# An End-to-End Tool Flow with Intrinsic-Level Kernel Optimization on Versal ACAP

Liyang Dou[1], Zhe Lin[2†], Kai Shi[1], Xinya Luan[1], Kang Zhao[1†]

[1] Beijing University of Posts and Telecommunications
[2] Sun Yat-sen University


# Contents

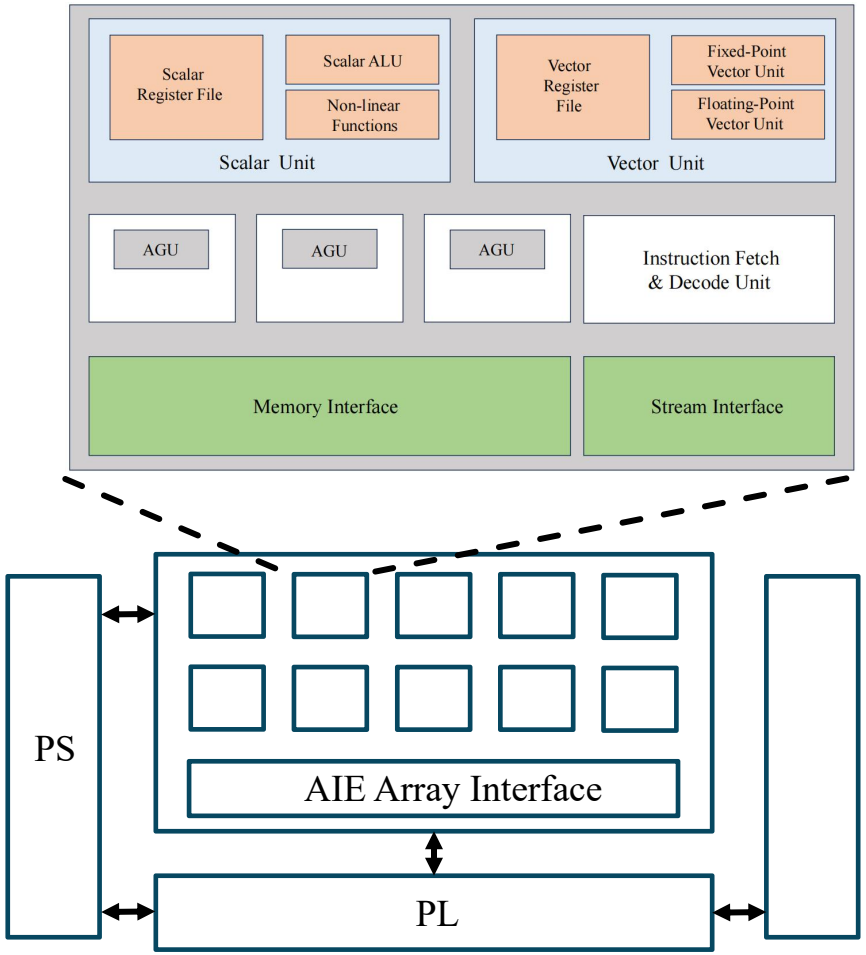
- Introduction
- Framework
- Evaluation
- Conclusion

# Part1: Introduction

# Background: Versal ACAP

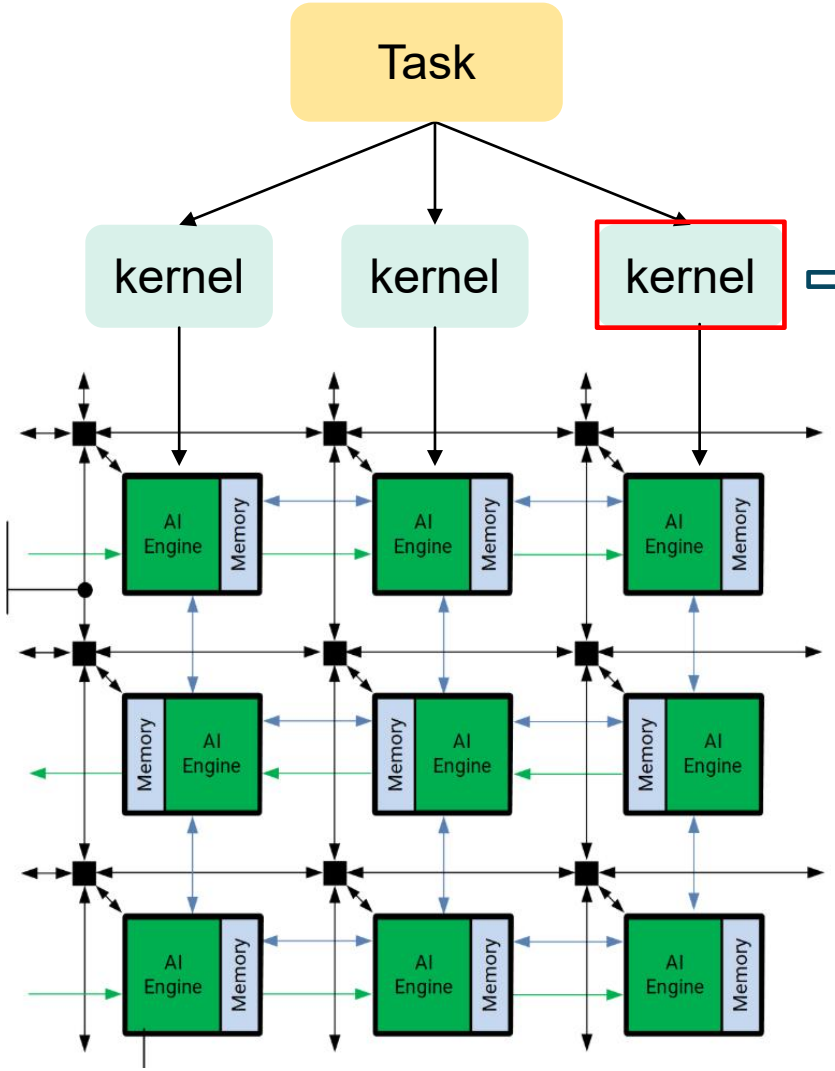
Performance & Energy efficiency ⇨ Monolithic arch ➡ Heterogenous arch

AIE




- **AI Engine**
  - VLIW & SIMD processors
  - Various communication mechanism
- **PS**
  - Arm processor
- **PL**
  - Larger on-chip memory
  - Multiple IOs between AIEs and PL

# Background： AIE Intrinsics




Fine grained control

**AIE Intrinsic API**

**Memory System**

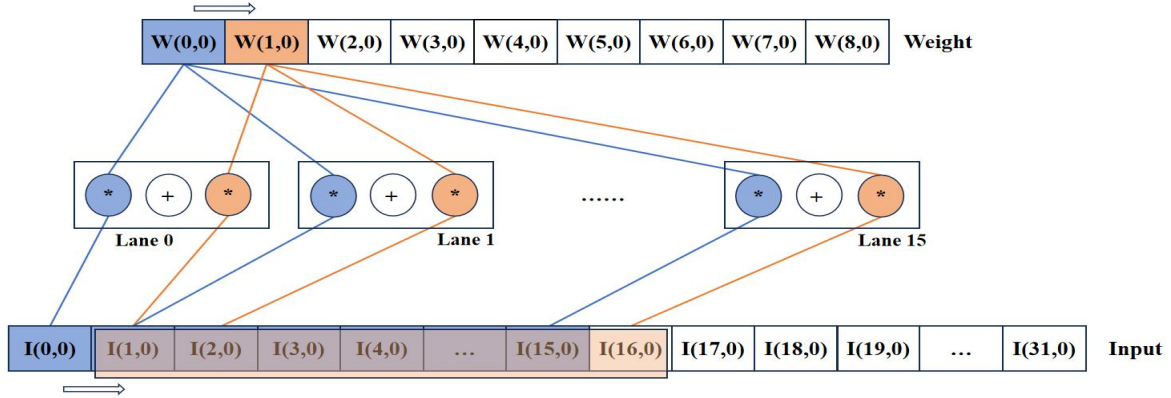
- Instruction Mem(16KB) + Data Mem (32KB)
- Register files
  - 32 scalar registers
  - 16 X 256-bit vector registers
  - 8 X 384-bit accumulator registers

**Function Unit**

- 32-bit RISC scalar core
- Two SIMD Units
  - Conventional SIMD for floating point
  - 2D SIMD for fixed point
- A configurable shuffle network that supports flexible data selection

# Background: Key features for kernel generation

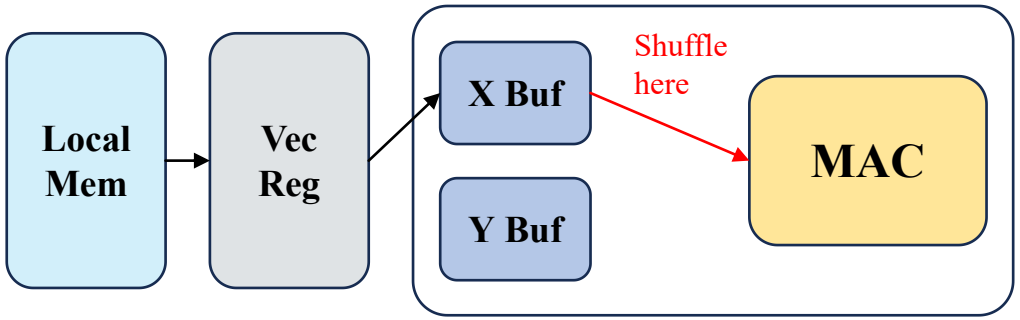
- **Feature1**: 2D SIMD Datapath
  - Lanes are parallel execution units.
  - Columns represent the number of dependant MAC operations per lane that can be reduced into a final result.

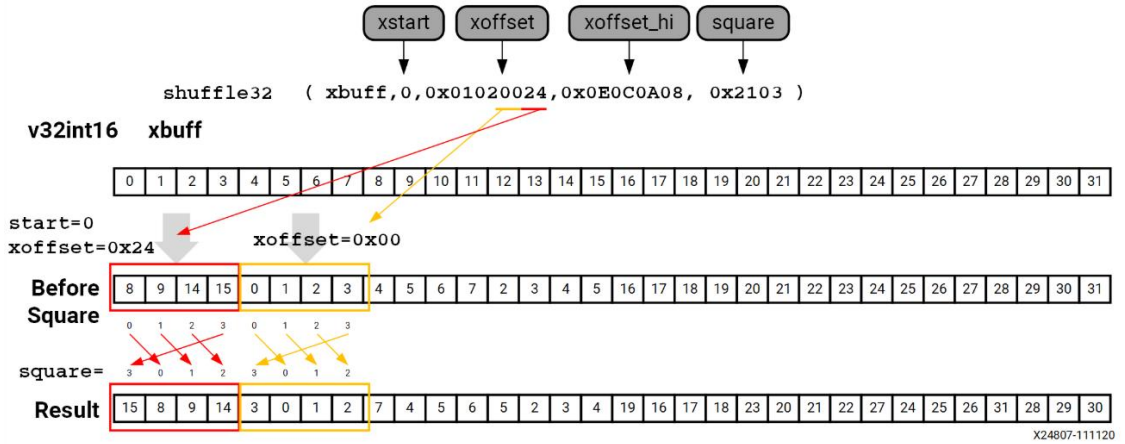



Grouped op $O(0:15,0) = W(0,0) * I(0:15,0) + W(1,0) * I(1:16,0)$

acc 1D SIMD 1D SIMD

- **Feature2**: Shuffle Network
  - Support flexible data selection between buffer and function units.
  - Manual config via intrinsic attributes.



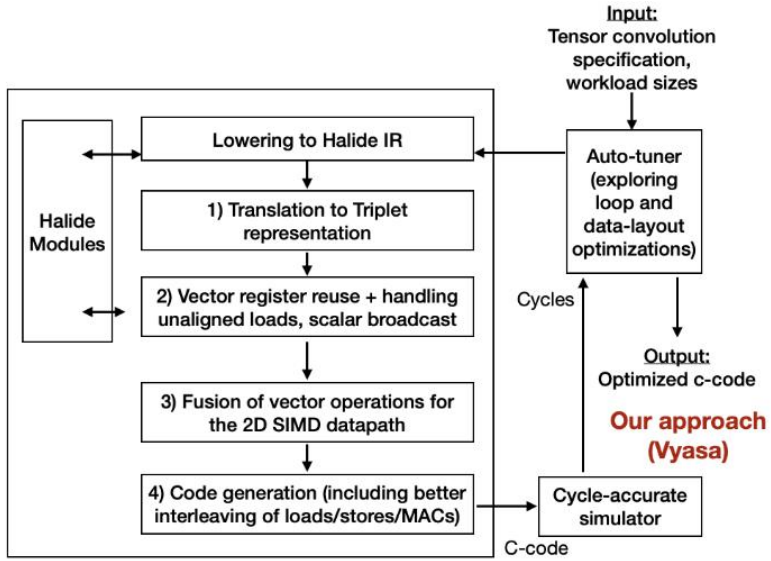
# Challenges: kernel developing with AIE intrinsics

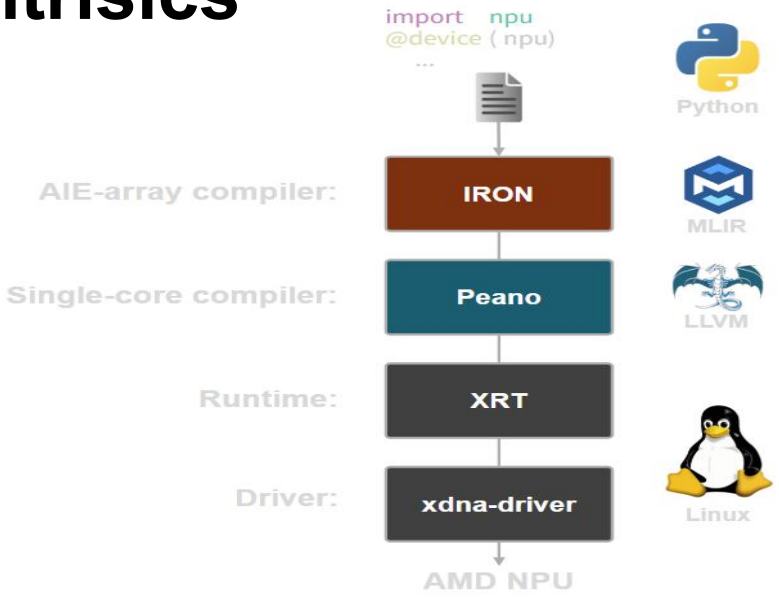
- **Challenge 1: Programming Complexity.**
  low-level intrinsics expose many hardware details:
  - registers
  - data selection
  - function units

  Many manual works need to be done, difficult to develop

- **Challenge 2: Hardware Utilization.**
  Effective use of 2D SIMD and VLIW relies on manual vectorization and scheduling, making performance optimization difficult.

# Related works: compiler targeting AIE Intrisics




## Vyasa

- Early kernel compiler work
- Based On Halide
- Support convolution workload

- **Limitation**
  - Hard to integrate into MLIR ecosystem
  - Constrained to convolution only
  - Generall purpose optimizations

## MLIR-AIE

- AMD's newlest opensource framework
- Support basic optimizatons (Vyasa included)

- **Limitation**
  - Not end-to-end (Python DSL / MLIR Affine IR)
  - Generall purpose optimizations

# Our work

**Programming Complexity**

- Low level control of register
- Data selection configuration
- Function Unit configuration

**Hardware Utilization**

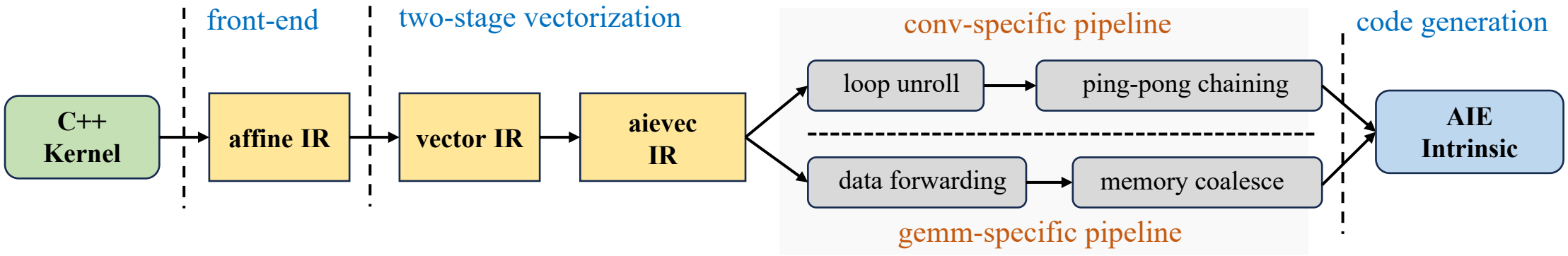
- SIMD vectorization
- Hard to optimize targeting different workload patterns

**Our work**

- **End to end tool flow**, supporting the compilation from C++ code to AIE intrinsics.
  - Support **2D operators** (Convolution and matrix multiplication) and **1D operators**.
- **Pattern specific optimizations** for convolution and matrix multiplication.

# Part2: Framework

# Framework




- Polyhedral model
- Integrate into MLIR eco system

- Upstrem mlir
- Hardware-agnostic
- Virtual vectors

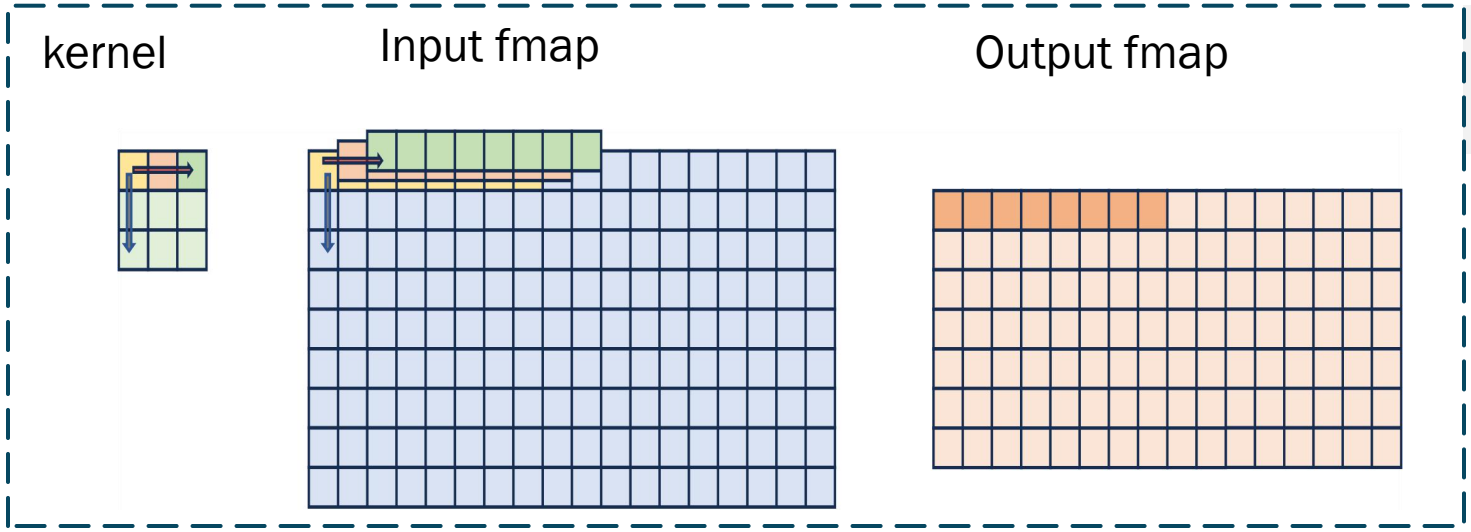
- Mlir-aie
- Hardware-specific
- 2D SIMD opt

- Our pattern specific optimization
- AIE-specific

- Simulated by aiesimulator
- Direct mapping

# Convolution Vectorization

- Step1: Sliding window strategy on each line of the kernel




Each line of the kernel, need three mac/mul ops

- Step2: AIE-specific vectorization (2D SIMD optimization)

Bundled into one mul/mac

O(0:15,0) = I(0:15,0)*W(0,0) + I(1:16,0)*W(1,0) + I(2:17,0)*W(2,0)

Two mac/mul ops now

# Convolution Vectorization

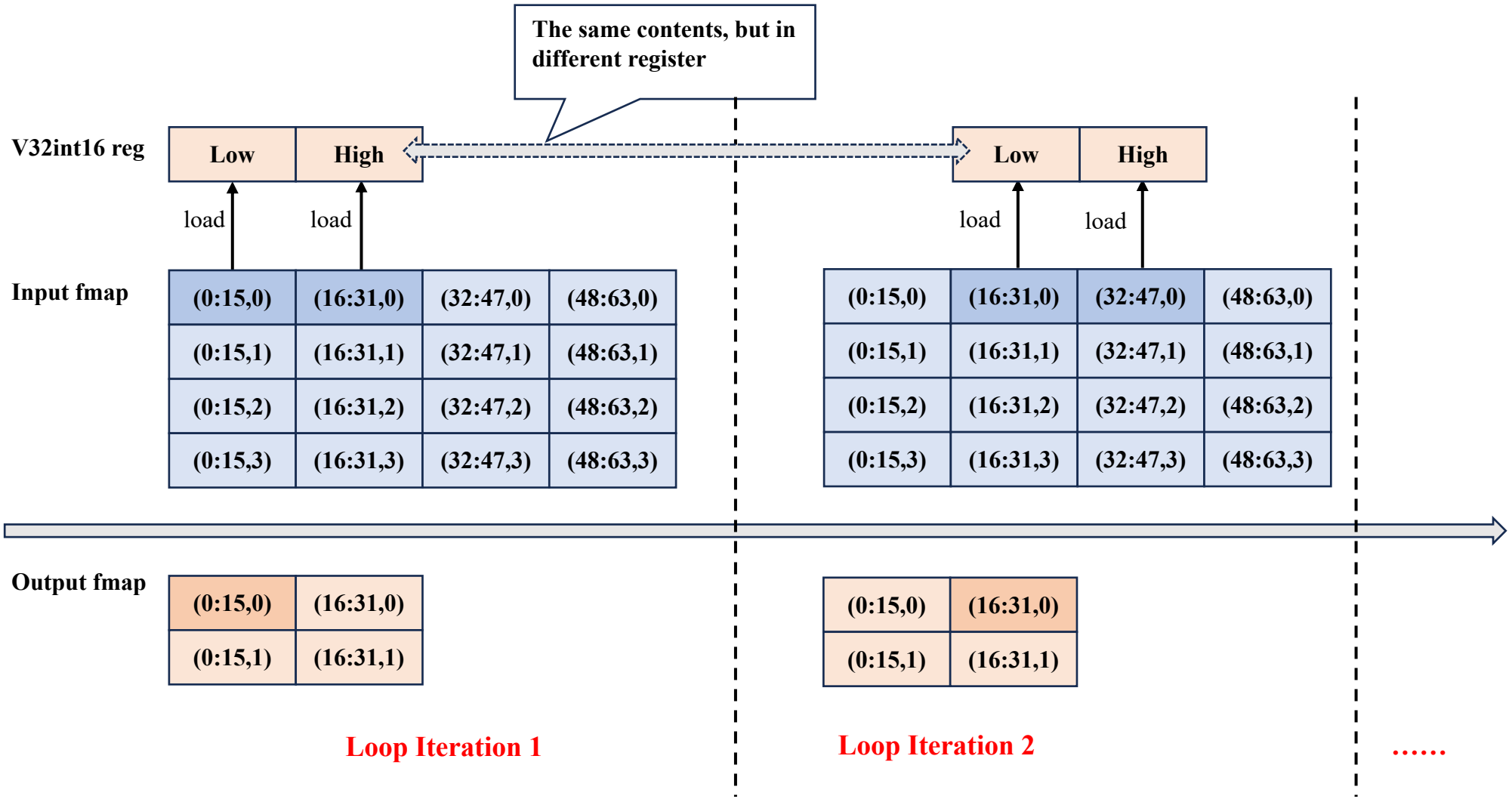
```
func.func @conv2d(%arg0, %arg1, %arg2) {
  ...
  %0 = aievec.upd %arg1[%c0, %c0] {index = 0, offset = 0}   ① loading weight
  ...
  scf.for %arg3 = %c0_0 to %c32 step 1{
    ...
    scf.for %arg4 = %c0_2 to %c32_3 step 16{                                   row1
      %3 = aievec.upd %arg0[%arg3, %arg4] {index = 0, offset = 0}   ① loading input
      %4 = aievec.upd %arg0[%arg3, %arg4], %3 {index = 1, offset = 256}
      %5 = aievec_aie1.mul %4, %0 {xoffsets = "0x03020100", xoffsets_hi = "0x07060504",  xstart = "0"}
      %6 = aievec_aie1.mac %4, %0, %5 {xoffsets = "0x03020100", xoffsets_hi = "0x07060504", xstart = "2"}
      …                                                                        row3
      %11 = aievec.upd %arg0[%2, %arg4] {index = 0, offset = 0}
      %12 = aievec.upd %arg0[%2, %arg4], %11 {index = 1, offset = 256}
      %13 = aievec_aie1.mac %12, %0, %10 {xoffsets = "0x03020100", xoffsets_hi = "0x07060504", xstart = "0"}
      %14 = aievec_aie1.mac %12, %0, %13 {xoffsets = "0x03020100", xoffsets_hi = "0x07060504", xstart = "2"}

      %15 = aievec.srs %14, 32
      vector.transfer_write %15, %arg2[%arg3, %arg4]   ④ result storage
    }
  }
  return
}
```

② shuffle network config

③ multiplication

# Potential reuse opportunity



## Potential reuse opportunity

- Within two iterations, half of the Vector store the same contents.
  - (0:31) and (16:47) are overlapped
  - (16:31) should be in high buffer In iter1 but low buffer in iter2 (buffer swap needed)

## What we do?

- Unroll loop
- Analyze load op pattern
  - Find overlaps
- Shuffle network configuration
  - Make data selection correct
- Eliminate useless load op

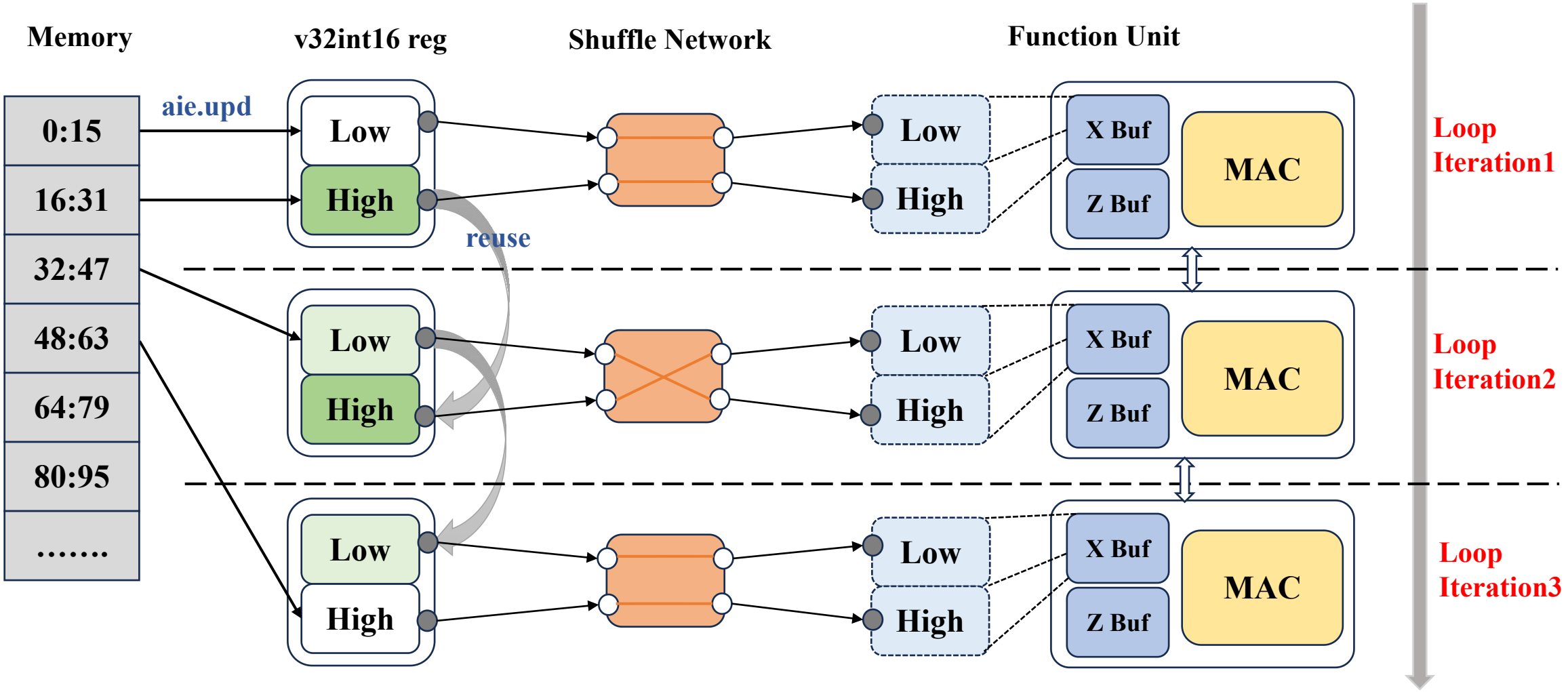
## What we have?

- Hierarchical vector registers
  - 2 v16int16 -> v32int16
  - Independent reg
- Shuffle network
  - Fine-grained control
  - Config in mac/mul intrinsic

## What we get?

- Within 2 Iters, reduce 12 load ops to 9 load ops, save 3
- Theoritically, this method can propagate like a chain across multiple Iterations.

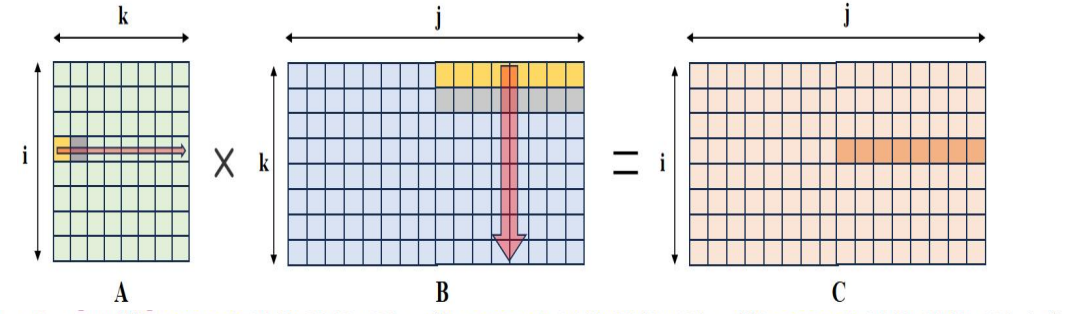
# Ping-pong chaining mechanism



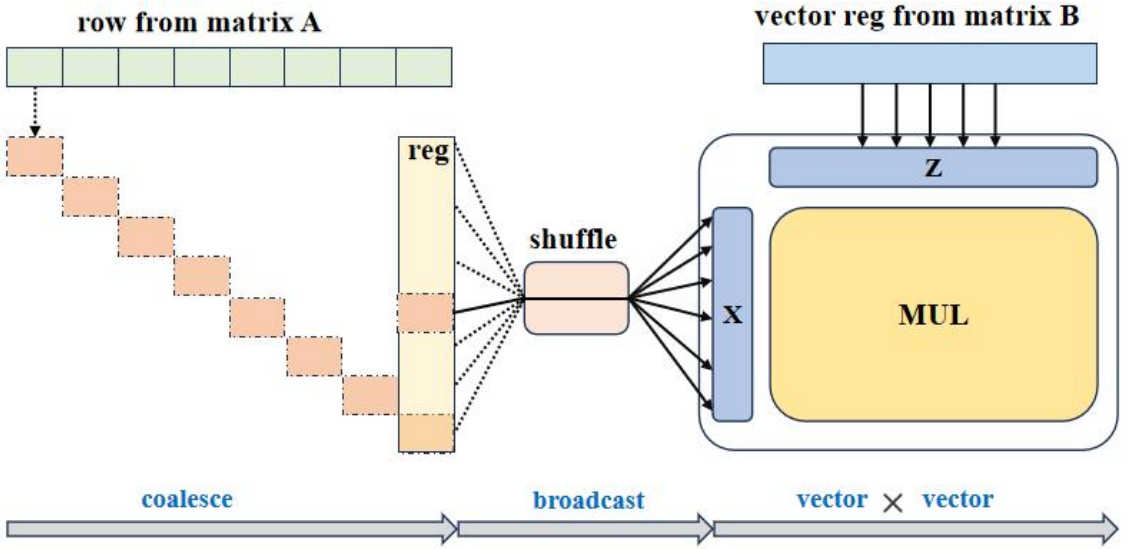
# Matrix Multiplication

## Scalar Coalescing

Load **a scalar** from A matrix in iteration ➡ load **a vector** from A matrix and shuffle and **broadcast** specific scalar needed in each iteration.




```
func @matmul(%A: memref<2048x2048xf64>, %B: memref<2048x2048xf64>, %C: memref<2048x2048xf64>) {
  %0 = memref_shape_cast %B : memref<2048x2048xf64> to memref<2048x512xvector<4xf64>>
  %1 = memref_shape_cast %C : memref<2048x2048xf64> to memref<2048x512xvector<4xf64>>
  affine.for %arg3 = 0 to 2048 {
    affine.for %arg4 = 0 to 512 {
      affine.for %arg5 = 0 to 2048 {
        %2 = affine.load %A[%arg3, %arg5] : memref<2048x2048xf64>
        %3 = splat %2 : vector<4xf64>
        %4 = affine.load %0[%arg5, %arg4] : memref<2048x512xvector<4xf64>>
        %5 = mulf %3, %4 : vector<4xf64>
        %6 = affine.load %1[%arg3, %arg4] : memref<2048x512xvector<4xf64>>
        %7 = addf %5, %6 : vector<4xf64>
        affine.store %7, %1[%arg3, %arg4] : memref<2048x512xvector<4xf64>>
      }
    }
  }
  return
}
```




Reduce load intrinsics, thus reduce access overhead

# Part3: Evaluation

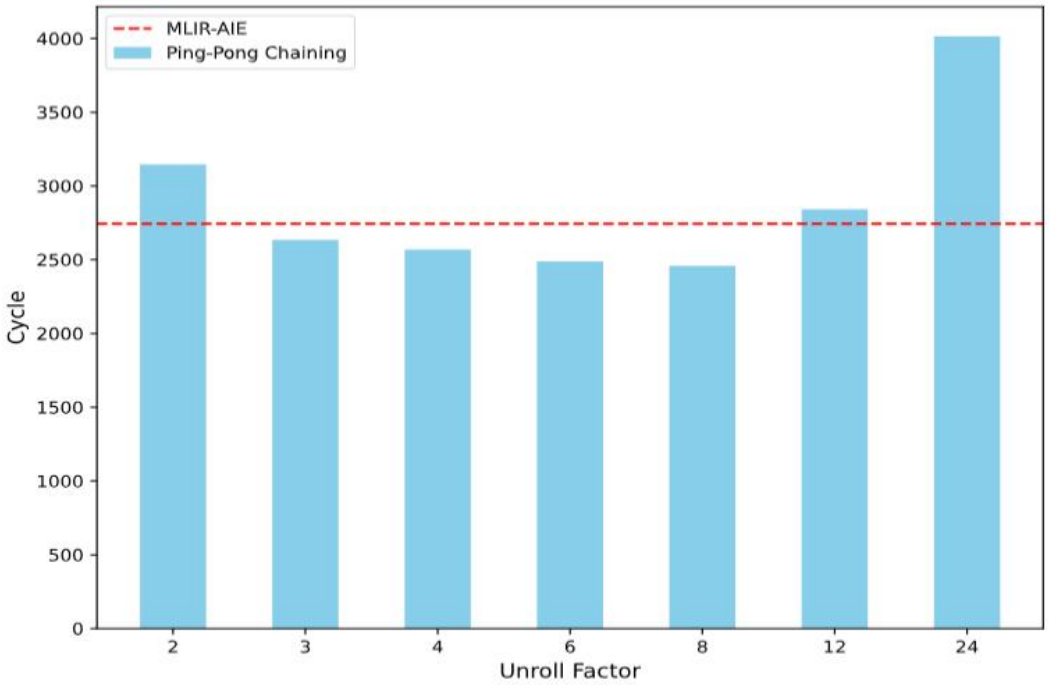
# Evaluation

- Experiment Setup
  - VCK190 platform (aiev1)
  - Aiesimulator
  - Baseline: MLIR-AIE with basic vectorization optimizations
- Result
  - Small size is good
    - Ease for VLIW optimization
  - Consistent speedup for large and medium sizes
    - Long loop body, hard for VLIW optimization, see more nops in assembly.
    - Register pressure, see register spill in assembly.

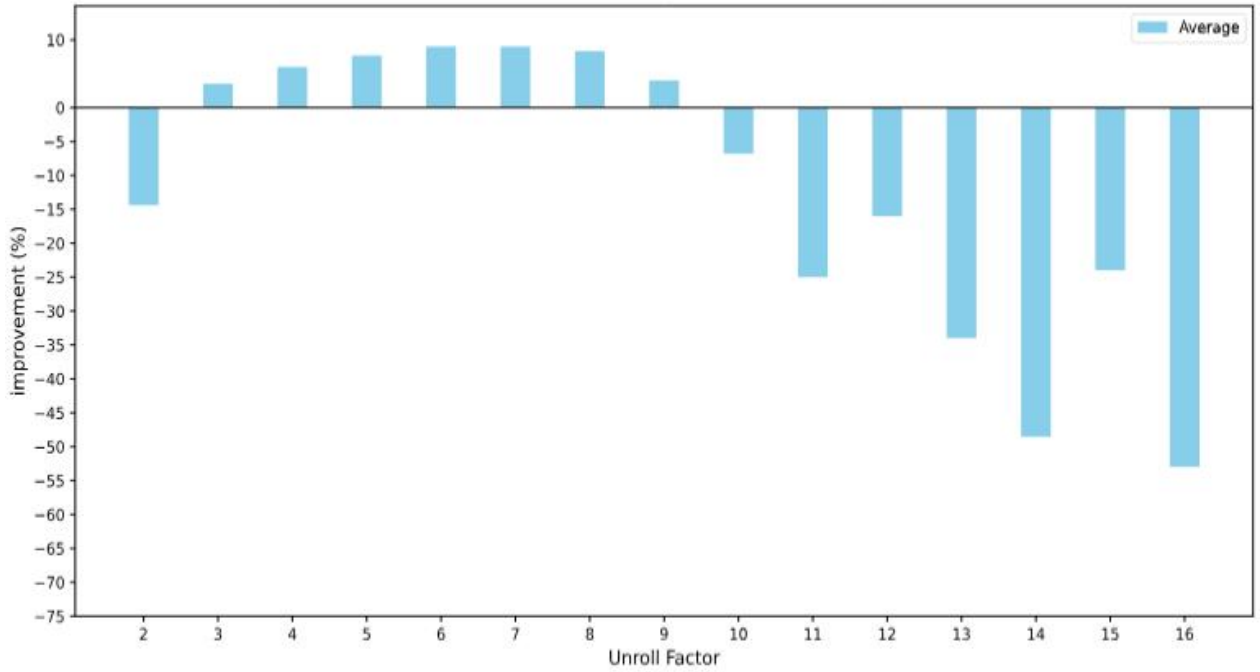
| Scale | Output Size | MLIR-AIE Baseline | Our Method | | Speedup |
|---|---|---|---|---|---|
| | | | Unroll Factor | Register Reuse | |
| small | 16 × 16 | 428 | 2 | 234 | 1.45× |
| | 32 × 32 | 828 | 2 | 442 | 1.46× |
| | 64 × 32 | 1628 | 2 | 858 | 1.53× |
| | 16 × 64 | 505 | 4 | 426 | 1.16× |
| | 32 × 64 | 985 | 4 | 826 | 1.16× |
| | 64 × 64 | 1943 | 4 | 1626 | 1.17× |
| medium | 16×160 | 1177 | 5 | 1098 | 1.07× |
| | 16×192 | 1403 | 6 | 1274 | 1.09× |
| | 16×224 | 1625 | 7 | 1482 | 1.09× |
| | 16×256 | 1849 | 4 | 1738 | 1.06× |
| | 16×288 | 2073 | 6 | 1882 | 1.09× |
| | 16×320 | 2297 | 5 | 2122 | 1.08× |
| large | 16×448 | 3193 | 7 | 2890 | 1.09× |
| | 16×480 | 3419 | 6 | 3100 | 1.09× |
| | 16×512 | 3643 | 8 | 3260 | 1.11× |

The longer the reuse chain, the better performane? → **Unroll factor matters!**

# Evaluation： parameter study




(a) Unroll Factor Impact on 16x384 convolution size.




(b) Average Unroll Factor Impact.

- For 16×384, the best performance appears at **UF=4-8**. **UF=2** lacks reuse chaining, and **UF=24** suffers from register pressure and poor VLIW scheduling.
- We further evaluate **16 convolution configurations** (column sizes 32 to 512, unroll factors 2 to 16) and observe that unroll factors between 3 and 9 deliver the best performance.

# Evaluation：weakenss of our work

**Weakness**

- VLIW is **black box** in current framework, unable to conduct qualitative analysis to help guide parameter selection.
- The current flow mainly targets a **single AIE core (Intrinsic kernel)** and does not yet support compilation at the **AIE-array level**, which would require a more sophisticated compilation framework (future work).

# Part4: Conclusion

# Conclusion

- Proposed an **MLIR-based** tool flow that automatically generates optimized **AIE intrinsic kernels** directly from C++.
- Supports a broad set of operators, including 1D operators, 2D convolutions, and matrix multiplication.
- Introduced a **ping-pong chaining mechanism** for convolution patterns, reducing memory–register transfer overhead.
- Demonstrated consistent performance gains over the MLIR-AIE baseline: **1.33×** speedup for small convolutions, **1.08×** speedup for medium and large workloads.

# Thank You