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)

Buddy Compiler

RISC-V Vector（RVV）Extension Introduction

Buddy Compiler

RVV Overview

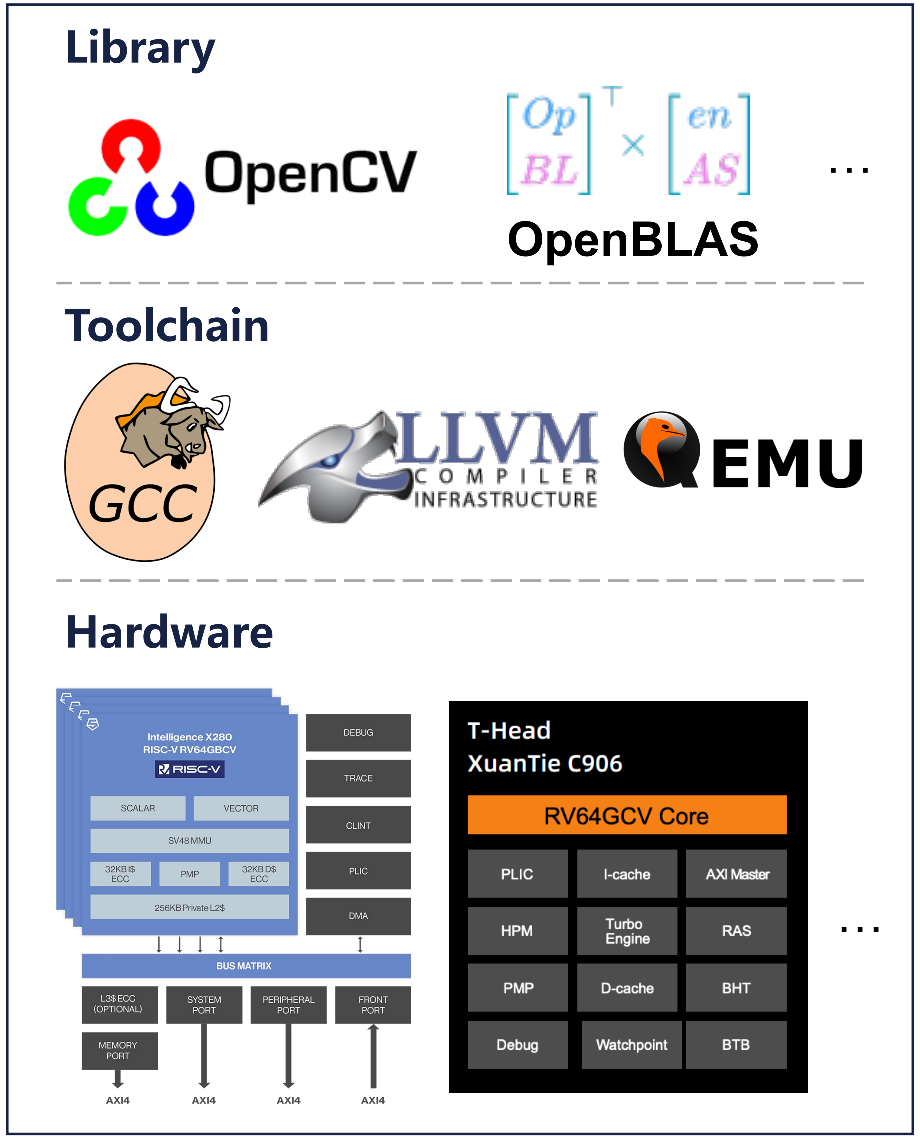
The RISC-V Vector extension adds support for high-performance vector operations that allow for the efficient processing of large amounts of data.

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.



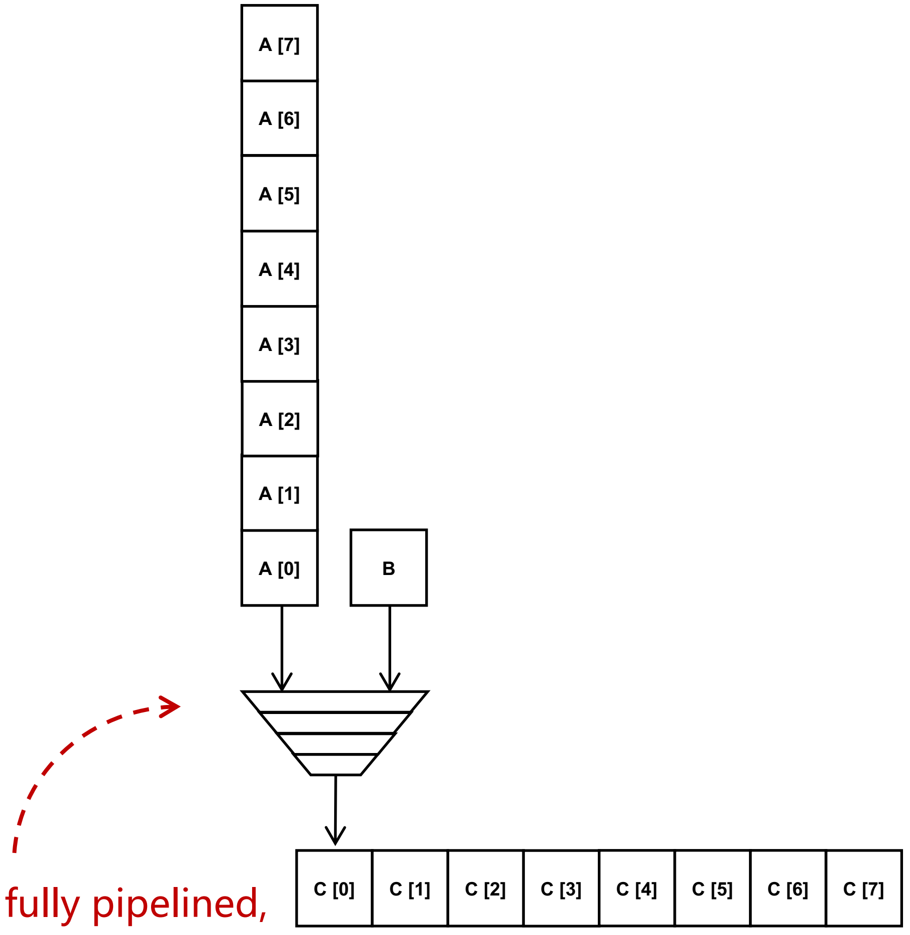
RVV Ecosystem

Motivation: RVV Special Features

Buddy Compiler

RVV Vector Processor

Functional Unit Pipelining: Different processes work simultaneously.

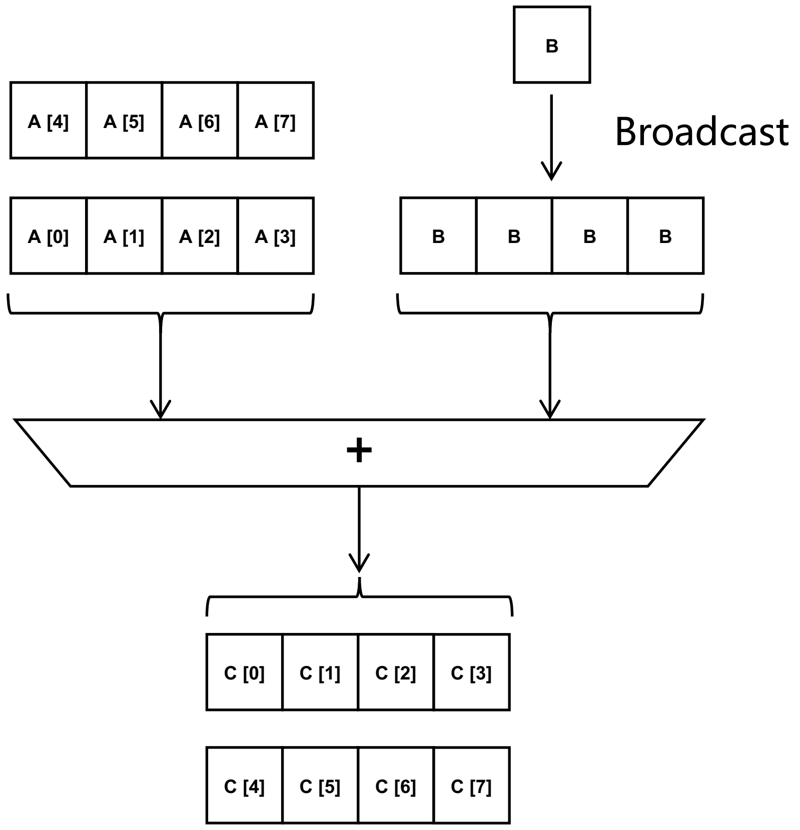


The functional unit is fully pipelined,

and it can start a new operation on every clock cycle. [1]

SIMD Array Processor

Functional Unit Array: Homogeneous unit work simultaneously.



[1] Hennessy J L, Patterson D A. Computer architecture: a quantitative approach. Sixth Edition.

Motivation: RVV Special Features

Buddy Compiler

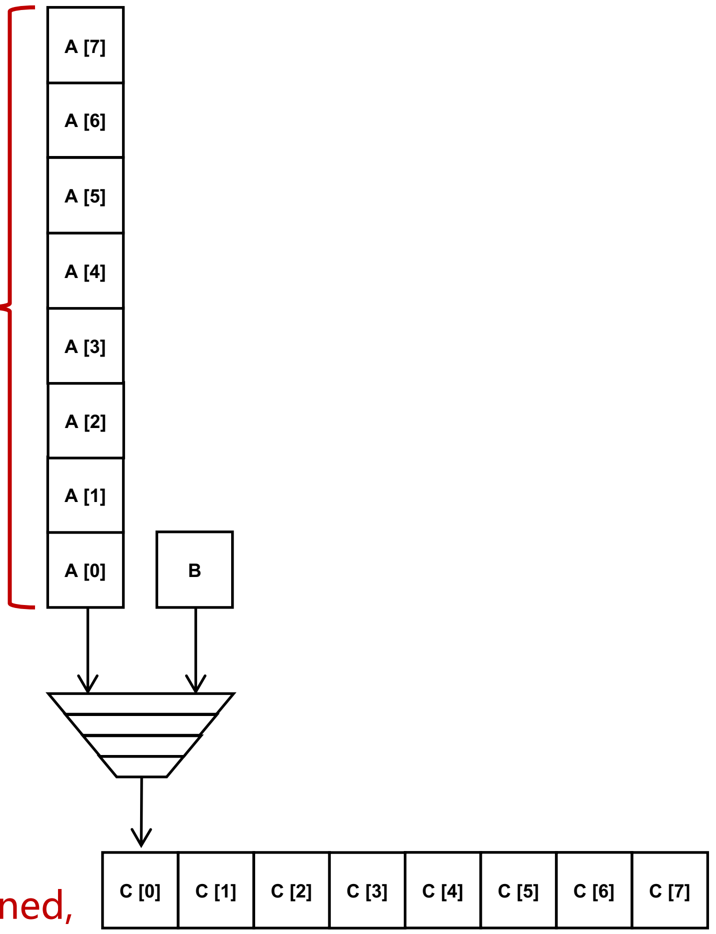
RVV Vector Processor

Functional Unit Pipelining: Different processes work simultaneously.

VLEN* - Processor Design

Vector Register Group

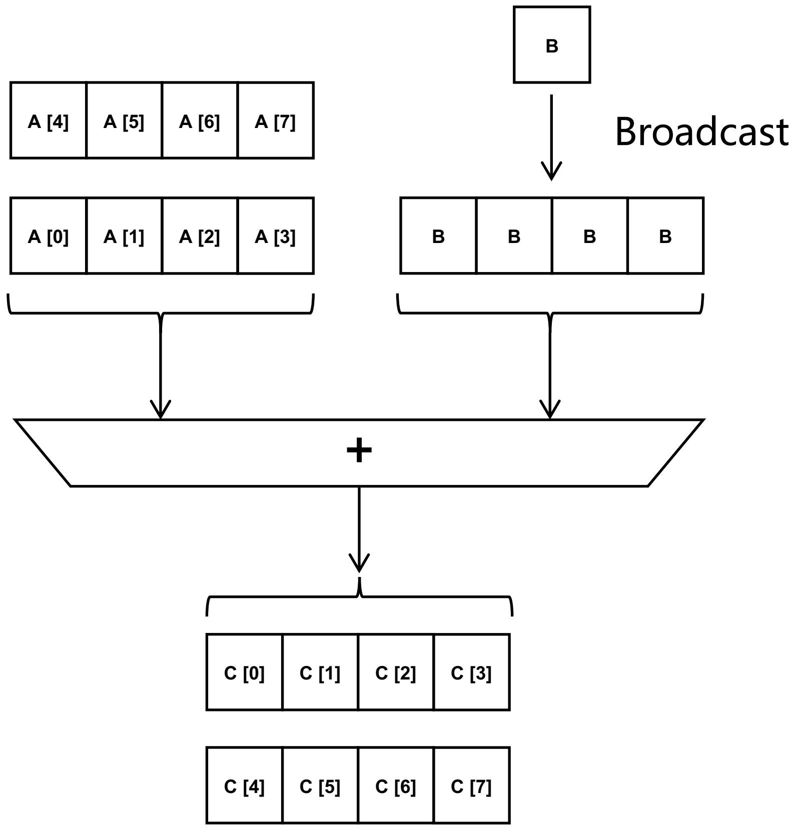
Dynamic VL* at Runtime



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* – Instruction Set

Fixed VL* at Runtime

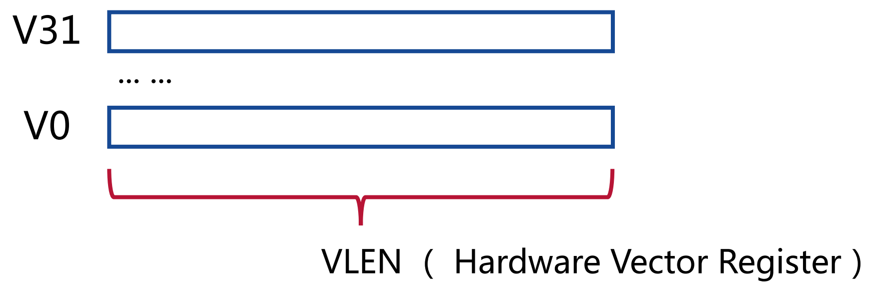
[1] Hennessy J L, Patterson D A. Computer architecture: a quantitative approach. Sixth Edition.

* VLEN = Hardware Vector Register Length   * VL = Processing Vector Length

Motivation: RVV Special Features

Buddy Compiler

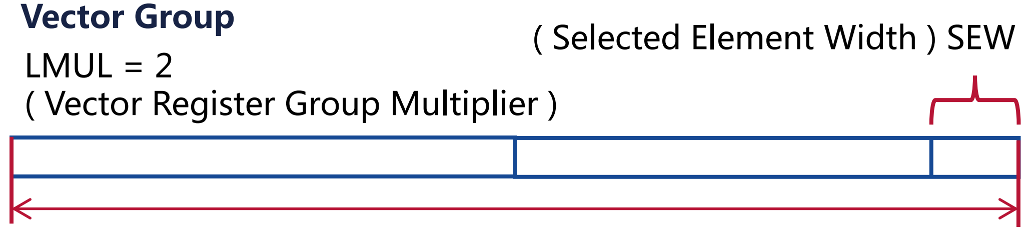
RVV Vector Register Configuration



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)

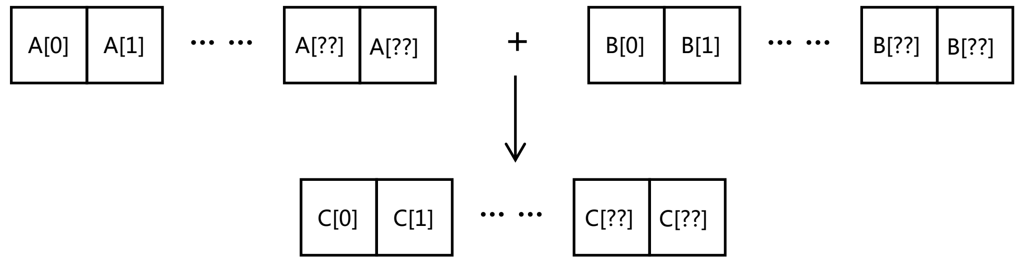
VLMAX assists in calculating the **VL** returned by the ***vsetvl*** command

(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

```
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
```

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
```

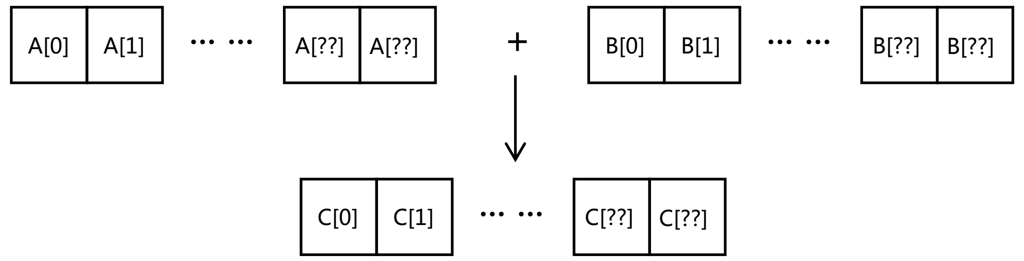
Motivation: RVV Special Features

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<?xi32>, vector<4xi32>
  %vec_input2 = affine.vector_load %input2[%idx * 4]
      : memref<?xi32>, vector<4xi32>
  %vec_output = arith.addi %vec_input1, %vec_input2 : vector<4xi32>
  affine.vector_store %vec_output, %output[%idx * 4]
      : memref<?xi32>, vector<4xi32>
}
// Tail processing
scf.if %tail {
  %cur_idx = arith.muli %iter, %c4 : index
  %mask = vector.create_mask %rem : vector<4xi1>
  %vec_input1 = vector.maskedload %input1[%cur_idx], %mask, %pass_thr
      : memref<?xi32>, vector<4xi1>, vector<4xi32> into vector<4xi32>
  %vec_input2 = vector.maskedload %input2[%cur_idx], %mask, %pass_thr
      : memref<?xi32>, vector<4xi1>, vector<4xi32> into vector<4xi32>
  %vec_output = arith.addi %vec_input1, %vec_input2 : vector<4xi32>
  vector.maskedstore %output[%cur_idx], %mask, %vec_output
      : memref<?xi32>, vector<4xi1>, vector<4xi32>
}
```

Tail processing with mask operations

RVV Tail Processing



Get the application vector length (d) at runtime

Mask-Based 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
```

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
```

Motivation : MLIR Limitation

Buddy Compiler

Information Required at Compile Time :

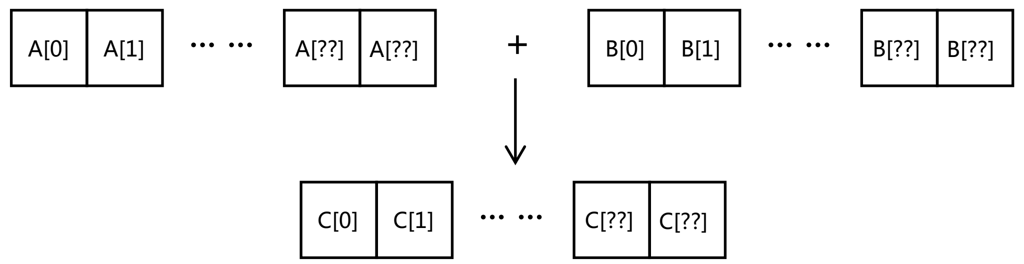
- Dynamic VL Configuration
  - AVL Configuration
  - LMUL Configuration
  - SEW Configuration
- Operations Dynamic VL Operand

No SETVL Operation
Cannot Set Dynamic VL

Vector operations do not accept dynamic VL parameters.

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

RVV Tail Processing



Get the application vector length (d) at runtime

Mask-Based 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
```

Strip-Mining Approach

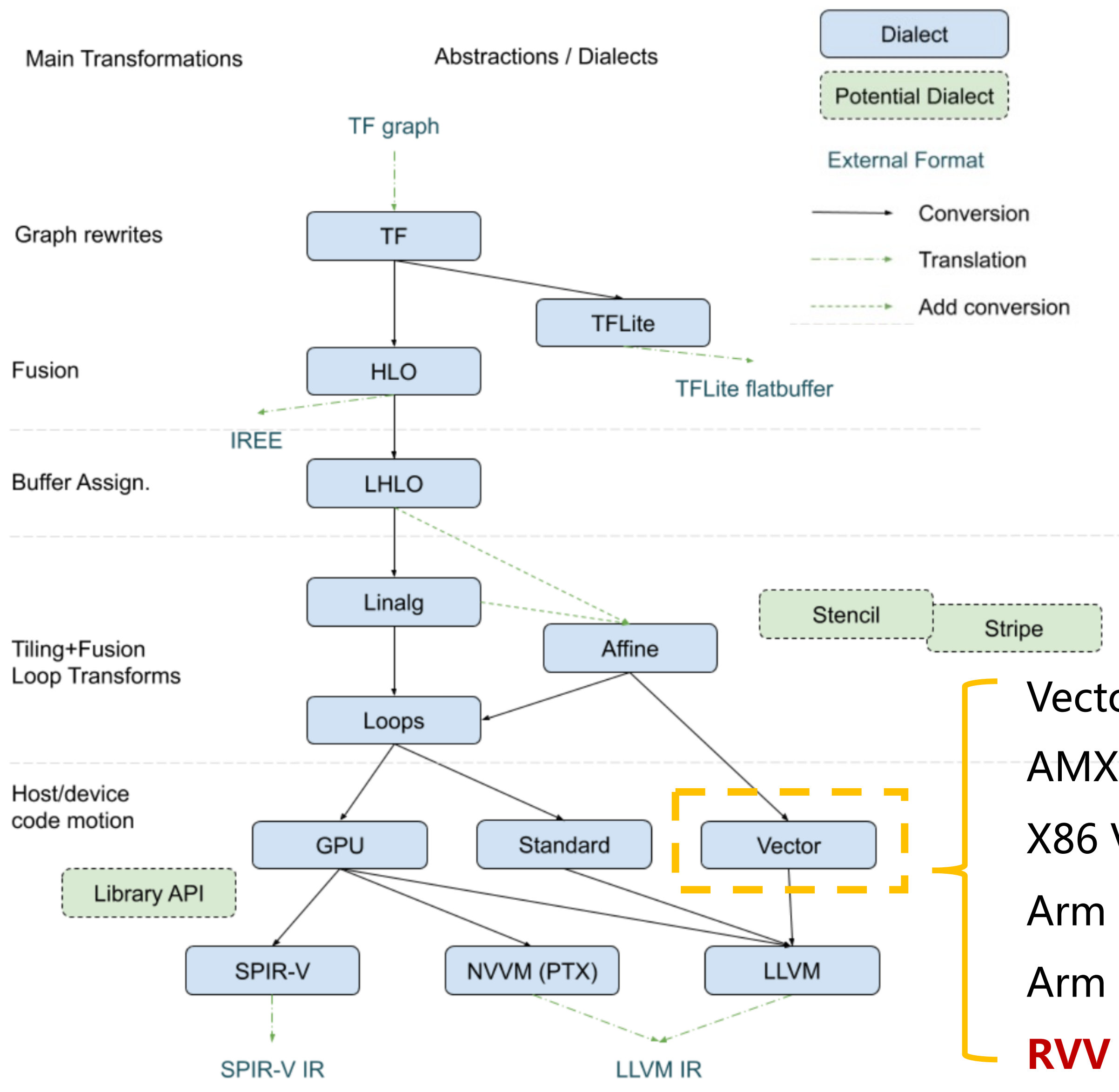
```
AVL = d
While(AVL > 0):
  do:
    ~~vl = setvl AVL , LMUL , SEW~~
    ~~vector load vl~~
    ~~vector add vl~~
    ~~vector store vl~~
    AVL = AVL - vl
End
```

MLIR Limitation

Set Dynamic VL

Ops Accept Dynamic VL

MLIR Abstraction Support for RVV Backend



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

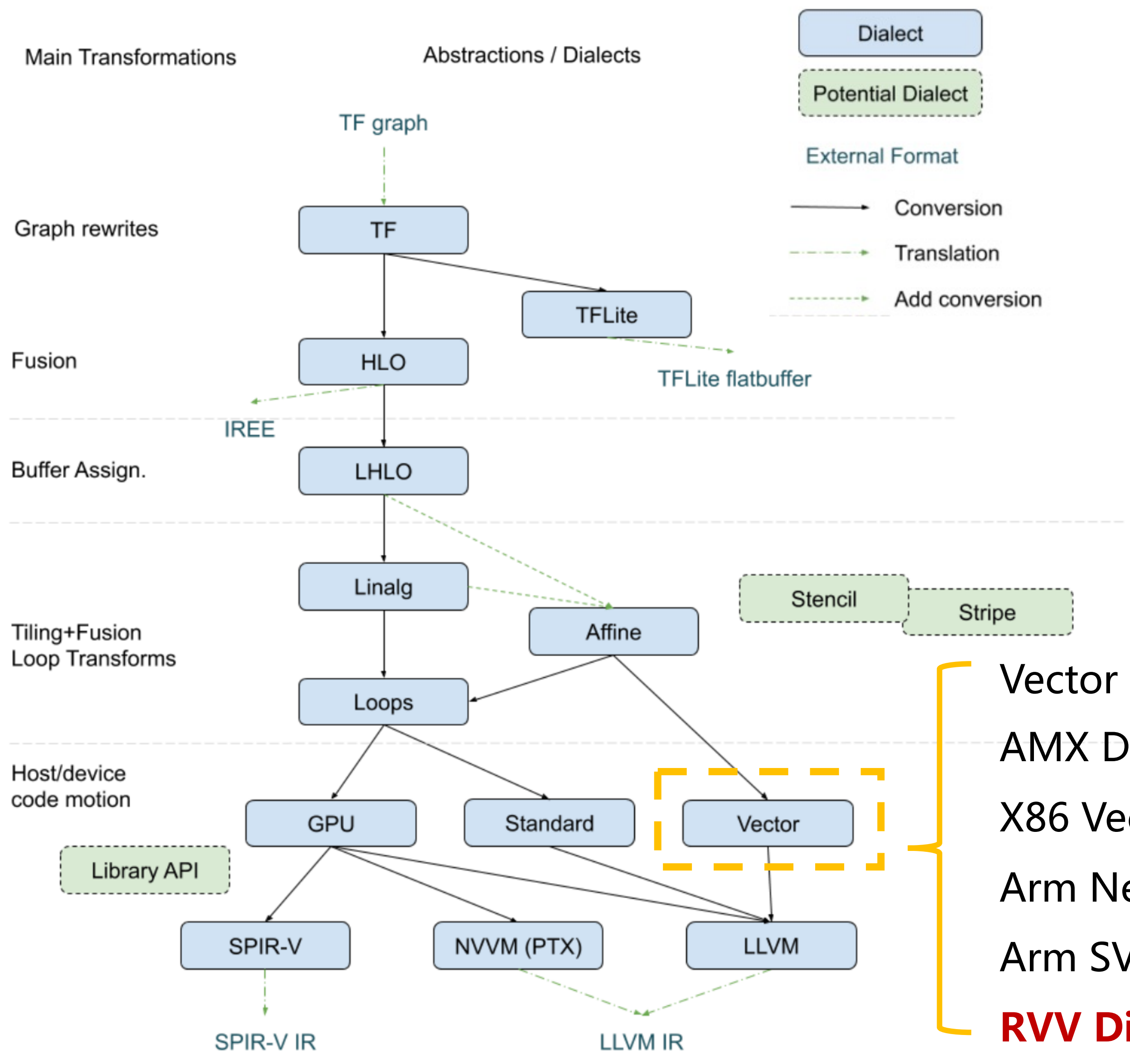
MLIR RISC-V Vector Dialect

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

MLIR Lowering Paths - https://mlir.llvm.org/docs/Dialects/Vector/

MLIR Abstraction Support for RVV Backend

Buddy Compiler



Vector Dialect

AMX Dialect

X86 Vector Dialect

Arm Neon Dialect

Arm SVE Dialect

RVV Dialect

MLIR RISC-V Vector Dialect

Fragmentation

- Operation
  - RVV Operation
  - RVV Intrinsic Operation
- Type

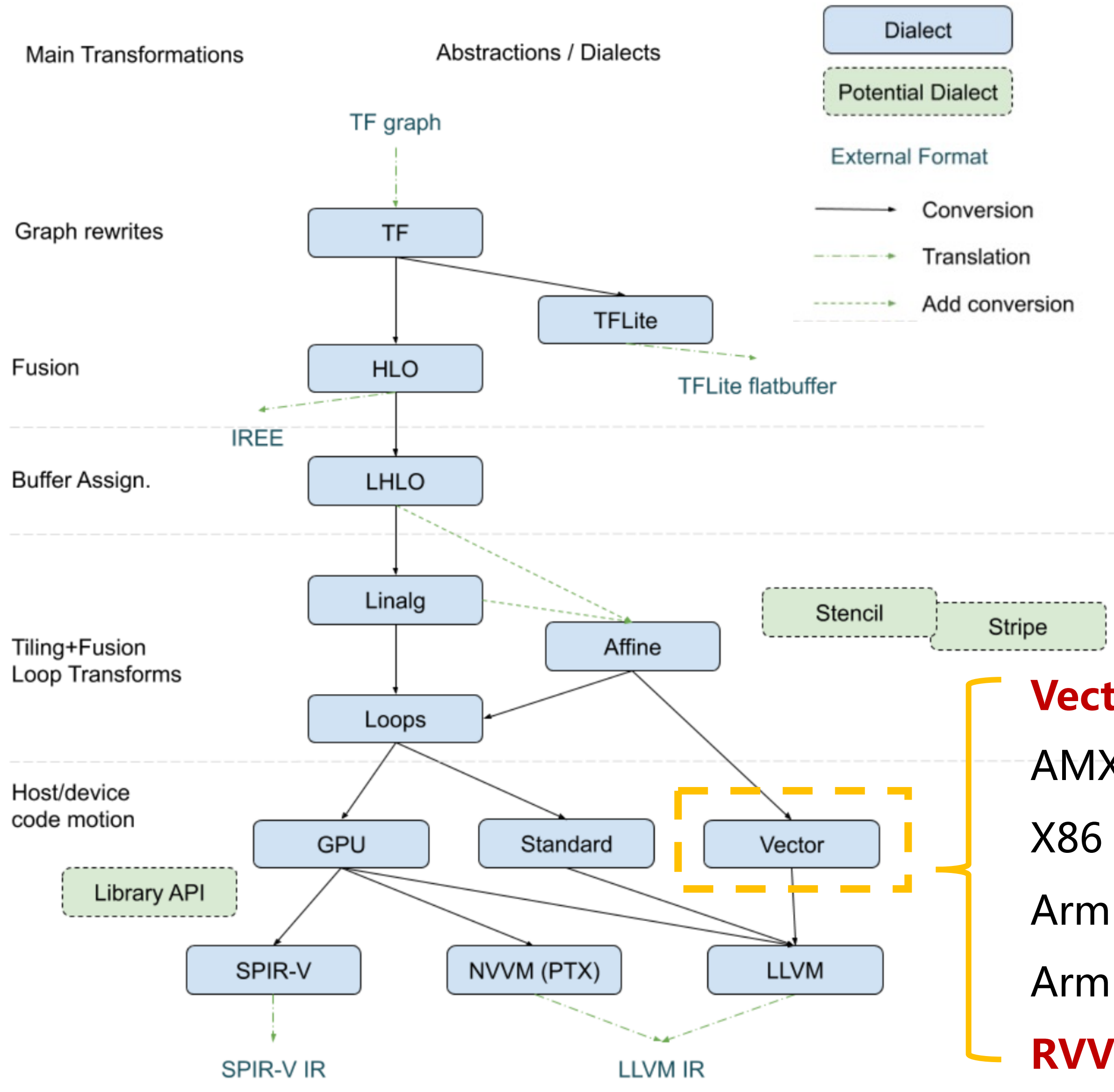
Upstream

  - Scalable Vector Type
- Conversion/Translation
  - RVV Dialect
  - LLVM Dialect
  - LLVM IR
- Integration Test

MLIR Lowering Paths - https://mlir.llvm.org/docs/Dialects/Vector/

MLIR Abstraction Support for RVV Backend

Buddy Compiler



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 $\longrightarrow$ **Add abstraction support for dynamic VL.**

AMX Dialect

X86 Vector Dialect

Arm Neon Dialect

Arm SVE Dialect

RVV Dialect $\longrightarrow$ **Add abstraction support for RVV-specific ops.**

MLIR Lowering Paths - https://mlir.llvm.org/docs/Dialects/Vector/

MLIR Abstraction Support for RVV Backend

```
// While loop for strip-mining.
%tmpAVL, %tmpIdx = scf.while (%avl = %dim, %idx = %c0)
    : (index, index) -> (index, index) {
  // If avl greater than zero.
  %cond = arith.cmpi sgt, %avl, %c0 : index
  // Pass avl, idx to the after region.
  scf.condition(%cond) %avl, %idx : index, index
} do {
^bb0(%avl : index, %idx : index):
  // 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<[4]xi1>, i32 {
    %ele = vector.load %input1[%idx] : memref<?xi32>, vector<[4]xi32>
    vector.yield %ele : vector<[4]xi32>
  } : vector<[4]xi32>
  %vec_input2 = vector_exp.predication %mask, %vl_i32 : vector<[4]xi1>, i32 {
    %ele = vector.load %input2[%idx] : memref<?xi32>, vector<[4]xi32>
    vector.yield %ele : vector<[4]xi32>
  } : vector<[4]xi32>
  %result_vector = rvv.add %vec_input1, %vec_input2, %vl
      : vector<[4]xi32>, vector<[4]xi32>, index
  vector_exp.predication %mask, %vl_i32 : vector<[4]xi1>, i32 {
    vector.store %result_vector, %output[%idx] : memref<?xi32>, vector<[4]xi32>
    vector.yield
  } : () -> ()
  // Update idx and avl.
  %new_idx = arith.addi %idx, %vl : index
  %new_avl = arith.subi %avl, %vl : index
  scf.yield %new_avl, %new_idx : index, index
}
```

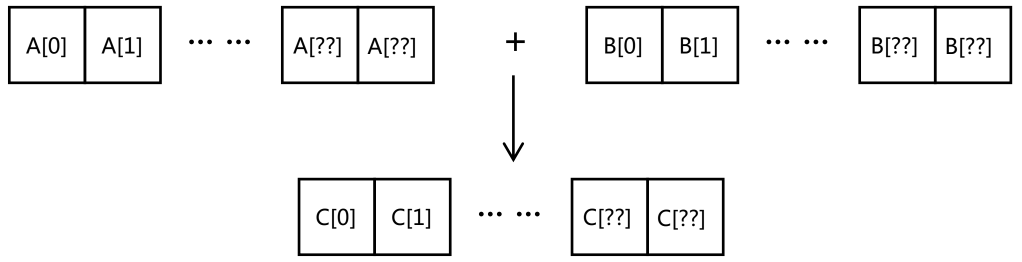
Set Dynamic Vector Length

Scalable Vector Type

Predication Region

Dynamic Vector Length

RVV Tail Processing



Get the application vector length (d) at runtime

Mask-Based Approach

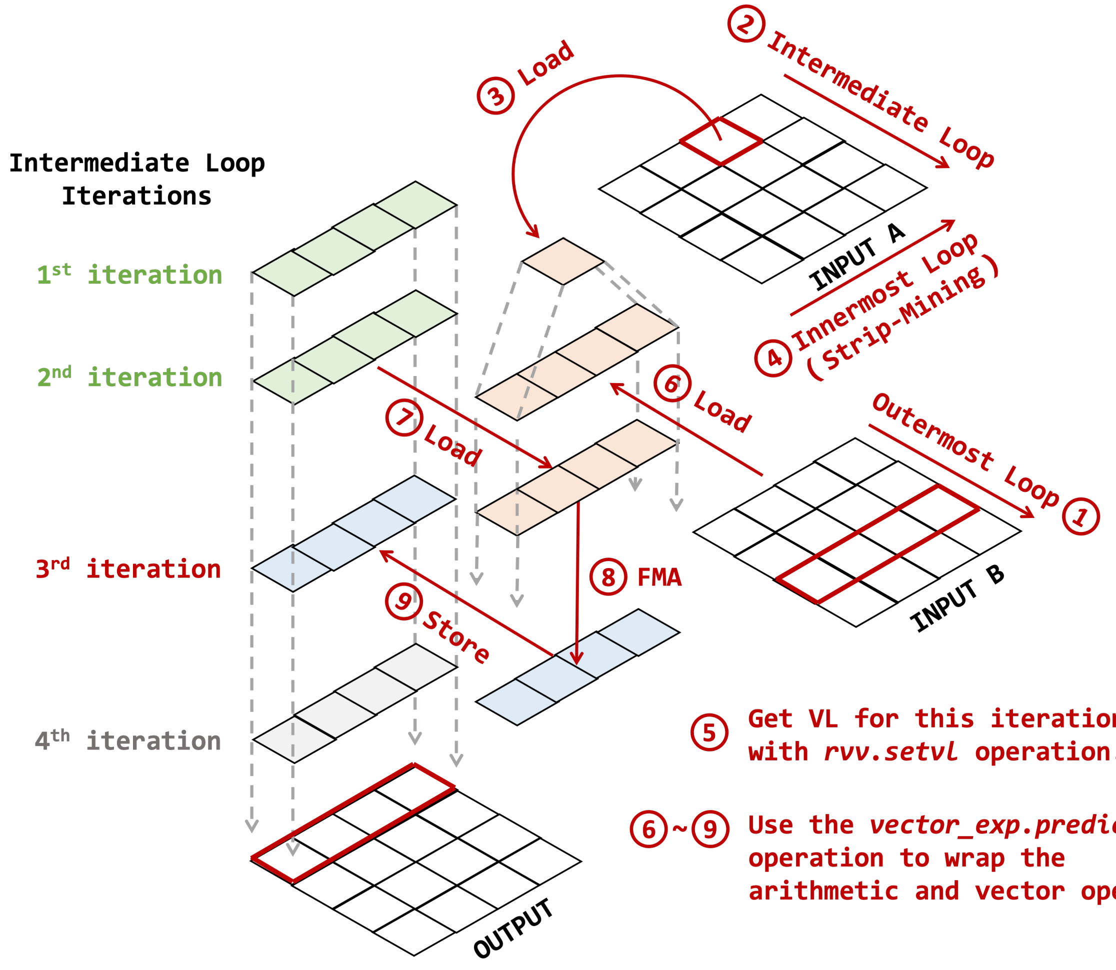
```
Tail = getTail(d)
Loop:
  if (not Tail)
    vector load
    vector add
    vector store
  else
    calculate mask
    masked load
    masked add
    masked store
  end if
End loop
```

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
```

Application: MatMul Optimization

Buddy Compiler



⑤ Get VL for this iteration with *rvv.setvl* operation.

⑥~⑨ Use the *vector_exp.predication* operation to wrap the arithmetic and vector operations.

```
affine.for %idx0 = 0 to %bRow {
  affine.for %idx1 = 0 to %aRow {
    %aEle = affine.load %mem_i32[%idx1, %idx0] : memref<10x10xi32>
    // While loop for strip-mining.
    %tmpAVL, %tmpIdx = scf.while (%avl = %bCol, %idx = %c0) : (index, index) -> (index, index) {
      // If avl greater than zero.
      %cond = arith.cmpi sgt, %avl, %c0 : index
      // Pass avl, idx to the after region.
      scf.condition(%cond) %avl, %idx : index, index
    } do {
    ^bb0(%avl : index, %idx : index):
      // Perform the calculation according to the vl.
      %vl = rvv.setvl %avl, %sew, %lmul : index                Step 5
      %vl_i32 = arith.index_cast %vl : index to i32
      %mask = vector.create_mask %vl : vector<[4]xi1>
      %input_vector = vector_exp.predication %mask, %vl_i32 : vector<[4]xi1>, i32 {
        %ele = vector.load %mem_i32[%idx0, %idx] : memref<10x10xi32>, vector<[4]xi32>
        vector.yield %ele : vector<[4]xi32>
      } : vector<[4]xi32>
      %mul_vector = rvv.mul %input_vector, %aEle, %vl : vector<[4]xi32>, i32, index
      %c_vector = vector_exp.predication %mask, %vl_i32 : vector<[4]xi1>, i32 {
        %ele = vector.load %result_mem[%idx1, %idx] : memref<10x10xi32>, vector<[4]xi32>
        vector.yield %ele : vector<[4]xi32>
      } : vector<[4]xi32>
      %result_vector = rvv.add %mul_vector, %c_vector, %vl : vector<[4]xi32>, vector<[4]xi32>, index
      vector_exp.predication %mask, %vl_i32 : vector<[4]xi1>, i32 {
        vector.store %result_vector, %result_mem[%idx1, %idx] : memref<10x10xi32>, vector<[4]xi32>
        vector.yield
      } : () -> ()
      // Update idx and avl.
      %new_idx = arith.addi %idx, %vl : index
      %new_avl = arith.subi %avl, %vl : index
      scf.yield %new_avl, %new_idx : index, index
    }
  }
}
```

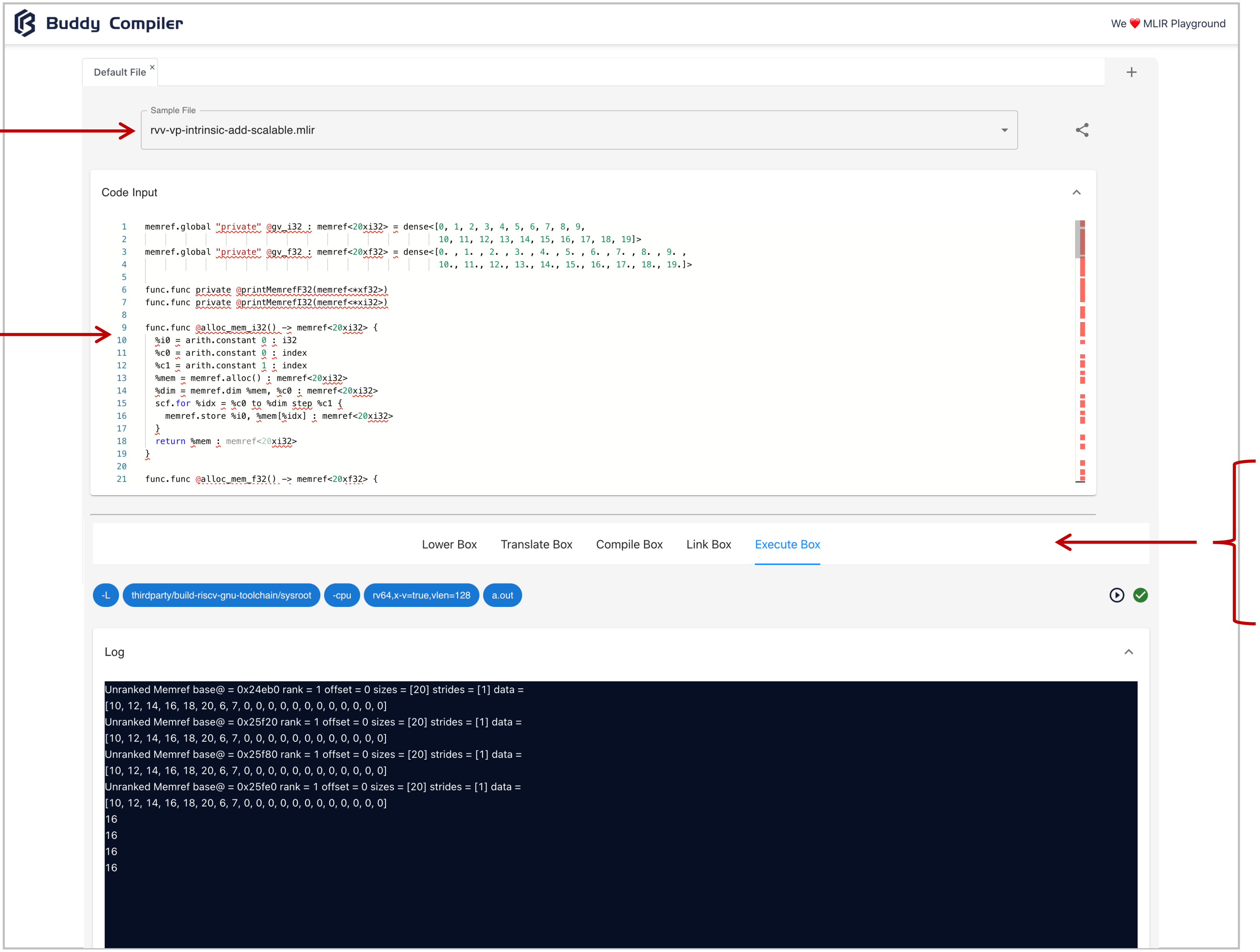
Step 6~9

Buddy Compiler As A Service: RVV Integration

Buddy Compiler

VP Intrinsic Example Cases

VP Intrinsic Example Code



rvv-vp-intrinsic-add-scalable.mlir

```
memref.global "private" @gv_i32 : memref<20xi32> = dense<[0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
                                                            10, 11, 12, 13, 14, 15, 16, 17, 18, 19]>
memref.global "private" @gv_f32 : memref<20xf32> = dense<[0. , 1. , 2. , 3. , 4. , 5. , 6. , 7. , 8. , 9. ,
                                                            10., 11., 12., 13., 14., 15., 16., 17., 18., 19.]>

func.func private @printMemrefF32(memref<*xf32>)
func.func private @printMemrefI32(memref<*xi32>)

func.func @alloc_mem_i32() -> memref<20xi32> {
  %i0 = arith.constant 0 : i32
  %c0 = arith.constant 0 : index
  %c1 = arith.constant 1 : index
  %mem = memref.alloc() : memref<20xi32>
  %dim = memref.dim %mem, %c0 : memref<20xi32>
  scf.for %idx = %c0 to %dim step %c1 {
    memref.store %i0, %mem[%idx] : memref<20xi32>
  }
  return %mem : memref<20xi32>
}

func.func @alloc_mem_f32() -> memref<20xf32> {
```

```
-L  thirdparty/build-riscv-gnu-toolchain/sysroot  -cpu  rv64,x-v=true,vlen=128  a.out
```

```
Unranked Memref base@ = 0x24eb0 rank = 1 offset = 0 sizes = [20] strides = [1] data =
[10, 12, 14, 16, 18, 20, 6, 7, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Unranked Memref base@ = 0x25f20 rank = 1 offset = 0 sizes = [20] strides = [1] data =
[10, 12, 14, 16, 18, 20, 6, 7, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Unranked Memref base@ = 0x25f80 rank = 1 offset = 0 sizes = [20] strides = [1] data =
[10, 12, 14, 16, 18, 20, 6, 7, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Unranked Memref base@ = 0x25fe0 rank = 1 offset = 0 sizes = [20] strides = [1] data =
[10, 12, 14, 16, 18, 20, 6, 7, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
16
16
16
16
```

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.

Buddy Compiler

Thanks

hongbin2019@iscas.ac.cn