Vyasa: A High-performance Vectorizing Compiler for Tensor Convolutions onto Xilinx AI Engine

Prasanth Chatarasi*

Stephen Neuendorffer[+], Samuel Bayliss[+], Kees Vissers[+], and Vivek Sarkar*
Habanero Research Group, Georgia Tech*, and
Xilinx Research Labs[+]

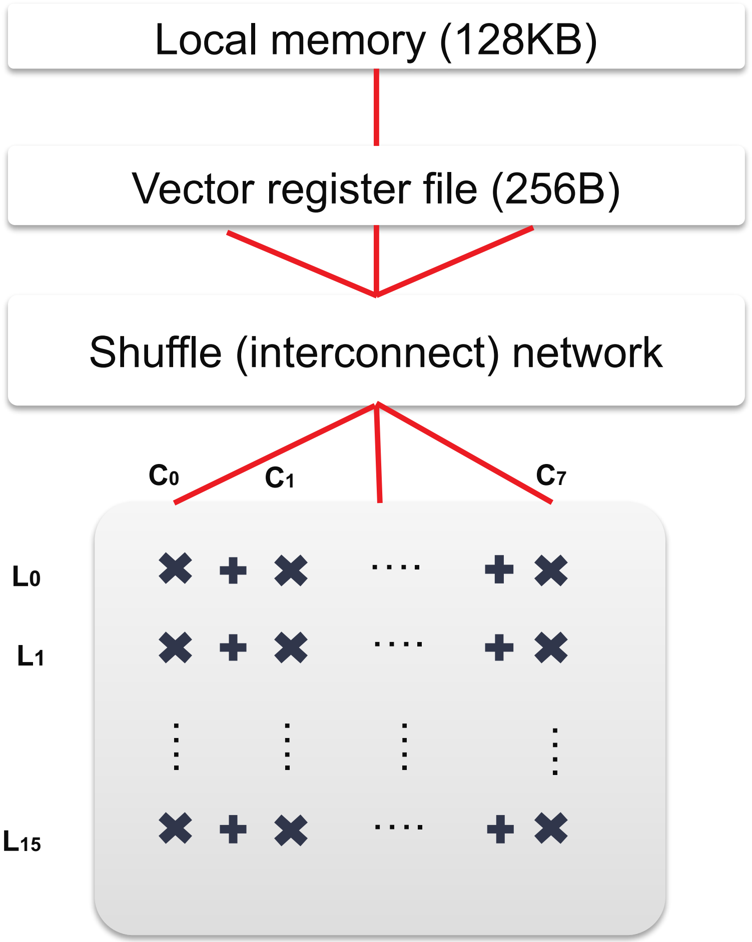
2020 IEEE High Performance Extreme Computing Virtual Conference
Sep 24th, 2020

Georgia Tech School of Computer Science

XILINX

Key architectural features of AI Engine

Abstract view of AI Engine



Local memory (128KB)

Vector register file (256B)

Shuffle (interconnect) network

Fixed Point SIMD Unit

1) 2D SIMD datapath for fixed point

- Reduction within a row/lane
- #Columns depend on operand precision
  - 32-bit types: 8 rows x 1 col
  - 16-bit types: 8 rows x 4 col (or)
    16 rows x 2 col
  - 8-bit types: 16 rows x 8 col

2) Shuffle Interconnection network

- Between SIMD and vector register file
- Supports arbitrary selection of elements from a vector register
  - Some constraints for 16-/8-bit types
- Selection parameters are provided via vector intrinsics

Problem Statement & Challenges

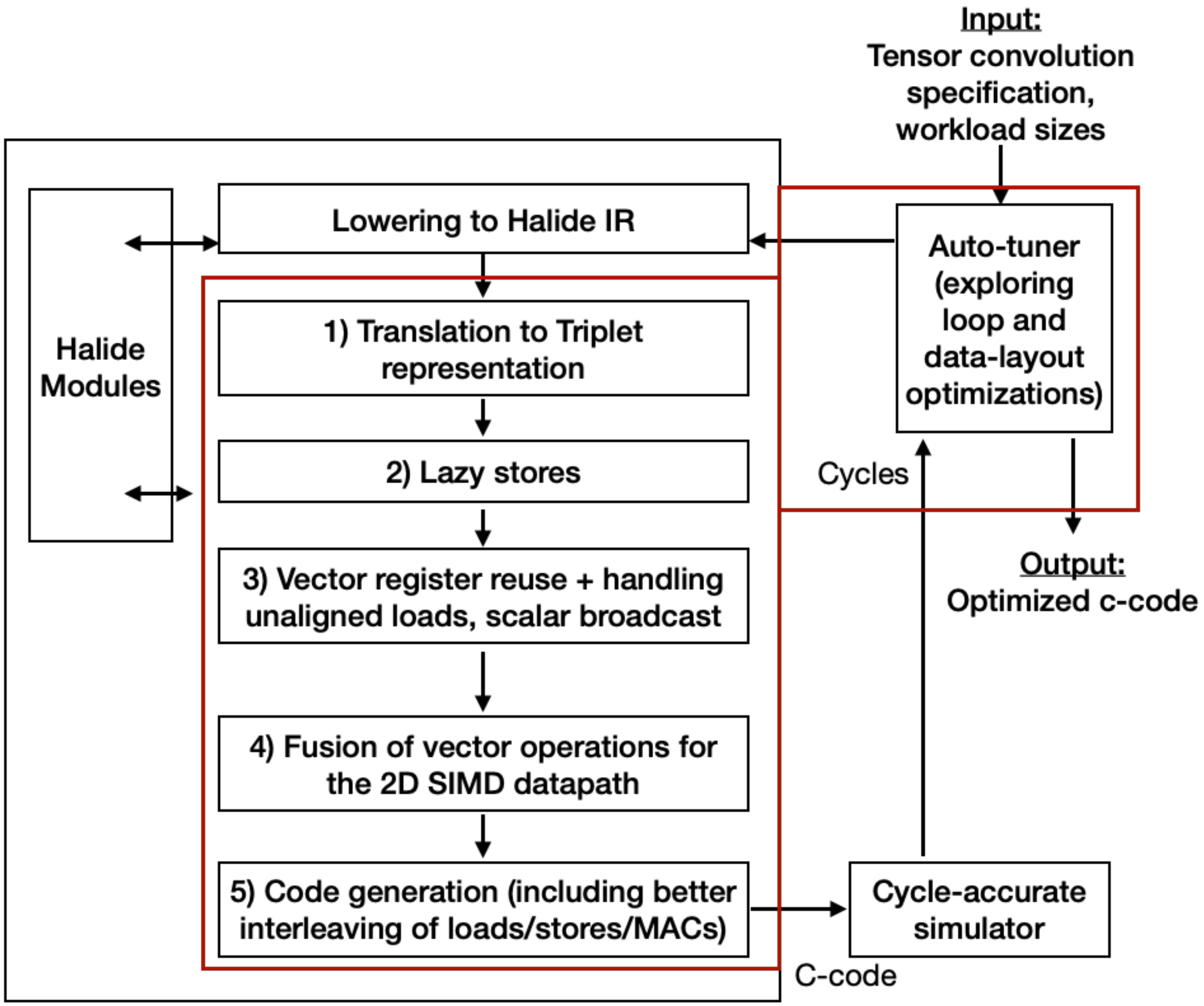
Problem statement: How to implement high-performance primitives for tensor convolutions on AI Engine?

- Current practice: Programmers manually use vector intrinsics to program 2D SIMD unit and also explicitly specify shuffle network parameters for data selection

- Challenges: Error prone, written code may not be portable to a different schedule or data-layouts, daunting to explore all choices to find best implementation, tensor convolutions vary in sizes and types

Our approach: Vyasa, a domain-specific compiler to generate high performance primitives for tensor convolutions from a high-level specification!

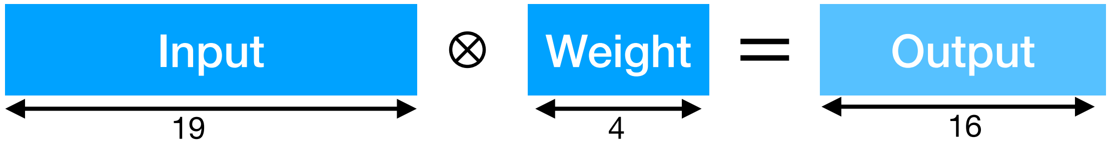
Our high-level approach (Vyasa)


Input:
Tensor convolution specification, workload sizes
Lowering to Halide IR
Halide Modules
1) Translation to Triplet representation
2) Lazy stores
3) Vector register reuse + handling unaligned loads, scalar broadcast
4) Fusion of vector operations for the 2D SIMD datapath
5) Code generation (including better interleaving of loads/stores/MACs)
Auto-tuner (exploring loop and data-layout optimizations)
Cycles
Output:
Optimized c-code
Cycle-accurate simulator
C-code

In this talk, I focus on step-3 and step-4 leveraging shuffle network and 2D SIMD datapath!

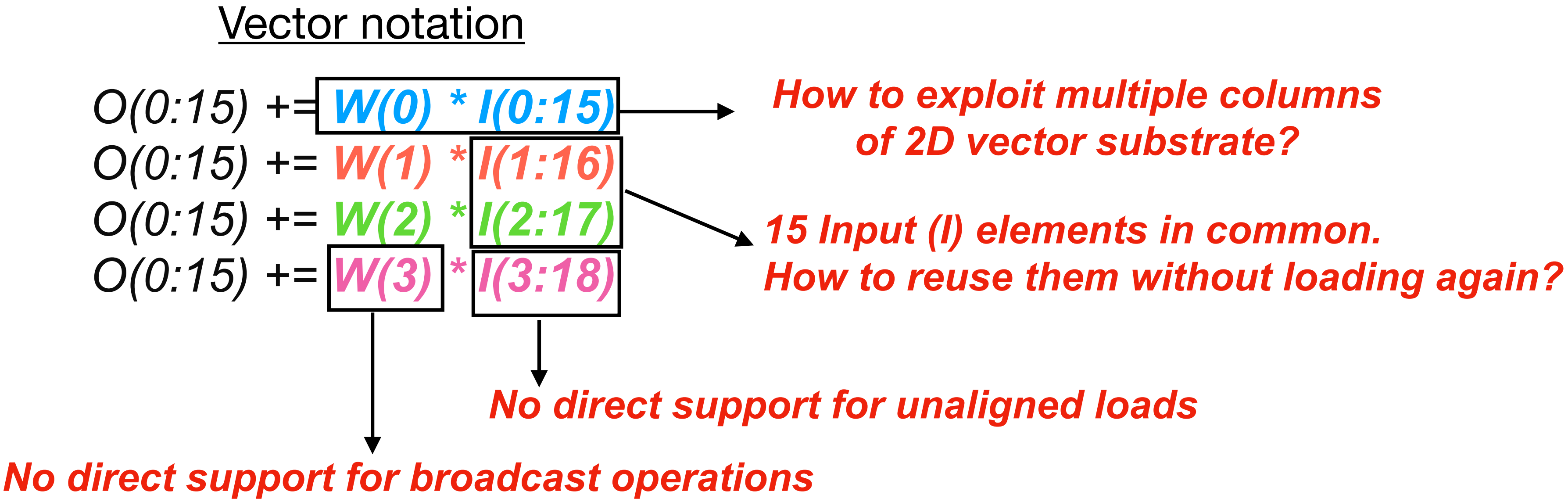
Running Example — CONV1D

HALIDE CODE: O(x) += W(w) * I(x+w);



```
for(x=0; x < 16; x++)
  for(w=0; w < 4; w++)
    O[x] += I[x+w]*W[w];
```

A sample schedule: Unroll w-loop and Vectorize x-loop (VLEN: 16)



Vector notation

*O(0:15) += W(0) * I(0:15)*

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

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

*O(0:15) += W(3) * I(3:18)*

How to exploit multiple columns of 2D vector substrate?

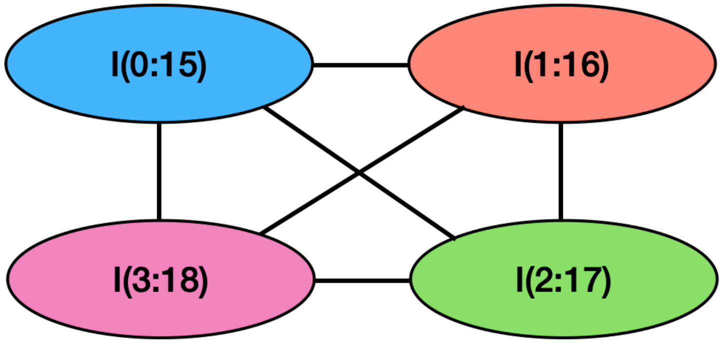
15 Input (I) elements in common.
How to reuse them without loading again?

No direct support for unaligned loads

No direct support for broadcast operations

1) Exploiting Vector Register Reuse

*O(0:15) += W(0) * I(0:15)*
*O(0:15) += W(1) * I(1:16)*
*O(0:15) += W(2) * I(2:17)*
*O(0:15) += W(3) * I(3:18)*



Connected component
V1 — I(0:31)

- Build “*temporal reuse graph*” with nodes being vector loads
  - Edge exists b/w nodes if there is at least one element in common

- <u>AI Engine allows to create logical vector registers of length up to 1024 bits</u>
  - Identify (aligned) connected components and assign each component to a vector register that can subsume the individual vector loads of the component.
  - Use shuffle interconnection network to select desired elements

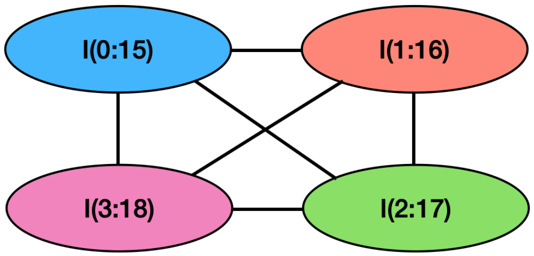
2) Grouping 1D Vector Operations

O(0:15) += W(0) * I(0:15)

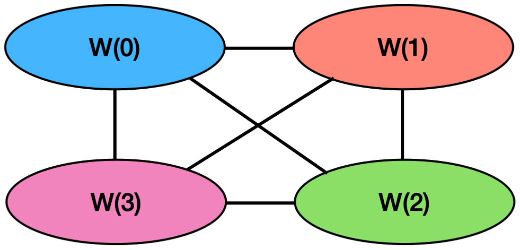
O(0:15) += W(1) * I(1:16)

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

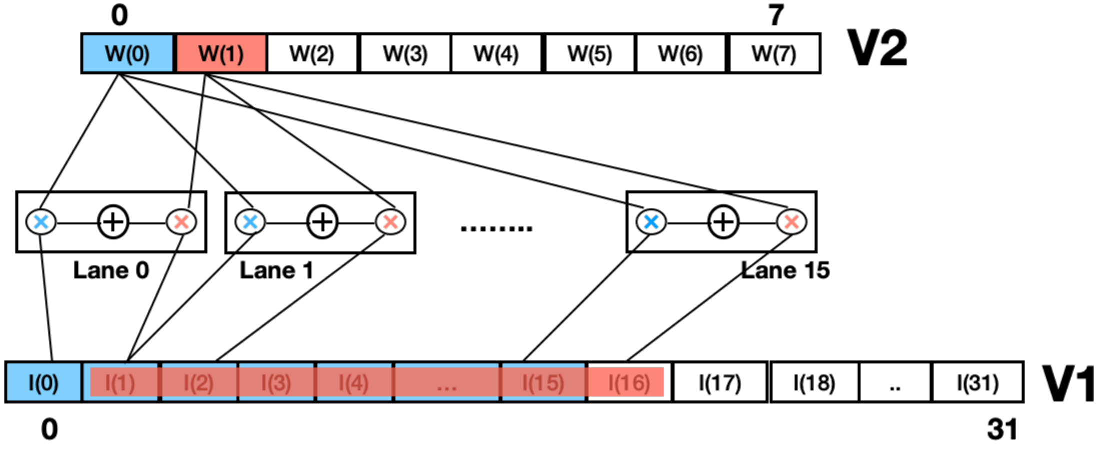
O(0:15) += W(3) * I(3:18)



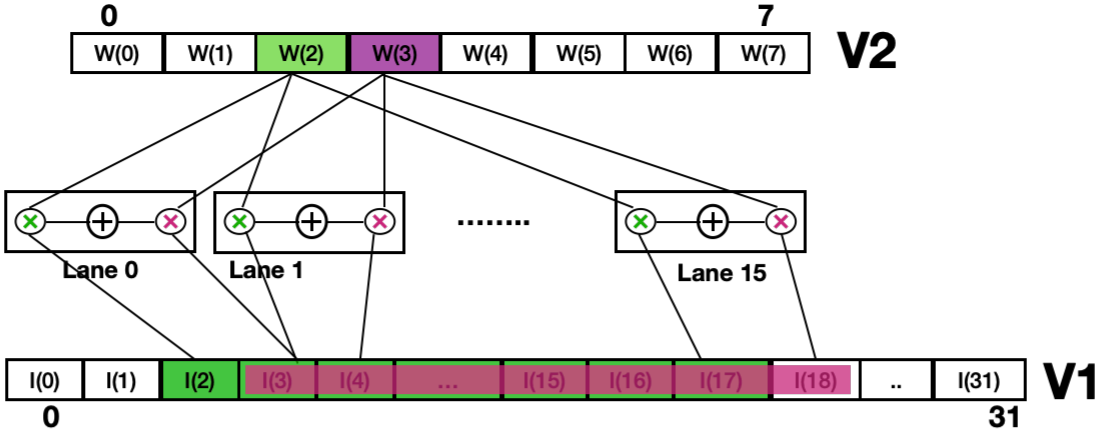
Connected component
V1 — I(0:31)



Connected component
V2 — W(0:7)



a) O(0:15) = W(0) * I(0:15) + W(1) * I(1:16)

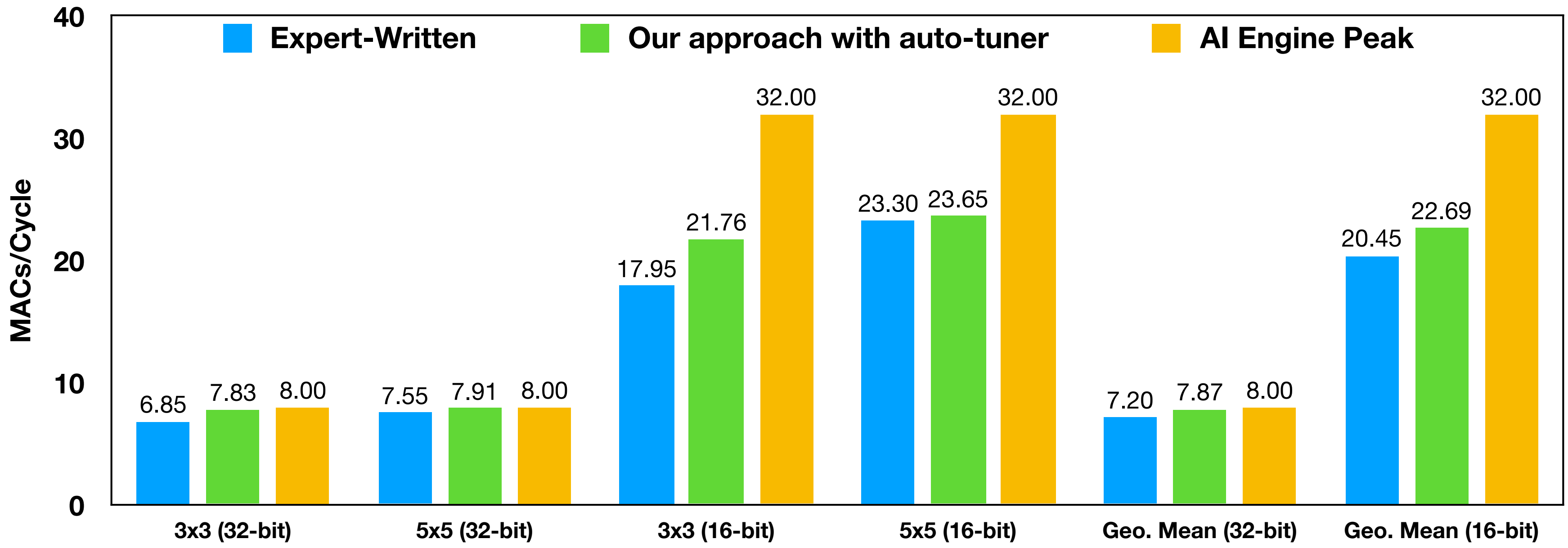


b) O(0:15) += W(2) * I(2:17) + W(3) * I(3:18)

All the 4 operations are performed with a single load of V1 and V2 (maximum reuse)

Evaluation: CONV2D's in CV (256x16)

HALIDE CODE: O(x, y) += W(r, s) * I(x+r, y+s);

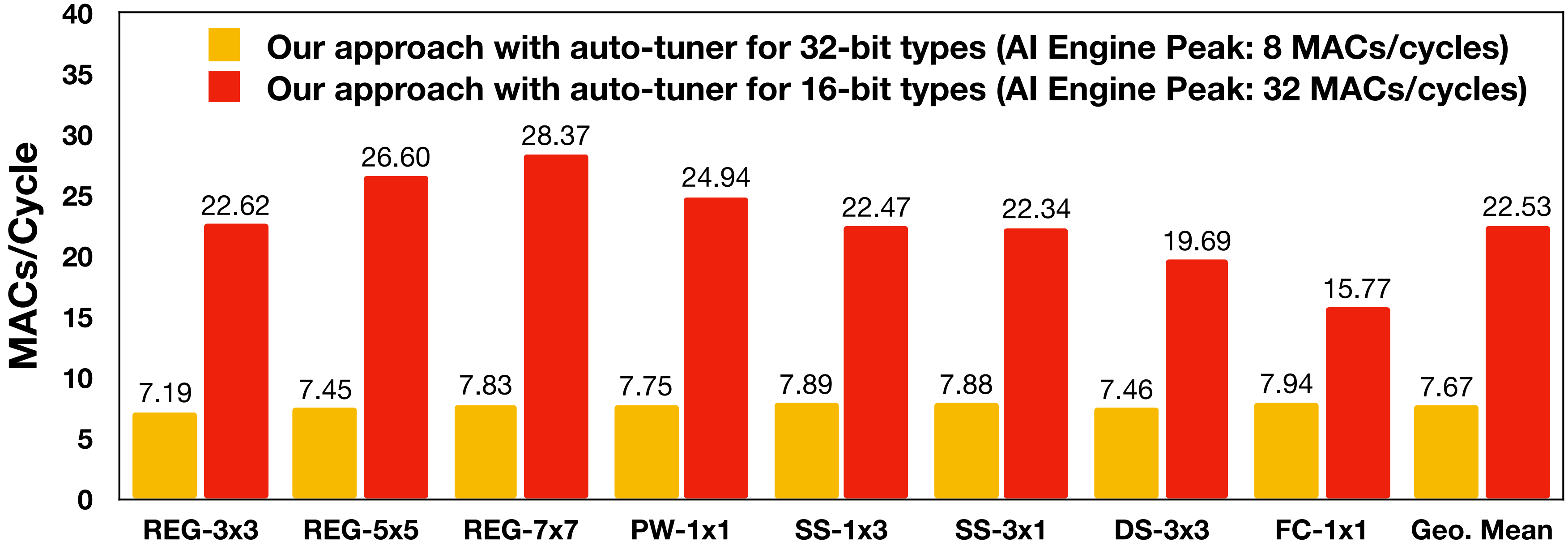


- *Expert-written codes* are available only for 3x3 and 5x5 filters
    - Available as part of the Xilinx's AI Engine compiler infrastructure

- *Auto-tuner was able to find better schedules*
    - Especially non-trivial unroll and jam factors

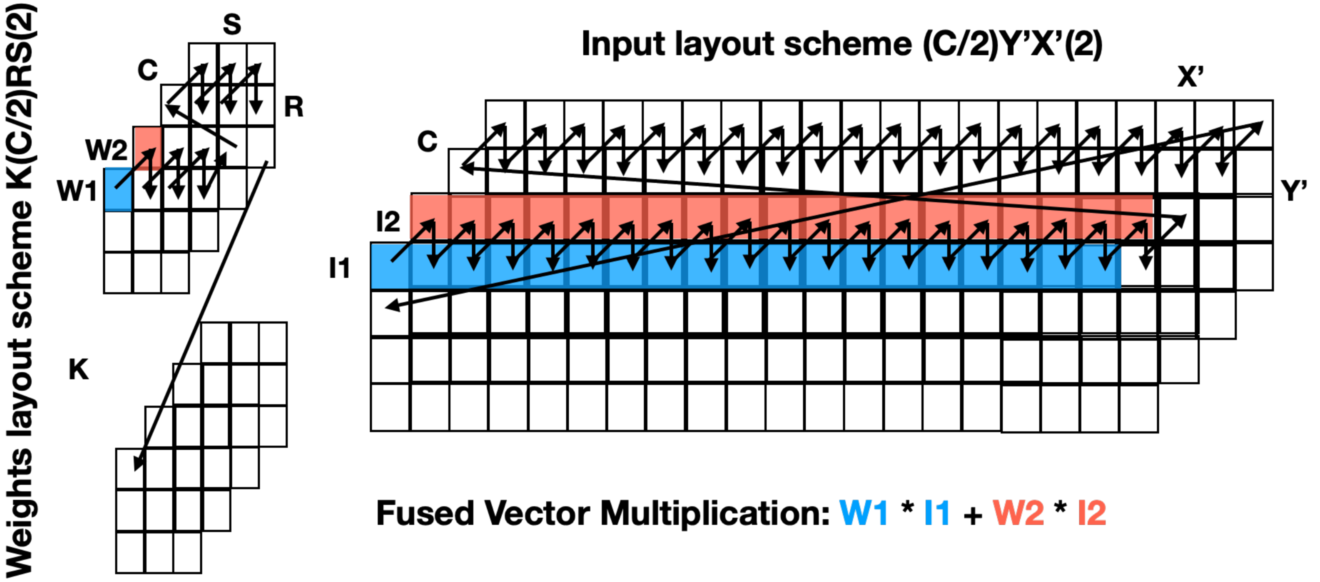
Evaluation: CONV2D's in CNN's (128x2x16)

<u>HALIDE CODE for REG CONV2D:</u> ***O(x, y, k, n) += W(r, s, c, k) * I(x+r, y+s, c, n);***



- *REG-CONV2D (3x3, 5x5, 7x7)*
  - Vectorization along Output width and Reduction along Filter channels
- *PW-CONV2D (1x1), SS-CONV2D (1x3, 3x1), FC-CONV2D (1x1)*
  - Vectorization along Output channels and Reduction along Filter channels
- *DS-CONV2D (3x3) — Padded each row*
  - Vectorization along Output width and Reduction along Filter width

Non-trivial data-layout choices


Weights layout scheme K(C/2)RS(2)
S
C
R
W2
W1
K
Input layout scheme (C/2)Y'X'(2)
X'
C
Y'
I2
I1
Fused Vector Multiplication: W1 * I1 + W2 * I2

- 16-bit *REG-CONV2D (3x3)*
  - Vectorization along Output width and Reduction along Filter channels
  - For the fused vector operation (W1xI1 + W2 x I2)
    - Data for (I1, I2) should be in a single vector register for the operation
    - I1(0) and I2(0) should be adjacent for shuffle network constraints
  - (C/2)Y'X'(2) refers to first laying out an input block of two channels followed by width, height, and remaining channels.

Summary and Related Work

- ***Summary***
  - Manually writing vector code for high-performant tensor convolutions achieving peak performance is extremely challenging!
  - **Domain-specific compilation can be the key!**
    - Proposed a convolution-specific IR for easier analysis and transformations
    - Our approach (Vyasa) can work for any convolution variant regardless of its variations and shapes/sizes.
    - Achieved close to the peak performance for a variety of tensor convolutions

- ***Related work***
  - 2D SIMD data paths and shuffle networks are unique to the AI Engine
    - AFWK, vector unit of PEPSC architecture is the only closely related work
      - A greedy approach in their compiler to identify fusible operations