Efficient Support of the Scan Vector Model for RISC-V Vector Extension

Hung-Ming Lai
Department of Computer Science,
National Tsing Hua University
Hsinchu, Taiwan
hmlai@pllab.cs.nthu.edu.tw

Jenq-Kuen Lee
Department of Computer Science,
National Tsing Hua University
Hsinchu, Taiwan
jklee@cs.nthu.edu.tw

ABSTRACT

RISC-V vector extension (RVV) provides wide vector registers, which is applicable for workloads with high data-level parallelism such as machine learning or cloud computing. However, it is not easy for developers to fully utilize the underlying performance of a new architecture. Hence, abstractions such as primitives or software frameworks could be employed to ease this burden. Scan, also known as all-prefix-sum, is a common building block for many parallel algorithms. Blelloch presented an algorithmic model called the scan vector model, which uses scan operations as primitives, and demonstrates that a broad range of applications and algorithms can be implemented by them. In our work, we present an efficient support of the scan vector model for RVV. With this support, parallel algorithms can be developed upon those primitives without knowing the details of RVV while gaining the performance that RVV provides. In addition, we provide an optimization scheme related to the length multiplier feature of RVV, which can further improve the utilization of the vector register files. The experiment shows that our support of scan and segmented scan for RVV can achieve 2.85x and 4.29x speedup, respectively, compared to the sequential implementation. With further optimization using the length multiplier of RVV, we can improve the previous result to 21.93x and 15.09x speedup.

CCS CONCEPTS

• **Computer systems organization → Single instruction, multiple data**.

KEYWORDS

RISC-V vector extension, SIMD, scan

ACM Reference Format:
Hung-Ming Lai and Jenq-Kuen Lee. 2022. Efficient Support of the Scan Vector Model for RISC-V Vector Extension. In *51th International Conference on Parallel Processing Workshop (ICPP Workshops '22), August 29-September 1, 2022, Bordeaux, France.* ACM, Bordeaux, France, 8 pages. https://doi.org/10.1145/3547276.3548518

1 INTRODUCTION

RISC-V vector extension (RVV)[5] is an extension of the open standard RISC-V ISA. It provides a wide vector register file that can process a large number of data at once. This results is not only with better performance but also lower energy consumption when processing workloads with high data-level parallelism for fields like scientific computing, multimedia, machine learning, cryptography[9], or cloud computing. As a relatively young SIMD ISA extension compared to its predecessors like the AVX to x86 or the Neon to the Cortex series processors, RVV incorporates many powerful features in the hope of striking the balance between high performance and scalability. However, it is not easy for high-level algorithms to fully utilize the underlying hardware resources. Hence, abstractions like primitive instructions or software frameworks can bridge the gap between them. They can provide higher-level interfaces that are easier to express the semantics while hiding the details of the underlying ISA. Scan, also called all-prefix-sum, is a common building block for many parallel algorithms [2]. It takes a binary operator $\oplus$ and an array $[a_0, a_1, ..., a_{n-1}]$ of n elements and returns an array of the sum of all preceding elements for each element $[a_0, (a_0 \oplus a_1), ..., (a_0 \oplus ... \oplus a_{n-1})]$. This is the definition of the inclusive scan operation. For exclusive scan operation, the output array starts with a left identity $I_\oplus$ of the binary operator $\oplus$ which is $[I_\oplus, a_0, (a_0 \oplus a_1), ..., (a_0 \oplus ... \oplus a_{n-2})]$. The scan operations have been supported on many different architectures including the Connection Machine[1], GPUs[10], and also on the AVX-512 extensions[15]. With efficient support of scan operations, parallel algorithms built upon them can execute efficiently. Otherwise, they would be the bottleneck of the overall performance. The scan operation is popularized by Blelloch[1], who presented an algorithmic model called *the scan vector model*. There are three classes of primitive vector instructions in this model, the elementwise instructions, the permutation instructions, and the scan instructions. Parallel algorithms can exploit these general-purpose primitives as a high-level interface to the low-level parallel computing resource that hardware provides.

In our work, we present an efficient support of the scan vector model for RVV. With this support, parallel algorithms can be developed upon those primitives without knowing the details of RVV while gaining the performance that RVV provides. We support both scan and segmented scan primitives for RVV. In addition, we provide an optimization scheme related to the length multiplier feature of RVV, which can further improve the utilization of the vector register files. The experiment shows that our support of scan and segmented scan for RVV can achieve 2.85x and 4.29x speedup, respectively, compared to the sequential implementation.

With further optimization using the length multiplier of RVV, we can improve the previous result to 21.93x and 15.09x speedup.

The remainder of the paper is organized as follows. We first introduce RVV and the scan vector model to illustrate why the architecture is a good fit for the model in Section 2. Then we introduce how to program with RVV intrinsic API in Section 3. Next, we present our approach to support the scan model with split radix sort as a running example in Section 4. In Section 5, we describe the support of a more general scan operation, segmented scan. Next, in Section 6, we present the experiment results and an optimization scheme based on the vector grouping feature of RVV. And finally, we conclude this paper in Section 7.

2 BACKGROUND

2.1 RISC-V Vector Extension

RISC-V Vector Extension (RVV) is an optional extension of the base RISC-V ISA. RISC-V is an open standard ISA with a modular design, which allows processor vendors to freely select a set of extensions to be implemented on their products that are tailored to their needs. RVV as an extension provides parallel computing capability to the base RISC-V ISA. Unlike the RISC-V P extension which uses the general-purpose register (GPR) to perform packed-SIMD execution that is limited to 32 or 64 bits with respect to whether the base ISA in use is RV32 or RV64, RVV adds an additional 32 vector register file to perform the SIMD operations. In addition, RVV does not set the vector length as an architecture constant. Instead, the vector length is an implementation-defined constant, allowing different vector lengths among different microarchitectures. This feature enables RVV programs to scale across different implementations of the architecture automatically without being compiled or rewritten again. Research communities have found the feature useful for environments such as TVM [3, 7]. Vector optimizations are also in an area of studies in the academic community [8, 13, 14].

2.2 The Scan Vector Model

Blelloch [1] presented an algorithmic model called the scan vector model and demonstrated that a broad range of parallel algorithms can be described by these primitives. The scan vector model is a parallel vector model with three classes of primitive vector instructions: elementwise instructions, scan instructions, and permutation instructions. The elementwise instructions possess high data-level parallelism since every element can be processed, independently. The permutation instructions also possess the same level of parallelism if there is additional storage for out-of-place replacements. The scan instructions, however, require the sum of all previous elements, which seems hard to be processed in parallel. In fact, the scan operations can be vectorized by many different algorithms. It can perform parallel prefix optimization assuming associativity with reduction operations.

3 PROGRAMMING WITH RISC-V VECTOR EXTENSION INTRINSIC

The RISC-V International vector working group creates the RVV intrinsic APIs[4] that have been supported in GCC and LLVM compilers. With the help of the intrinsic APIs, developers can program in C language while having more direct access to the RVV instructions. Different from programming with inline assembly, intrinsic is more expressive and readable and allows compilers to perform further optimization.

3.1 Vector Length Agnostic Programming Model

Because of the implementation-defined vector register length feature of RVV, developers need to develop their application in a vector length agnostic (VLA) fashion. In VLA programming model, the vector register length is treated as an unknown parameter that is determined at runtime. This provides direct support for a parallel computing technique called *strip-mining*. Strip-mining is a common approach for handling a large number of elements with a size that exceeds that of a vector register can be fit into. It works by handling a number of elements each iteration and iterates until all elements have been processed. For ISA with vector length specific design, using strip-mining needs an additional loop to handle the case that the vector length can not divide the number of elements without remainder. For example, if an ISA's SIMD instructions can process 4 elements. When it processes 13 elements, it needs to handle the remaining 1 element after 3 strip-mining iterations. This could result in a larger code size and worse performance. As for RVV, with the help of the configuration-setting instructions, there is no need to handle the remainder elements. We illustrate an example of strip-mining on RVV in C code using RVV intrinsic and its corresponding RVV assembly in Listing 1 and Listing 2.

```
1  // Perform element-wise addition between a and b.
2  // Store the result to a.
3  void vector_add(size_t n, int *a, int *b) {
4      size_t vl;
5      for (; n > 0; n -= vl) {
6          vl = vsetvl_e32m1(n);
7          vint32m1_t va = vle32_v_i32m1(a, vl);
8          vint32m1_t vb = vle32_v_i32m1(b, vl);
9          va = vadd(va, vb, vl);
10         vse32(a, va, vl);
11         a += vl;
12         b += vl;
13     }
14 }
```

Listing 1: Example of strip-mining for RVV in C

Listing 1 demonstrates how to vectorize a pairwise addition operation by using the strip-mining technique with RVV intrinsic. This `vector_add` function takes the number of elements `size_t n` and two integer array `a` and `b` as input. The result of the function is stored in `a`. In each iteration inside the for-loop, the application vector length (AVL), in this case is the `n` in line 6, will be passed as an argument to the vector configuration instruction - `vsetvl`. The `e32m1` suffix after it is the selected element width (SEW) and the length multiplier (LMUL). The SEW specifies the element width on which the vector operations will be operated. In this example, since the type `int` is 32-bit long, the SEW is set to 32 by the `e32` suffix.

```
# assume
# a0 stores n
# a1 stores address pointing to a[]
# a2 stores address pointing to b[]
vector_add:
    beqz a0, End
Loop:
    vsetvli a3, a0, e32, m1, ta, mu
    # load vl=a3 elements of data from a[] and b[]
    vle32.v v8, (a1)
    vle32.v v9, (a2)
    # add data from a[] and b[] to v8
    vadd.vv v8, v8, v9
    # store the result to a[]
    vse32.v v8, (a1)
    slli a4, a3, 2
    # a += vl
    add a1, a1, a4
    # n -= vl
    sub a0, a0, a3
    # b += vl
    add a2, a2, a4
    bnez a0, Loop
End:
    ret
```

Listing 2: Example of strip-mining for RVV in assembly

Listing 2 is the assembly of Listing 1. The `vsetvl_e32m1` intrinsic from the C code example in Listing 1 corresponds to the `vsetvli` RVV instruction in Listing 2.

3.2 Vector Masking

RVV also provides selected execution with vector masking. RVV always uses vector register v0 to store the mask value. This can save the encoding for specifying the mask register. The behavior of masked elements is specified by the *mask policy*. There are two types of mask policy that can be configured by the `vsetvl` instruction. They are the *mask agnostic policy* and the *mask undisturbed policy*. If using the mask agnostic policy, the masked value is undefined after the execution. Listing 3 is the function signature of a masked add instruction. The mask policy is encoded by the `maskedoff` argument. If it is passed with a `vundefined()`, which is the vector initialization function from RVV intrinsic API, then the agnostic policy is used. Otherwise, it uses the mask undisturbed policy and lets the masked elements have values from `maskedoff`.

```
vint32m1_t vadd_vv_i32m1_m (vbool32_t mask,
                            vint32m1_t maskedoff,
                            vint32m1_t op1,
                            vint32m1_t op2,
                            size_t vl);
```

Listing 3: Function signature of masked add instruction

3.3 Vector Length Multiplier

RVV allows grouping multiple vector registers together with the length multiplier (LMUL). The value of LMUL is specified by a 3-bit control state register `vlmul`. The LMUL values that every implementation must support are integers 1, 2, 4, and 8. For LMUL being larger than 1, multiple vector registers with consecutive numbers will form a vector register group. Instructions should reference the first vector register from the group. For instance, if the value of LMUL is 8, there are 4 groups v0-7, v8-15, v16-23, and v24-31. Then if a vector instruction uses v8 as an operand, it operates on the vector group v8-15. The type system in the RVV intrinsic API has explicit LMUL settings. One can not pass an incompatible type to functions.

4 SUPPORT OF THE SCAN VECTOR MODEL

The scan vector model consists of three classes of primitive instructions - elementwise instructions, scan instructions, and permutation instructions. In this section, we will present how we support these primitive instructions, and together we can develop parallel algorithms purely based on these primitive instructions without concerning how the primitives are mapped to RVV.

4.1 Elementwise Instructions

The elementwise instructions operate on two vectors with the same length. They perform arithmetic or logical primitive such as $+,-,*$, logical and, and logical or on each element and produce a vector with the same length as the input vector. To vectorize this class of instructions for RVV, we use strip-mining to iterate through all blocks of elements in the vector. Listing 4 is the example of how we vectorize the one of the elementwise instructions - `p-add` instruction. The p-add instruction adds two vectors together. In this example, we implement a variant that take an array of `unsigned int`, `a`, and a scalar `x` which will perform elementwise addition of `a` and a vector with broadcast value `x`. We first pass the AVL `n` to the `vsetvl` instruction to get the actual vector length `vl`. Then we add the vector variable `va` with the scalar `a` by the `vadd` intrinsic. The result is stored back to `a` by the `vse32` instruction. Finally, we let the pointer `a` point to the next element to be processed and subtract the AVL by `vl`.

```
void p_add(int n, unsigned int *a, unsigned int x) {
    size_t vl;
    for (; n > 0; n -= vl) {
        vl = vsetvl_e32m1(n);
        vuint32m1_t va = vle32_v_u32m1(a, vl);
        va = vadd(va, x, vl);
        vse32(a, va, vl);
        a += vl;
    }
}
```

Listing 4: Elementwise instruction - p-add

4.2 Permutation Instructions

The permutation instructions move elements based on an index array that specifies the new index of an element. We can achieve this by using the indexed store instructions - *VSUXEI* in RVV. Since the in-place permute operation will create data dependency, we support the out-of-place permutation. Listing 5 is the code of the permute instruction.

```
void permute(int n, unsigned int *src, unsigned int *dst,
     unsigned int *index) {
    size_t vl;
    for (; n > 0; n -= vl) {
        vl = vsetvl_e32m1(n);
        vuint32m1_t v = vle32_v_u32m1(src, vl);
        vuint32m1_t vindex = vle32_v_u32m1(index, vl);
        vindex = vsll(vindex, 2, vl);
        vsuxei32(dst, vindex, v, vl);
        src += vl;
        index += vl;
    }
}
```

Listing 5: Permutation instruction - permute

4.3 Unsegmented Plus-scan

Scan operations are the most important class of primitives for the scan vector model. Algorithms built upon the scan vector model assume that scan operations have low time complexity. As a result, the support for the scan operations on the actual target machine will directly affect the performance. In our support, we once again use the strip-mining technique to handle the array with a large number of elements. Listing 6 is the code of our support of the plus-scan instructions. The scan instruction is composed of the outer loop that strip-mining through every element and an inner loop that performs in-register scan. In each iteration, there are three steps. First, one is performing the in-register scan. Second, one is adding the carry number from previous iteration. The last is to update the carry number by retrieving the value from the last element in this iteration.

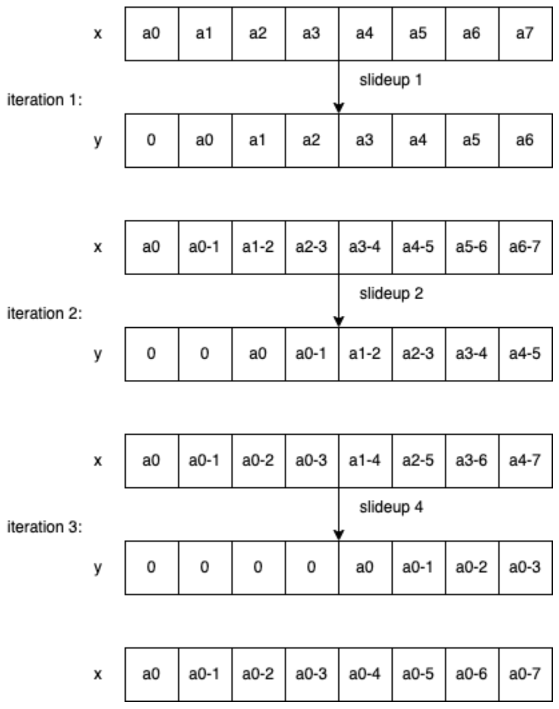


Figure 1: In-register scan steps

To support the in-register scan for VLA architecture like RVV, we need to treat the vector register length as a parameter. In Figure 1, we can observe that a vector containing 8 elements needs three iterations to complete. It turns out that the number of iteration required is $\lceil \lg N \rceil$ for $vl = N$.

```
void plus_scan_ui(int n, unsigned int *src) {
    size_t vl;
    size_t vlmax = vsetvlmax_e32m1();
    unsigned int carry = 0;
    vuint32m1_t x, y, vec_zero;
    vec_zero = vmv_v_x_u32m1(0, vlmax);
    for (; n > 0; n -= vl) {
        vl = vsetvl_e32m1(n);
        x = vle32_v_u32m1(src, vl);
        for (size_t offset = 1; offset < vl;
             offset = offset << 1) {
            y = vslideup_vx_u32m1(vec_zero, x,
                                  offset, vl);
            x = vadd_vv_u32m1(x, y, vl);
        }
        x = vadd_vx_u32m1(x, carry, vl);
        vse32(src, x, vl);
        carry = src[vl - 1];
        src += vl;
    }
    return;
}
```

Listing 6: Scan instruction - unsegmented plus-scan

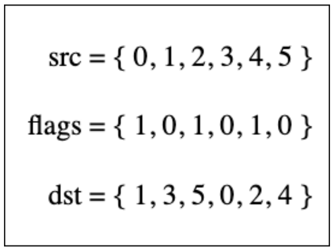
4.4 Split Radix Sort

Here we present how we can deploy useful applications to RVV with primitives mentioned in the previous sections by a running example of the radix sort algorithm. Aside from the primitive vector instructions, there are also several useful operations that can be built from primitive vector instructions. For instance, the enumerate and the split operations in the split radix sort algorithm. With the primitives and the operations, an algorithm can be built upon them efficiently and hide all the details related to RVV. For an array with elements that are 32-bit unsigned integers, we iterate from the less significant bit to the most significant bit and split the array in each iteration based on the value of the bit. In i^{th} iteration, elements with i^{th} bit value equal to 1 will be moved to the right, and elements with i^{th} bit value equal to 0 will be moved to the left while preserving the original order. Figure 2 shows the sorting process of the split radix sort algorithm.

$$\begin{aligned}
\text{src} &= \{\ \ 5,\ \ 2,\ \ 1,\ \ 3,\ \ 1\ \} \\
\text{src in binary} &= \{\ 101, 010, 001, 011, 001\ \} \\
\text{src 0th bit} &= \{\ \ 1,\ \ 0,\ \ 1,\ \ 1,\ \ 1\ \} \\
\\
\text{split src by 0th bit value} &= \{\ \ 2,\ \ 5,\ \ 1,\ \ 3,\ \ 1\ \} \\
\text{src in binary} &= \{\ 010, 101, 001, 011, 001\ \} \\
\text{src1st bit} &= \{\ \ 1,\ \ 0,\ \ 0,\ \ 1,\ \ 0\ \} \\
\\
\text{split src by 1st bit value} &= \{\ \ 5,\ \ 1,\ \ 1,\ \ 2,\ \ 3\ \} \\
\text{src in binary} &= \{\ 101, 001, 001, 010, 011\ \} \\
\text{2nd bit} &= \{\ \ 1,\ \ 0,\ \ 0,\ \ 0,\ \ 0\ \} \\
\\
\text{split src by 2nd bit value} &= \{\ \ 1,\ \ 1,\ \ 2,\ \ 3,\ \ 5\ \}
\end{aligned}$$

Figure 2: Sorting process of the split radix sort algorithm

We first examine the split operation that is used extensively in the scan vector model. The split operation is a form of permutation that uses a vector of flags to split the source vector into two halves. Figure 3 shows an example that the source vector `src` is split according to their flags value. Elements with flag value 0 (1, 3, 5) will be put at the bottom (starting at index 0). Elements with a flag value 1 will be packed to the destination vector after the last element with flag value 0.


src = { 0, 1, 2, 3, 4, 5 }

flags = { 1, 0, 1, 0, 1, 0 }

dst = { 1, 3, 5, 0, 2, 4 }

Figure 3: An example of split operation

Since the operation includes permute, we need an additional storage buffer to store the vector after permutation. In addition, there are also intermediate results.

```c
void split(int n, unsigned int *src, unsigned int *dst,
     unsigned int *flags) {
    unsigned int *i_up
        = malloc(n * sizeof(unsigned int));
    unsigned int *i_down
        = malloc(n * sizeof(unsigned int));
    unsigned int count
        = enumerate(n, flags, i_up, 0);
    enumerate(n, flags, i_down, 1);
    p_add(n, i_down, count);
    p_select(n, flags, i_down, i_up);
    permute(n, src, dst, i_up);
    free(i_up);
    free(i_down);
}
```

Listing 7: Split Operation

Enumerate is an operation that takes a vector of flags as input and outputs a vector of integer that the value of an element is set to i if its flag is the i^{th} true flag. The enumerate instructions are essentially an exclusive plus-scan on a flags vector that true flag is set to 1. Unlike the general plus-scan, the enumerate operation puts a restriction on the input vector. The input flags should contain only value 1 or 0. This restriction gives chances for optimization. In Listing 8, instead of using the unsegmented scan mentioned in the previous section, we exploit the viota instruction of RVV. The viota instruction is essentially an in-register enumerate operation, it allows only mask register as input, and outputs an exclusive scan vector with specified SEW and LMUL. To support the whole vector enumerate instruction, we need to propagate the in-register result to the next strip-mining iteration. This is where vcpop instruction comes in handy. In line 13, vcpop(mask, vl) returns the value to an unsigned integer variable count. Then, in every iteration, the result of the viota will add count to all the elements.

```c
1  unsigned int enumerate(int n, unsigned int *flags,
        unsigned int *dst, bool setBit) {
2      size_t vl;
3      unsigned int count = 0; // count number of bits set
4      unsigned int carry;
5      for (; n > 0; n -= vl) {
6          vl = vsetvl_e32m1(n);
7          vuint32m1_t v = vle32_v_u32m1(flags, vl);
8          vbool32_t mask = vmseq(v, setBit, vl);
9          v = viota_m_u32m1(mask, vl);
10         v = vadd(v, count, vl);
11         vse32(dst, v, vl);
12         count += vcpop(mask, vl);
13         flags += vl;
14         dst += vl;
15     }
16     return count;
17 }
```

Listing 8: Enumerate operation

Listing 9 is the code for the complete split radix sort algorithm. On lines 2 and 4, we allocate new storage space for split operation and the bit flags. Since we are sorting the type unsigned int, we need to iterate over 32 bits from lines 6 to 13. In every iteration, we first get the i^{th} bit flags, then we split the vector src by these flags. After the split, buffer contains the new vector. Thus we swap the memory spaces buffer and src pointing to, making sure src is always pointing to the new vector. Since there are 32 bits for type unsigned int, which is an even number, the original memory space pointed by src will contain the sorted vector after all iterations have ended.

```c
1  void split_radix_sort(int n, unsigned int *src) {
2      unsigned int *buffer
3          = malloc(n * sizeof(unsigned int));
4      unsigned int *flags
5          = malloc(n * sizeof(unsigned int));
6      for (int i = 0; i < 32; i++) {
7          get_flags(n, src, flags, i);
8          split(n, src, buffer, flags);
9          // swap src and buffer
10         unsigned int* tmp = src;
11         src = buffer;
12         buffer = tmp;
13     }
14     free(buffer);
15     free(flags);
16     return;
17 }
```

Listing 9: Split radix sort

5 SUPPORT OF THE SEGMENTED SCAN FOR RISC-V VECTOR EXTENSION

It is useful to split one array into several segments that can execute scan operations independently. For instance, an algorithm like quick sort needs to split the whole array into different segments and then sort each segment recursively. To represent different segments, Blelloch [2] suggested using structures called the *segment-descriptor*. There are three possible segment-descriptor: *head-flags*, *lengths*, and *head-pointers*. The *head-flags* uses an array of flags to indicate the starting index of each segment. The *lengths* specifies the length of each segments. The *head-pointers* uses an array of pointers pointing to the starting element of each segment.

In our work, we choose the *head-flags* as our segment-descriptor, since it can be mapped to RVV instructions more directly without additional interpretation.

5.1 Segmented Plus-scan

We now use the segmented version of the plus-scan operation to demonstrate how we exploit RVV to support this operation. Listing 10 is the code of our segmented scan. The strip-mining part is similar to the unsegmented scan, with one difference in handling the carry propagation. Unlike unsegmented plus-scan that all the elements can add the carry value after the in-register scan. Segmented scan can only add carry values to the elements before the start of the next new segment. So we need to find the position of the next new head-flag and create a mask that can add carry values only to elements before this head-flag. With the help of the `vmsbf` instruction in line 15, we can get a mask exactly as we want by providing a mask value created by the head-flags to this instruction. In the next section, we will discuss how we implement the in-register segmented scan in detail.

```
1  void seg_plus_scan_ui(int n, unsigned int *src, unsigned
        int *head_flags) {
2      size_t vl;
3      size_t vlmax = vsetvlmax_e32m1();
4      unsigned int carry = 0;
5      vuint32m1_t x, y, vec_zero, vec_one;
6      vuint32m1_t flags, flags_slideup;
7      vbool32_t mask, carry_mask;
8      vec_zero = vmv_v_x_u32m1(0, vlmax);
9      vec_one = vmv_v_x_u32m1(1, vlmax);
10     for (; n > 0; n -= vl) {
11         vl = vsetvl_e32m1(n);
12         x = vle32_v_u32m1(src, vl);
13         flags = vle32_v_u32m1(head_flags, vl);
14         mask = vmsne_vx_u32m1_b32(flags, 0, vl);
15         carry_mask = vmsbf(mask, vl);
16         flags = vmv_s_x_u32m1(flags, 1, vl);
17         for (size_t offset = 1; offset < vl;
18              offset = offset << 1) {
19             mask = vmsne_vx_u32m1_b32(flags, 1, vl);
20             y = vslideup_vx_u32m1(vec_zero,
21                                   x, offset, vl);
22             x = vadd_vv_u32m1_m(mask, x, x, y, vl);
23             flags_slideup = vslideup_vx_u32m1(vec_one,
24                                               flags,
25                                               offset,
26                                               vl);
27             flags = vor_vv_u32m1(flags,
28                                  flags_slideup,
29                                  vl);
30         }
31         x = vadd_vx_u32m1_m(carry_mask, x, x, carry, vl);
32         vse32(src, x, vl);
33         carry = src[vl - 1];
34         src += vl;
35         head_flags += vl;
36     }
37     return;
38 }
```

Listing 10: Segmented plus-scan

5.2 In-register Segmented Plus-scan

Just like the unsegmented scan, the in-register segmented scan requires lg N slideup instructions for vector register with N elements. Since if there are no head-flags appearing in this strip-mining iteration, it needs to produce the same result as an unsegmented scan does. To support the segmented scan, we want to disable certain elements to add their prefix result within in-register scan operation. Figure 4 illustrates this idea. In Figure 4, we can see that if we derive the correct mask, we can use the same slideup-and-add instructions sequence as the one used in the unsegmented scan. It turns out that the mask itself can be derived by sliding up the previous mask and performing a pairwise logical and. However, the mask register

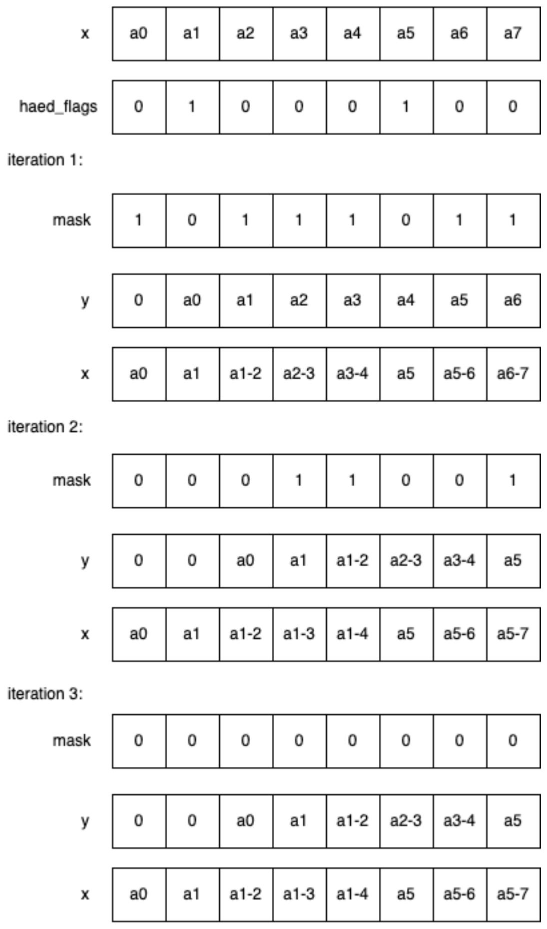

x a0 a1 a2 a3 a4 a5 a6 a7
haed_flags 0 1 0 0 0 1 0 0
iteration 1:
mask 1 0 1 1 1 0 1 1
y 0 a0 a1 a2 a3 a4 a5 a6
x a0 a1 a1-2 a2-3 a3-4 a5 a5-6 a6-7
iteration 2:
mask 0 0 0 1 1 0 0 1
y 0 0 a0 a1 a1-2 a2-3 a3-4 a5
x a0 a1 a1-2 a1-3 a1-4 a5 a5-6 a5-7
iteration 3:
mask 0 0 0 0 0 0 0 0
y 0 0 a0 a1 a1-2 a2-3 a3-4 a5
x a0 a1 a1-2 a1-3 a1-4 a5 a5-6 a5-7

Figure 4: In-register segmented scan steps

does not provide slideup instruction. Thus we need to use a whole vector register to perform these operations. On lines 23 and 27 of Listing 10, we update the flags (which is the inverse of the mask) for the next iteration.

6 EXPERIMENTS

6.1 Experiment Environment

Our experiment result is evaluated on an open-source RISC-V ISA simulator called Spike[6], with RVV configured to 256-bit, 512-bit, and 1024-bit vector register length to test the performance scalability. Since Spike is only a functional model of RISC-V, which means that it is not cycle-accurate, we use dynamic instruction count instead as the performance metric.

6.2 Evaluation of the Support of the Scan Vector Model for RVV

In this study, we implement the segmented and unsegmented scan. We also implement the primitive vector instructions and operations required by the split radix sort algorithm mentioned in Section 4. This includes elementwise instructions *p-add* and *p-select*, the permutation instruction *permute*, the operation *enumerate*, and *split*.

N	split_radix_sort()	qsort()	speedup
10^2	23988	17158	0.7152743038
10^3	94842	277480	2.92570802
10^4	803690	3470344	4.318013164
10^5	19603490	43004753	2.193729433
10^6	195102988	511107188	2.619678936

Table 1: Spike simulation result comparing dynamic instruction counts of split-radix-sort for RVV and a baseline qsort from stdlib

N	p_add()	p_add_baseline()	speedup
10^2	66	632	9.575757576
10^3	297	6002	20.20875421
10^4	2826	60001	21.23177636
10^5	28134	600001	21.32654439
10^6	281259	6000001	21.33265424

Table 2: Spike simulation result comparing dynamic instruction counts of p-add for RVV and a baseline sequential p-add implementation

N	plus_scan()	plus_scan_baseline()	speedup
10^2	311	626	2.012861736
10^3	2670	6026	2.256928839
10^4	26281	60026	2.284007458
10^5	262531	600026	2.285543422
10^6	2625031	6000026	2.285697197

Table 3: Spike simulation result comparing dynamic instruction counts of plus-scan for RVV and a baseline sequential plus-scan implementation

In this section, we show the performance of comparing our RVV support version to the sequential version baseline. The baseline is also RVV directly from LLVM to RVV without our scan support. The baseline is a pure C code without the use of RVV intrinsics. The following experiment is conducted on the Spike simulator with 1024-bit vector length configuration, which can have 32 32-bit elements, using LMUL value 1.

Table 1 is the experiment result of comparing the split radix sort algorithm implemented by our support of the scan vector model to the `qsort()` from the C standard library `stdlib.o`. Table 2 is the experiment result of comparing the elementwise instruction - *p-add* and a baseline sequential implementation. Table 3 is the experiment result of comparing the unsegmented plus-scan instruction and a baseline sequential implementation. Table 4 is the experiment result of comparing the segmented plus-scan instruction and a baseline sequential implementation.

6.3 Evaluation of Length Multiplier Optimization

As discussed in Section 4, RVV intrinsics provide explicit LMUL setting. We can specify the LMUL by changing the suffix - m<lmul> in the intrinsic function and also for the data types. Intuitively, a larger LMUL means a larger number of elements a vector instruction can operate on, which should result in less dynamic instruction count. However, that is not always the case. The reason is that the size of the whole vector register files is fixed for a given RVV microarchitecture. Grouping multiple vector registers as one will result in less number of vector registers for register allocation. This causes more register pressure and hence more register spilling would occur. As a result, we can observe performance anomaly when we set too large LMUL.

N	seg_plus_scan()	seg_plus_scan_baseline()	speedup
10^2	331	1124	3.395770393
10^3	2639	11024	4.177339901
10^4	25693	110024	4.282255867
10^5	256289	1100024	4.292123345
10^6	2562539	11000024	4.292626961

Table 4: Spike simulation result comparing dynamic instruction counts of segmented plus-scan for RVV and a baseline sequential segmented plus-scan implementation

	Instruction count of seg_plus_scan()			
N/LMUL	1	2	4	8
10^2	331	1124	145	2090
10^3	2639	11024	887	2668
10^4	25693	110024	8377	9284
10^5	256289	1100024	82907	74650
10^6	2562539	11000024	828205	728586

Table 5: Dynamic instruction count of the segmented plus scan instruction with different LMUL setting

In Table 5, we can see that if we set the LMUL to 8, `seg_plus_scan` experiences a slowing down when the number of elements is 10^2 and 10^3 compared to the case that LMUL is 1. This is because the compiled result contains many instructions for register spilling and loading the data back, which causes great overhead. This is not completely the start-up overhead since we are comparing two vectorized code instead of comparing sequential implementation with vectorized implementation. When the number of elements to be processed is small, the overhead of the register spilling can not be covered by the advantage of a smaller iteration. Still, when the number of elements of the vector is greater than 10^4, the advantage of smaller strip-mining iterations starts to appear. And another observation is that, in Table 6, the ratio of LMUL to speedup over LMUL=1 is decreasing when using larger LMUL. This is because when using larger LMUL, the number of instructions that are for register spilling will increase. Here we provide a conclusion and a suggestion for choosing LMUL. In conclusion, for workloads with small vector size, the overhead of register spilling can be significant. For workloads with very large vector size, the dynamic instruction count can be covered. However, the proportion of memory copy instructions will increase, which may lead to lower performance even if the instruction count is smaller.

	Ratio of (speedup to LMUL=1)/(current LMUL)		
N/LMUL	2	4	8
10^2	0.7290748899	0.5706896552	0.01979665072
10^3	0.8551523007	0.7437993236	0.1236413043
10^4	0.8695931767	0.7667721141	0.3459311719
10^5	0.8720338349	0.772820751	0.4291510382
10^6	0.872330539	0.7735219541	0.4396425062

Table 6: Ratio of different LMUL setting's speedup comparing to LMUL set to 1 divided by current LMUL setting

	segmented plus-scan	p-add
vlen	instruction count	instruction count
128	115039	22534
256	72539	11284
512	43789	5659
1024	25693	2851

Table 7: Instruction count over different *vlen* for segmented plus-scan and p-add

6.4 Evaluation of Performance Scalability over Different Microarchitectures

The vector length agnostic design of RVV allows processor vendors to choose their own implementation that meets their product requirements. Commercial RVV IPs like the Andes's NX27V[12] and Sifive's Intelligence X280 [11] provide a configurable vector register length up to 512-bit. Compared to other VLS SIMD architecture, it is easier for RVV code to scale over different microarchitectures. However, not every algorithm can have the same scalability. Table 7 shows the instruction count of segmented plus-scan and p-add over different `vlen` setting. In Figure 5, we can see that the speedup compared to `vlen`=128 has nearly the same value as vlen/128 (the ideal speedup). However, operations that are not elementwise parallel such as scan, can not scale as much as elementwise instructions. In Figure 5, we can see that even vlen=1024 has 8 times the length of the vector register compared to vlen=128, the speedup is only 4.65.

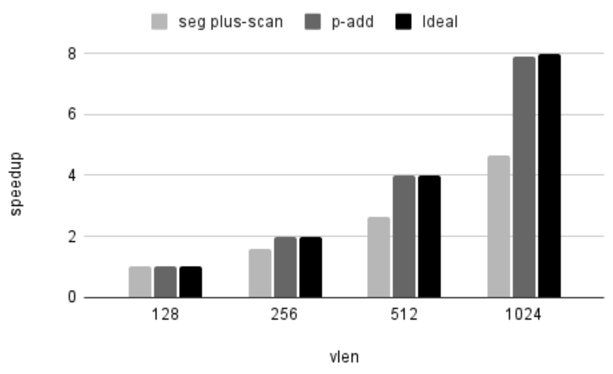


Figure 5: Speedup compared to `vlen` = 128 over different `vlen` for segmented plus-scan and p-add

7 CONCLUSION

In this paper, we support the scan vector model for RISC-V vector extension (RVV). We provide design details for our support with the three classes of primitive instructions in the model. We support both scan and segmented scan primitives for RVV. Then we demonstrate how to use these primitive instructions as an interface to deploy parallel computing workloads to RVV. Finally, in the experiment, we show that with our support of the scan vector model, primitive instructions and algorithms can be executed on RVV target efficiently. For scan and segmented scan instructions, our support can achieve 2.85x and 4.29x speedup, respectively. With further optimization using the length multiplier of RVV, we can improve results to 21.93x and 15.09x speedup.

ACKNOWLEDGMENTS

This work is supported in part by Taiwan MOST and Realtek.

REFERENCES

[1] Guy E Blelloch. 1990. *Vector models for data-parallel computing*. Vol. 2. MIT press Cambridge.
[2] Guy E. Blelloch. 2004. Prefix sums and their applications. (5 2004). https://doi.org/10.1184/R1/6608579.v1
[3] Yi-Ru Chen, Chia-Hsuan Chang, Chao-Lin Lee Hui-Hsin Liao, CC Lin, Chun-Chieh Yang Yuan-Ming Chang, Heng-Kuan Lee I-Wei Wu, and Jenq-Kuen Lee. 2019. Support TVM QNN Flow on RISC-V with SIMD Computation. TVM and Deep Learning Compiler Conference, Seattle, Dec 2019 (lightning talk and slides).
[4] RISC-V International. 2022. *RISC-V Vector Extension Intrinsic Document*. Retrieved June 10, 2022 from https://github.com/riscv-non-isa/rvv-intrinsic-doc
[5] RISC-V International. 2022. *riscv-v-spec*. Retrieved June 10, 2022 from https://github.com/riscv/riscv-v-spec
[6] RISC-V International. 2022. *Spike RISC-V ISA Simulator*. Retrieved June 10, 2022 from https://github.com/riscv-software-src/riscv-isa-sim
[7] Jenq-Kuen Lee, Chun-Chieh Yang, Piyo Chen Allen Lu, CH Chang Yuan-Ming Chang, HH Liao Yi-Ru Chen, Ssu-Hsuan Lu Chao-Lin Lee, and Shao-Chung Wang. 2019. Supporting TVM on RISC-V Architectures with SIMD Computations. TVM and Deep Learning Compiler Conference, Seattle, Dec 2019 (lightning talk and slides).
[8] Yu-Te Lin and Jenq-Kuen Lee. 2016. Vector Data Flow Analysis for SIMD Optimizations on OpenCL Programs. *Concurr. Comput.: Pract. Exper.* 28, 5 (apr 2016), 1629–1654. https://doi.org/10.1002/cpe.3714
[9] Sabine Pircher, J Geier, Alexander Zeh, and Daniel Mueller-Gritschneder. 2021. Exploring the RISC-V Vector Extension for the Classic McEliece Post-Quantum Cryptosystem. In *2021 22nd International Symposium on Quality Electronic Design (ISQED)*. IEEE, 401–407.
[10] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. 2007. Scan Primitives for GPU Computing. In *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, Mark Segal and Timo Aila (Eds.). The Eurographics Association. https://doi.org/10.2312/EGGH/EGGH07/097-106
[11] SiFive. 2022. SiFive Intelligence X280. https://www.sifive.com/cores/intelligence-x280
[12] Andes Technology. 2022. AndesCore™ NX27V Processor. http://www.andestech.com/en/products-solutions/andescore-processors/riscv-nx27v/
[13] Shao-Chung Wang, Li-Chen Kan, Chao-Lin Lee, Yuan-Shin Hwang, and Jenq-Kuen Lee. 2017. Architecture and Compiler Support for GPUs Using Energy-Efficient Affine Register Files. *ACM Trans. Des. Autom. Electron. Syst.* 23, 2, Article 18 (nov 2017), 25 pages. https://doi.org/10.1145/3133218
[14] Shao-Chung Wang, Li-Chen Kan, Chao-Lin Lee, Yuan-Shin Hwang, and Jenq-Kuen Lee. 2017. Architecture and Compiler Support for GPUs Using Energy-Efficient Affine Register Files. *ACM Trans. Des. Autom. Electron. Syst.* 23, 2, Article 18 (nov 2017), 25 pages. https://doi.org/10.1145/3133218
[15] Wangda Zhang, Yanbin Wang, and Kenneth A. Ross. 2020. Parallel Prefix Sum with SIMD. In *International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures, ADMS@VLDB 2020, Tokyo, Japan, August 31, 2020*, Rajesh Bordawekar and Tirthankar Lahiri (Eds.). 1–11. http://www.adms-conf.org/2020-camera-ready/ADMS20_05.pdf