RISC-V Vector Extension Support in MLIR: Motivation, Abstraction, and Application

Speaker: Hongbin Zhang | hongbin2019@iscas.ac.cn

Authors: Hongbin Zhang (ISCAS), Diego Caballero (Google), Xulin Zhou (JNU), Tao Liang (HIT), Yingchi Long (HIT), Haolin Pan (ISCAS)
RISC-V Vector (RVV) Extension Introduction

RVV Overview

The RISC-V Vector extension adds support for high-performance vector operations that allow for the efficient processing of large amounts of data.

RVV Features

- Dynamic vector length at runtime, smaller code size.
- Vector length agnostic (VLA), better code portability.
- Functional unit pipelining, larger data-level parallelism.

RVV Ecosystem

- Library: OpenCV, OpenBLAS, etc.
- Compiler: GCC, LLVM
- Emulator: QEMU
- Hardware: Intelligence X280, XuanTie C906, etc.
The functional unit is fully pipelined, and it can start a new operation on every clock cycle. [1]

Motivation: RVV Special Features

**RVV Vector Processor**

Functional Unit Pipelining: Different processes work simultaneously.

The functional unit is fully pipelined, and it can start a new operation on every clock cycle. [1]


**SIMD Array Processor**

Functional Unit Array: Homogeneous unit work simultaneously.

VLEN* – Processor Design

Vector Register Group

Dynamic VL* at Runtime

VLEN* – Instruction Set

Fixed VL* at Runtime

---

* VLEN = Hardware Vector Register Length  * VL = Processing Vector Length
Motivation: RVV Special Features

RVV Vector Register Configuration

V31

... ...

V0

VLEN (Hardware Vector Register)

Vector Group

LMUL = 2
(Vector Register Group Multiplier)

VLEN x LMUL

VLMAX = VLEN x LMUL / SEW
(The maximum number of elements that a vector instruction can operate on)

VLMAX assists in calculating the VL returned by the vsetvl command
(VL: Processing Vector Length)

VLA (Vector Length Agnostic):
RVV code adapts to the machine’s vector register length at runtime.

RVV Tail Processing

Get the application vector length (d) at runtime

Mask-Based Approach

AVL = d
While(AVL > 0):
  do:
    if (not Tail)
      vector load
      vector add
      vector store
    else
      calculate mask
      masked load
      masked add
      masked store
  end if
End loop

Strip-Mining Approach

AVL = d
While(AVL > 0):
  do:
    vl = setvl AVL, LMUL, SEW
    vector load vl
    vector add vl
    vector store vl
  AVL = AVL - vl
End

VLMAX assists in calculating the VL returned by the vsetvl command
(VL: Processing Vector Length)
Motivation: RVV Special Features

Iterations for fixed vector length

```
affine for %idx = %c0 to %iter {
    %cur_idx = arith.muli %idx, %c4 : index
    %vec_input1 = affine.vector_load %input1[idx * 4]
        : memref<%x132>, vector<4x132>
    %vec_input2 = affine.vector_load %input2[idx * 4]
        : memref<%x132>, vector<4x132>
    %vec_output = arith.addi %vec_input1, %vec_input2 : vector<4x132>
    affine.vector_store %vec_output, %output[idx * 4]
        : memref<%x132>, vector<4x132>
}
```

Tail processing with mask operations

```
// Tail processing
scf.if %tail {
    %cur_idx = arith.muli %iter, %c4 : index
    %mask = vector.create_mask %rem : vector<4x11>
    %vec_input1 = vector.maskedload %input1[%cur_idx], %mask, %pass_thr
        : memref<%x132>, vector<4x11>, vector<4x132> into vector<4x132>
    %vec_input2 = vector.maskedload %input2[%cur_idx], %mask, %pass_thr
        : memref<%x132>, vector<4x11>, vector<4x132> into vector<4x132>
    %vec_output = arith.addi %vec_input1, %vec_input2 : vector<4x132>
    vector.maskedstore %output[%cur_idx], %mask, %vec_output
        : memref<%x132>, vector<4x11>, vector<4x132>
}
```

RVV Tail Processing

```
C[0] C[1] ... C[??] C[??]

Get the application vector length (d) at runtime

Mask-Based Approach

AVL = d
While(AVL > 0):
    do:
        vl = setvl AVL, LMUL, SEW
        vector load vl
        vector add vl
        vector store vl
        AVL = AVL - vl
End
```

Strip-Mining Approach

```
Tail = getTail(d)
Loop:
    if (not Tail)
        vector load
        vector add
        vector store
    else
        calculate mask
        masked load
        masked add
        masked store
    End if
End loop
```
Information Required at Compile Time:
- Dynamic VL Configuration
- AVL Configuration
- LMUL Configuration
- SEW Configuration
- Operations Dynamic VL Operand

Vector operations do not accept dynamic VL parameters.

%0 = arith.addf %v, %v : vector<8xf32>

MLIR Limitation

RVV Tail Processing

Mask-Based Approach

Strip-Mining Approach

Set Dynamic VL

Ops Accept Dynamic VL

Get the application vector length (d) at runtime

Tail = getTail(d)
Loop:
if (not Tail)
   vector load
   vector add
   vector store
else
   calculate mask
   masked load
   masked add
   masked store
end if
End loop

MLIR Limitation

RVV Tail Processing

+ B[0] B[1] ... B[?] B[?]

C[0] C[1] ... C[?] C[?]

No SETVL Operation Cannot Set Dynamic VL
MLIR Abstraction Support for RVV Backend

MLIR RISC-V Vector Dialect
- Operation
  - RVV Operation
  - RVV Intrinsic Operation
- Type
  - Scalable Vector Type
- Conversion/Translation
  - RVV Dialect
  - LLVM Dialect
  - LLVM IR
- Integration Test
MLIR Abstraction Support for RVV Backend

MLIR RISC-V Vector Dialect
- Operation
  - RVV Operation
- RVV Intrinsic Operation
- Type
  - Scalable Vector Type
- Conversion/Translation
  - RVV Dialect
  - LLVM Dialect
  - LLVM IR
- Integration Test

Fragmentation

Upstream

Vector Dialect
- AMX Dialect
- X86 Vector Dialect
- Arm Neon Dialect
- Arm SVE Dialect
- RVV Dialect

MLIR Lowering Paths - https://mlir.llvm.org/docs/Dialects/Vector/
MLIR Abstraction Support for RVV Backend

**Proposed Approach**
- Add abstraction support for dynamic VL in vector dialect.
- Add abstraction support for RVV-specific VL in RVV dialect.
- Improve generality with VP Intrinsic.
- Implement vectorization pass using a combination of Vector and RVV dialects.

**Vector Dialect** → Add abstraction support for dynamic VL.
- AMX Dialect
- X86 Vector Dialect
- Arm Neon Dialect
- Arm SVE Dialect

**RVV Dialect** → Add abstraction support for RVV-specific ops.
MLIR Abstraction Support for RVV Backend

### Dynamic Vector Length

```python
// While loop for strip-mining.
%tmpAVL, %tmpIdx = scf.while (%avl = %dim, %idx = %c0)
  | : (index, index) -> (index, index) {
// Ifavlgreaterthanzero.
  %cond = arith.cmpi sgt, %avl, %c0 : i64
// Passavl, idx to the other region.
  scf.condition(%cond) %avl, %idx : index, index
}
%bb0(%avl : index, %idx : index): Set Dynamic Vector Length
  // Perform the calculation according to the vl.
%vl = rvv.setvl %avl, %sew, %lmul : index
%vl_i32 = arith.index_cast %vl : index to i32
%vec_input1 = vector.exp.predication %mask, %vl_i32 : [vector<4x1i32>, i32 {
  %ele = vector.load %input1[idx] : memref<7x32>, vector<4x1i32>
  vector.yield %ele : vector<4x1i32>
} : vector<4x1i32>
%vec_input2 = vector.exp.predication %mask, %vl_i32 : [vector<4x1i32>, i32 {
  %ele = vector.load %input2[idx] : memref<7x32>, vector<4x1i32>
  vector.yield %ele : vector<4x1i32>
}: vector<4x1i32>
%result_vec = rvv.add %vec_input1, %vec_input2, %vl
  | : vector<4x1i32>, vector<4x1i32>, index
vector.exp.predication %mask, %vl_i32 : vector<4x1i32>, i32 {
  vector.store %result_vec, %output[idx] : memref<7x32>, vector<4x1i32>
  vector.yield
} : () -> ()
// Update idx andavl.
%new_idx = arith.addi %idx, %vl : index
%new_avl = arith.subi %avl, %vl : index
scf.yield %new_avl, %new_idx : index, index
```

### Scalable Vector Type

```python
[...]
```

### Predication Region

```python
Get the application vector length (d) at runtime

**Mask-Based Approach**

AVL = d
Loop:
if (not Tail)
  vector load
  vector add
  vector store
else
  calculate mask
  masked load
  masked add
  masked store
endif
End loop
```

**Strip-Mining Approach**

AVL = d
While(AVL > 0):
do:
  vl = setvl AVL, LMUL, SEW
  vector load vl
  vector add vl
  vector store vl
  AVL = AVL - vl
End

```python
RVV Tail Processing

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>B[0]</td>
<td>B[1]</td>
<td>...</td>
<td>B[?]</td>
<td>B[?]</td>
</tr>
<tr>
<td>C[0]</td>
<td>C[1]</td>
<td>...</td>
<td>C[?]</td>
<td>C[?]</td>
</tr>
</tbody>
</table>
```

## RVV Tail Processing

### Dynamic Vector Length

```python
// While loop for strip-mining.
%tmpAVL, %tmpIdx = scf.while (%avl = %dim, %idx = %c0)
  | : (index, index) -> (index, index) {
// Ifavl greater than zero.
  %cond = arith.cmpi sgt, %avl, %c0 : index
// Passavl, idx to the other region.
  scf.condition(%cond) %avl, %idx : index, index
}
%bb0(%avl : index, %idx : index): Set Dynamic Vector Length
  // Perform the calculation according to the vl.
%vl = rvv.setvl %avl, %sew, %lmul : index
%vl_i32 = arith.index_cast %vl : index to i32
%vec_input1 = vector.exp.predication %mask, %vl_i32 : [vector<4x1i32>, i32 {
  %ele = vector.load %input1[idx] : memref<7x32>, vector<4x1i32>
  vector.yield %ele : vector<4x1i32>
} : vector<4x1i32>
%vec_input2 = vector.exp.predication %mask, %vl_i32 : [vector<4x1i32>, i32 {
  %ele = vector.load %input2[idx] : memref<7x32>, vector<4x1i32>
  vector.yield %ele : vector<4x1i32>
}: vector<4x1i32>
%result_vec = rvv.add %vec_input1, %vec_input2, %vl
  | : vector<4x1i32>, vector<4x1i32>, index
vector.exp.predication %mask, %vl_i32 : vector<4x1i32>, i32 {
  vector.store %result_vec, %output[idx] : memref<7x32>, vector<4x1i32>
  vector.yield
} : () -> ()
// Update idx andavl.
%new_idx = arith.addi %idx, %vl : index
%new_avl = arith.subi %avl, %vl : index
scf.yield %new_avl, %new_idx : index, index
```
Application: MatMul Optimization

Step 5

Get VL for this iteration with rvv.setvl operation.

Step 6~9

Use the vector.exp.predication operation to wrap the arithmetic and vector operations.
Buddy Compiler As A Service: RVV Integration

VP Intrinsic Example Cases

VP Intrinsic Example Code

MLIR Lowering Translate to LLVM IR

Execute with QEMU

Error Report

https://buddy.isrc.ac.cn/
Summary

RVV Features
- Dynamic vector length at runtime, smaller code size.
- Vector length agnostic (VLA), better code portability.
- Functional unit pipelining, larger data-level parallelism.

MLIR Limitations for RVV Backend
MLIR cannot exploit the VLA features of RVV
- No vector operation can set dynamic VL.
- Vector operations do not accept dynamic VL parameters.

Proposed Approach and Application
- Add SetVL operation in RVV-specific dialect.
- Add vector predication operation in Vector dialect.
- Implement MatMul optimization with mixed Vector and RVV dialects.

[WIP] Upstream Proposal (New Page)
- Integrate vector length configuration with the current mask operation.
- Create a standalone vector length operation.
- Integrate dynamic vector representation into ODS.
Thanks

hongbin2019@iscas.ac.cn