and funct6 encoding are proposed. vzipeven (funct6 = 001100), vzipodd (011100), vzip2a (000100), vzip2b (010100), vunzip2a (001000), vunzip2b (011000). All instructions are well defined for LMUL >= 1. In all cases, vl gives the maximum number of destination elements that are written. The source registers may overlap with each other, but not with the destination register, i.e. vd != vs1 and vd != vs2.

vzip{even,odd}

These instructions take packed structures with two fields, which we call \`a\` and \`b\`, and interleaves all \`a\` (\`b\`) elements in the destination, in the vzipeven (vzipodd) instruction, respectively. The operation is shown in the pseudo-sail code below

```
let num_elem = VL; /* VL <= VLMAX = VLEN * LMUL / SEW */
let vs2_offset = if (funct6 == vzipeven) then 0 else 1;
let vs1_offset = if (funct6 == vzipeven) then -1 else 0;
foreach (i from 0 to (num_elem - 1)) {
    let tmp = match (i % 2) {
        0 => vs2_val[i + vs2_offset],
        1 => vs1_val[i + vs1_offset],
    }
    result[i] = if (mask_val[i] == 0) then vd_val[i] else tmp;
}
```

It is simple enough to define this operation for LMUL < 1, but that would cause inconsistencies with the definitions of the other instructions in this proposal. The instruction is therefore restricted to be defined for LMUL >=1.

An example of where vzip{even,odd} can be used is in matrix transposition, where the values in a vector register can be thought of as rows in a matrix on input, and the columns of a matrix on output. An example of a 4x4 matrix of 32-bit elements with vlen = 128 is given below. This can be iterated over with different inputs to perform the matrix transpose of larger matrices.

```
C/C++
// Assume that v1, v2, v3, v4 hold the data for a 4x4 matrix
// LMUL=1, SEW=32, vl = 4, ta, ma to start with
vsetivli zero, 4, e32, m1, ta, ma
vzipeven.vv v5, v1, v2
vzipodd.vv v6, v1, v2
vzipeven.vv v7, v3, v4
vzipodd.vv v8, v3, v4

vsetivli zero, 2, e64, m1, ta, ma
vzipeven.vv v1, v5, v7
vzipeven.vv v2, v6, v8
vzipodd.vv v3, v5, v7
vzipodd.vv v4, v6, v8
```

To undo the vzip{even,odd} operation, we need only vzip{even,odd} operations.

vunzip2{a,b}

An operation that frequently occurs in practice is the need to unpack data structures with multiple fields, such as rgba data, or complex numbers. The vunzip2{a,b} instructions can be used to perform these operations, and the pseudo-sail code is given below.

```
let num_elem_per_reg = get_num_elem(0, SEW);
let half_ind = num_elem_per_reg * LMUL / 2;
let num_elem = VL; /* VL <= VLMAX = VLEN * LMUL / SEW */
let offset = if (funct6 == vunzip2a) then 0 else 1;
foreach (i from 0 to (num_elem - 1)) {
    let ind = (2 * i) % half_ind;
    let tmp = match (floor(i / half_ind)) {
        0 => vs2_val[ind + offset],
        1 => vs1_val[ind + offset],
    }
    result[i] = if (mask_val[i] == 0) then vd_val[i] else tmp;
}
```

At LMUL=1, the operation of vunzip2{a,b} takes either the even or odd numbered lanes from vs1, packs them into the first half of vd, and packs the even or odd numbered lanes from vs2 into the second half of vd. At LMUL > 1, this operation acts as though the register group is a vector of length LMUL * vlen. The instruction is not defined for LMUL < 1, as the logic about which element from which register to read from is entirely different from the cases where LMUL >= 1. This would increase the complexity of the instruction, and would reduce any gains seen against something like a vrgather. The instructions here aim to give specialized and optimized performance, so we avoid generalizing in cases where performance could suffer.

An example of where this instruction is useful is in the complex multiplication of numbers. The example below shows how to perform the pairwise multiplication of complex numbers, unpacking the data with an unzip2{a,b}. We assume 128-bit vlen and 32-bit SEW

C/C++
```
// v1, v2 have complex numbers [a0r, a0i, a1r, a1i], [a2r, a2i, a3r, a3i]
// v3, v4 have complex numbers [b0r, b0i, b1r, b1i], [b2r, b2i, b3r, b3i]
// We want to perform operations to compute (akr * bkr - aki * bki) + j(akr
* bki + aki * bkr)
vsetivli zero, 4, e32, m1, ta, ma
vunzip2a.vv v5, v1, v2  // a real
vunzip2b.vv v6, v1, v2  // a imaginary
vunzip2a.vv v7, v3, v4  // b real
vunzip2b.vv v8, v3, v4  // b imaginary

// v9 and v10 will have real and imaginary components of the multiplication
vfmul.vv v9, v7, v5
vfnmsac.vv v9, v8, v6
vfmul.vv v10, v8, v5
vfmacc.vv v10, v7, v6
```
To pack the data again, the vzip2{a,b} instructions from the next section are required. Note that this operation is moving from an array-of-structs to a struct-of-arrays. We can unpack data quite simply for 2, 4 and 8 element structs by increasing SEW between calls to vunzip2{a,b}. This is analogous to segmented loads, which load structured data from memory. The segmented load is heavy-handed as a method for unpacking in the case that data already exists in registers, as a write to memory would be required before a read. Other options, such as a vrgather with a fixed pattern are also possible to achieve this interleave, but this approach is not vector-length agnostic (the pattern depends on the number of elements in a vector). Use of vrgather also increases setup time and register pressure, due to the extra registers populated and used by vrgather indices. Masked vslideup/down can also be used, but the masking introduces tight dependencies between instructions, meaning instruction-level parallelism is restricted.

vzip2{a,b}

The vzip2{a,b} instructions are useful for repacking structured data, where different fields are stored in different registers. The following pseudo-sail code describes the operation of the instruction:

```
let num_elem_per_reg = get_num_elem(0, SEW);
let num_elem = VL; /* VL <= VLMAX = VLEN * LMUL / SEW */
let offset = if (funct6 == vzip2a) then 0 else (num_elem_per_reg * LMUL /
2);
foreach (i from 0 to num_elem - 1) {
    let ind = floor(i / 2);
    let tmp = match (i % 2) {
        0 => vs2_val[offset + ind],
        1 => vs1_val[offset + ind],
    }
    result[i] = if (mask_val[i] == 0) then vd_val[i] else tmp;
}
```

At LMUL=1 the vzip2a (vzip2b) instruction takes the elements from the low (high) half of vs1 and vs2, respectively, and writes alternating elements to the start of vd. At LMUL > 1, this operation acts as though the register group is a vector of length LMUL * vlen. The instruction is not defined for LMUL < 1.

An example of where this is useful is in packing rgba components back into a single stream. The example below shows how to perform the packing of 16-bit rgba data. We assume 128-bit vlen and 16-bit SEW.

```C/C++
// v1 red, v2 green, v3 blue, v4 alpha
vsetivli zero, 8, e16, m1, ta, ma
vzip2a.vv v5, v1, v2            // [r0, g0, r1, g1, r2, g2, r3, g3]
vzip2b.vv v6, v1, v2            // [r4, g4, r5, g5, r6, g6, r7, g7]
vzip2a.vv v7, v3, v4            // [b0, a0, b1, a1, b2, a2, b3, a3]
vzip2b.vv v8, v3, v4            // [b4, a4, b5, a5, b6, a6, b7, a7]

vsetivli zero, 4, e32, m1, ta, ma
vzip2a.vv v1, v5, v7            // [r0, g0, b0, a0, r1, g1, b1, a1]
vzip2b.vv v2, v5, v7            // [r2, g2, b2, a2, r3, g3, b3, a3]
vzip2a.vv v3, v6, v8            // [r4, g4, b4, a4, r5, g5, b5, a5]
vzip2b.vv v4, v6, v8            // [r6, g6, b6, a6, r7, g7, b7, a7]
```

This operation is analogous to segmented stores, but the destination of the operation ends up in registers rather than memory. When the result is required in registers, the segmented store is an expensive way to pack the data, as it needs to be written to and then read from memory again. Other options for packing are to use a vrgather with a fixed pattern, but again this is not vector-length agnostic, and it incurs extra setup cost and increases register pressure. Another option is masked vslideup/down, but that adds in dependencies between instructions to limit instruction-level parallelism.

Conclusion

The instructions in this fast-track proposal give a way to implement the shuffling of data in registers in a way that occurs frequently in practice. Although there are several ways to achieve the same results with the current standard, the methods all have some bottleneck, whether this be dependence on memory, switching behaviour dependent on the vector length, or adding unnecessary dependencies between instructions. The interleaving of data occurs so frequently in many different application areas, that it is useful to have specialized instructions that allow for a more optimal implementation of these transformations.

Appendix: Spike code

C/C++

```
//
#define VZIP_IMPL(regexp, idxexp) \
do { \
require_extension(EXT_VZIP); \
require_align(insn.rd(), P.VU.vflmul); \
require_align(insn.rs2(), P.VU.vflmul); \
require_align(insn.rs1(), P.VU.vflmul); \
require(insn.rd() != insn.rs2() && insn.rd() != insn.rs1()); \
require(P.VU.vflmul >= 1.0); \
require_vm; \
VI_AGNOSTIC_SUPPORT_LOOP_BASE \
  reg_t reg = (regexp); /*true iff rs2 (the first register)*/ \
  reg_t idx = (idxexp); \
  switch (sew) { \
  case e8: { \
       if (inactive_element || tail_element) { \
       P.VU.elt<uint8_t>(rd_num, i, true) = AGN_VAL; \
       } else { \
       P.VU.elt<uint8_t>(rd_num, i, true) = reg ? P.VU.elt<uint8_t>(rs2_num,
idx) : P.VU.elt<uint8_t>(rs1_num, idx); \
       } \
       break; \
  } \
  case e16: { \
       if (inactive_element || tail_element) { \
       P.VU.elt<uint16_t>(rd_num, i, true) = AGN_VAL; \
       } else { \
       P.VU.elt<uint16_t>(rd_num, i, true) = reg ?
P.VU.elt<uint16_t>(rs2_num, idx) : P.VU.elt<uint16_t>(rs1_num, idx); \
       } \
       break; \
  } \
  case e32: { \
       if (inactive_element || tail_element) { \
       P.VU.elt<uint32_t>(rd_num, i, true) = AGN_VAL; \
       } else { \
       P.VU.elt<uint32_t>(rd_num, i, true) = reg ?
P.VU.elt<uint32_t>(rs2_num, idx) : P.VU.elt<uint32_t>(rs1_num, idx); \
       } \
       break; \
  } \
  default: { \
       if (inactive_element || tail_element) { \
       P.VU.elt<uint64_t>(rd_num, i, true) = AGN_VAL; \
       } else { \
      P.VU.elt<uint64_t>(rd_num, i, true) = reg ?
P.VU.elt<uint64_t>(rs2_num, idx) : P.VU.elt<uint64_t>(rs1_num, idx); \
      } \
      break; \
  } \
  } \
VI_AGNOSTIC_SUPPORT_LOOP_END; \
} while (0)

//vzipeven:
VZIP_IMPL(((i % 2) == 0), ((i % 2) == 0 ? i : i - 1));
//vzipodd:
VZIP_IMPL(((i % 2) == 0), ((i % 2) == 0 ? i + 1 : i));
//vzip2a:
VZIP_IMPL(((i % 2) == 0), (i / 2));
//vzip2b:
VZIP_IMPL(((i % 2) == 0), ((i / 2) + (P.VU.exec_vlmax / 2)));
//vunzip2a:
VZIP_IMPL((i < (P.VU.exec_vlmax / 2)), (2 * i) % (P.VU.exec_vlmax));
//vunzip2b:
VZIP_IMPL((i < (P.VU.exec_vlmax / 2)), (((2 * i) % (P.VU.exec_vlmax)) + 1));
```

Appendix: Motivation for design decisions

There were several design decisions made that reasonable architects may have made differently. Here, we explain why we came to these decisions. A key factor was to limit the complexity of the datapath required for these instructions, to enable more of them and at a lower cost than required for vrgather. It was also designed to be able to reuse some of the logic needed for segment loads and stores.

Instruction breakdown

We decided to have pairs of instructions for two reasons. The first was to avoid making it a semi-widening operation, which would have had odd definitions of either vl or SEW. The second was that the operations can be independently useful if only part of the output is desired, for example in extracting the real part of complex values.

Behaviour of vl

Vl was declared to perform the mask only on the output. This was done for both consistency with similar instructions like vrgather as well as to reduce datapath complexity: both implementations which work one element at a time and implementations which work at a larger granularity are able to avoid special cases for these instructions.

Use of VLMAX

Unlike other vector instructions, these instructions depend on VLMAX. This was done to make it easier to reuse the logic needed for segment loads and stores and to ensure that the select logic for the muxes needed didn't depend on vl, again to reduce datapath complexity.

Behaviour of LMUL

Higher LMULs were defined to act as larger registers for consistency with other vector instructions like vrgather.

Lack of support for LMUL<1

LMUL<1 was not supported since it would greatly increase the amount of possibilities required for the muxes and therefore increase datapath complexity. While LMUL>1 can re-use the wiring for each pair of registers, LMUL<1 could not.