FlexACC: A Programmable Accelerator with Application-Specific ISA for Flexible Deep Neural Network Inference

En-Yu (Daniel) Yang, Tianyu Jia, David Brooks, Gu-Yeon Wei

Harvard University, Cambridge, MA

ASIP University Day
Wed, Nov 17, 2021
Virtual

This work has also been accepted and presented at 2021 IEEE 32nd International Conference on Application-specific Systems, Architectures and Processors (ASAP)

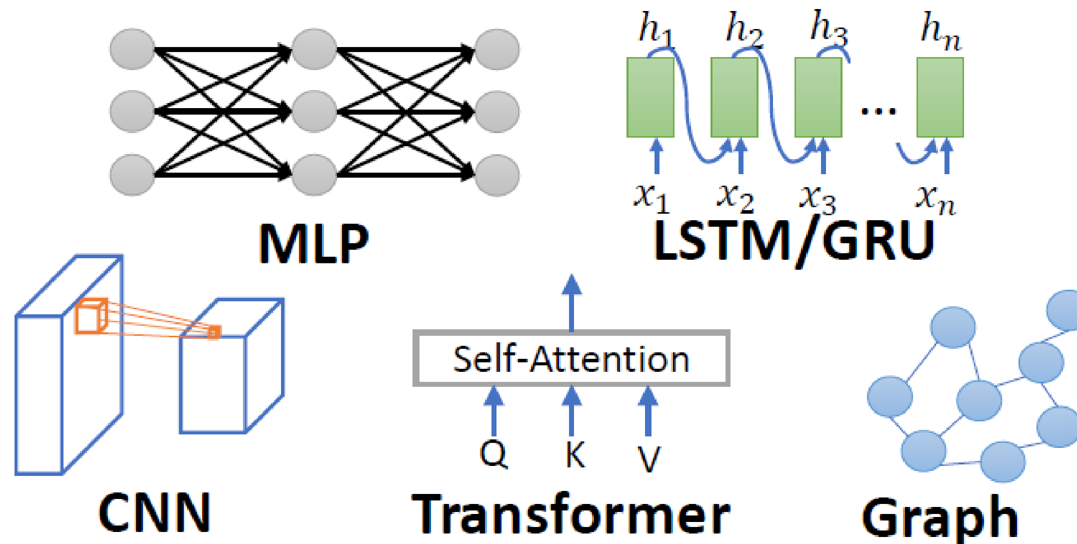
Outline

- **Introduction and Motivation**
- Proposed FlexACC Design
- Software Mapping
- Experimental Results

Introduction

- DNN has become important in many application domains like image classification, speech recognition, natural language processing, etc.

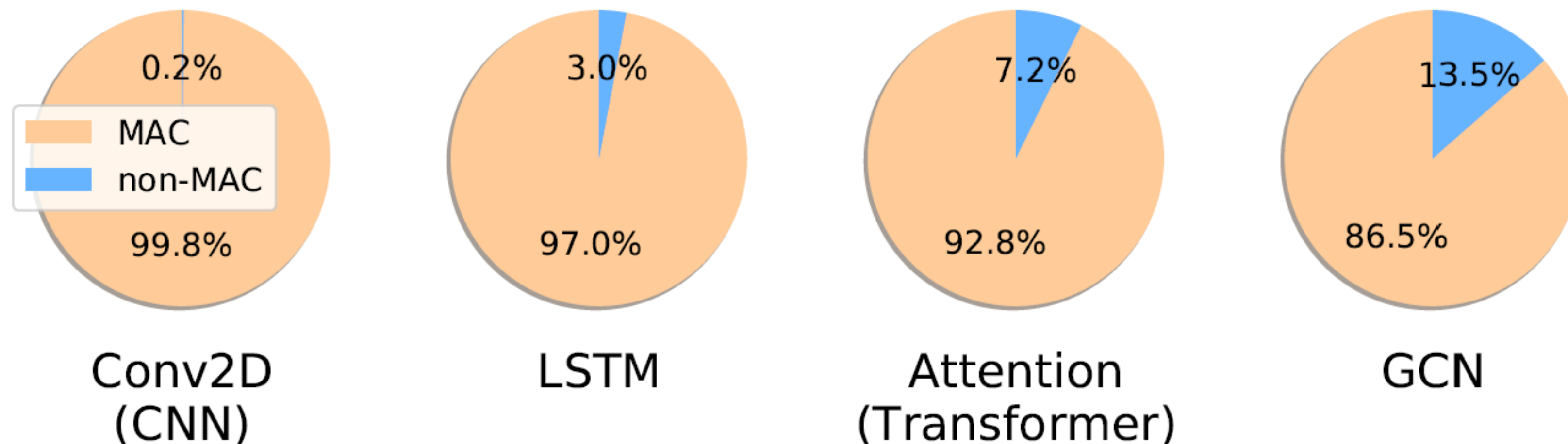
- Diverse types of DNN models are proposed to solve different tasks



DNN Workload Analysis

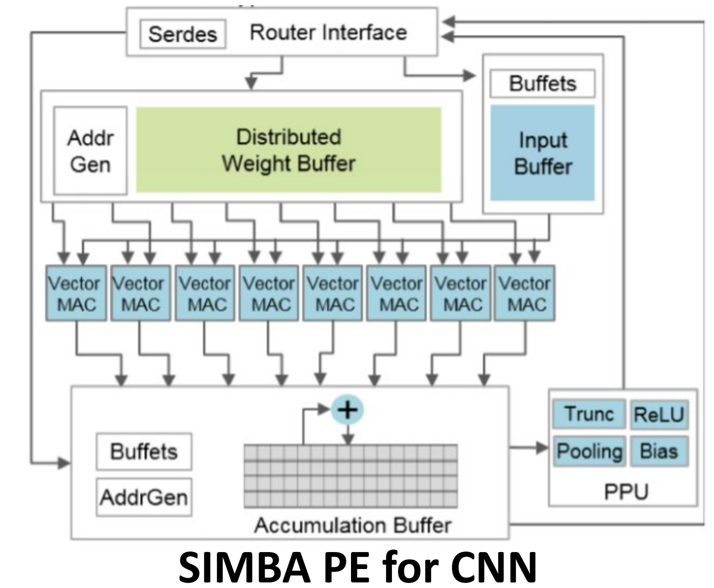
- Different DNN models have different percentages of MAC and non-MAC operations

- Non-MAC operations are critical in overall performance, and computation patterns differ significantly in different DNN types
  - LSTM: Sigmoid/Tanh, vector operations
  - Attention: Softmax

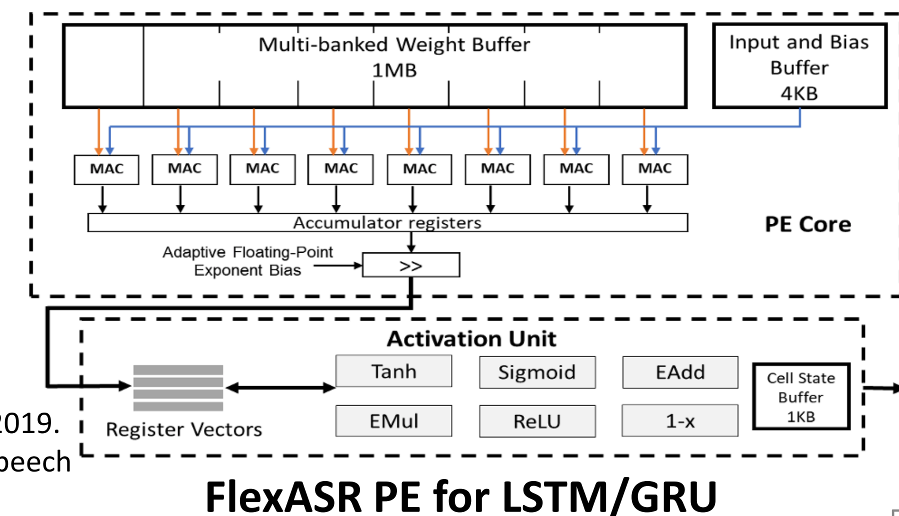


Limitations in Prior Works

- DNNs are changing rapidly, but accelerators are customized for a small range of models
- Limited programmability (or flexibility) makes existing hardware hard to adapt to the rapid evolution of software



SIMBA PE for CNN



FlexASR PE for LSTM/GRU

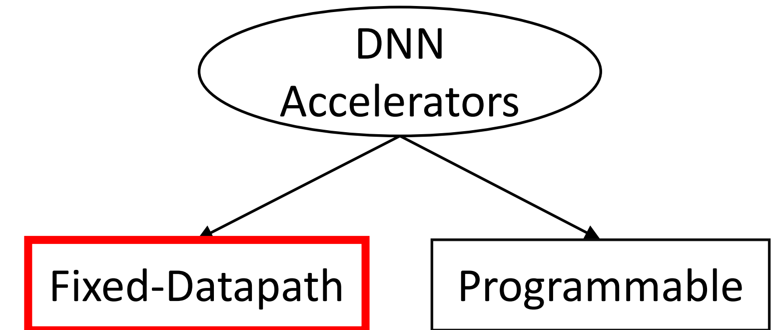
Shao, Yakun Sophia, et al. "Simba: Scaling deep-learning inference with multi-chip-module-based architecture." MICRO 2019.
Tambe, Thierry, et al. "9.8 A 25mm 2 SoC for IoT Devices with 18ms Noise-Robust Speech-to-Text Latency via Bayesian Speech Denoising and Attention-Based Sequence-to-Sequence DNN Speech Recognition in 16nm FinFET." ISSCC 2021.

Supporting Wide Range of DNNs

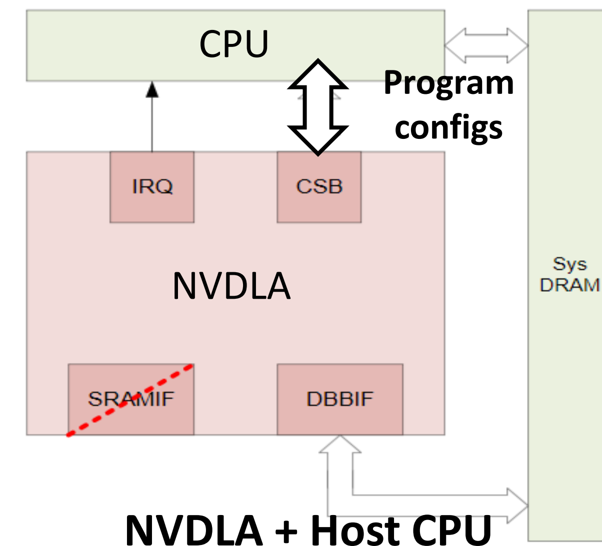
- How to make DNN accelerator design more programmable (or flexible)?
- What is the cost (e.g., hardware area/energy) of supporting more DNN models in a design?

Accelerator Design Choices (1)

- Fixed-Datapath
  - Controlled by a set of configuration registers with a host CPU
  - Higher customization but lower flexibility
  - Examples: NVDLA, SIMBA, FlexASR, etc.



Tremendous engineering effort is often needed to design for a wide range of DNNs

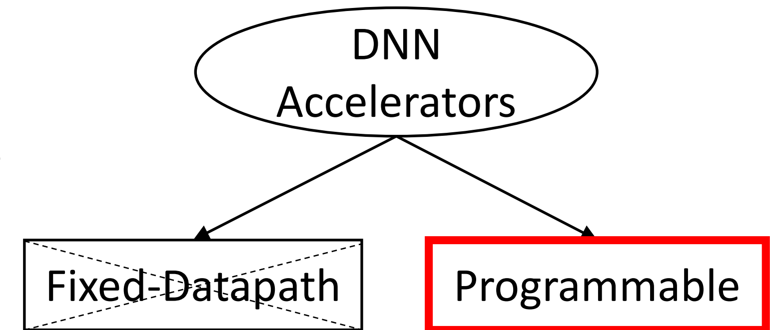


NVDLA + Host CPU

NVDLA, http://nvdla.org/

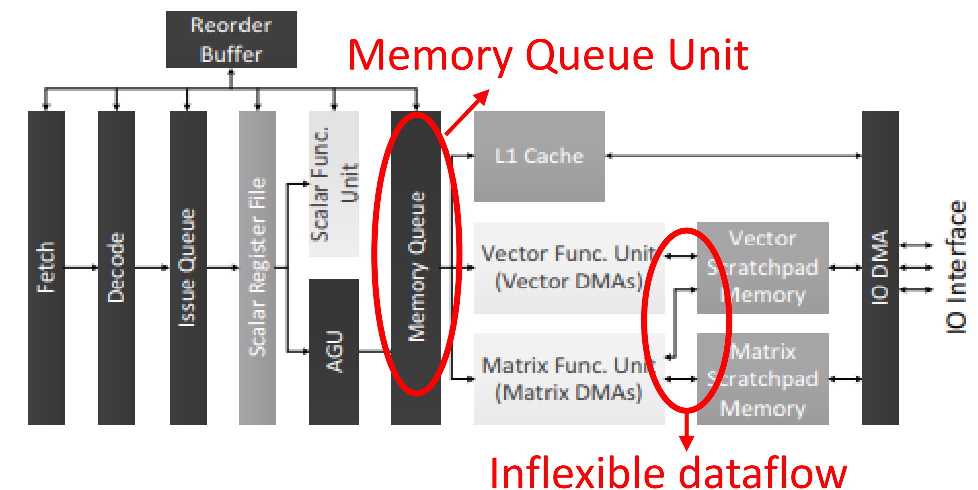
Accelerator Design Choices (2)

- Programmable Datapath
  - General-purpose CPU with customized hardware units
  - Fine-grained controller/instruction set
  - Examples: Cambricon, etc.



- Weakness in Cambricon
  - Overhead on scheduling of multicycle instructions

Proposed FlexACC: programmable datapath with fine-grained "single cycle" instructions for different software patterns



Cambricon Accelerator

Liu, Shaoli, et al. "Cambricon: An instruction set architecture for neural networks." ISCA 2016.

Key Contributions of This Work

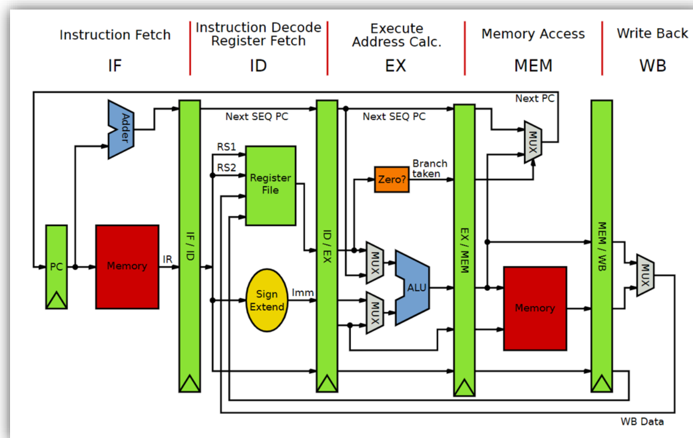
- A comprehensive workload analysis is conducted on a diverse set of DNN models (i.e., CNN, LSTM, Transformer, and GCN)
- We design FlexACC with tightly coupled RISC-V and customized DNN acceleration instructions to support different DNN workloads
- We quantitatively compare FlexACC with fixed-datapath baselines to study the cost of programmability

Outline

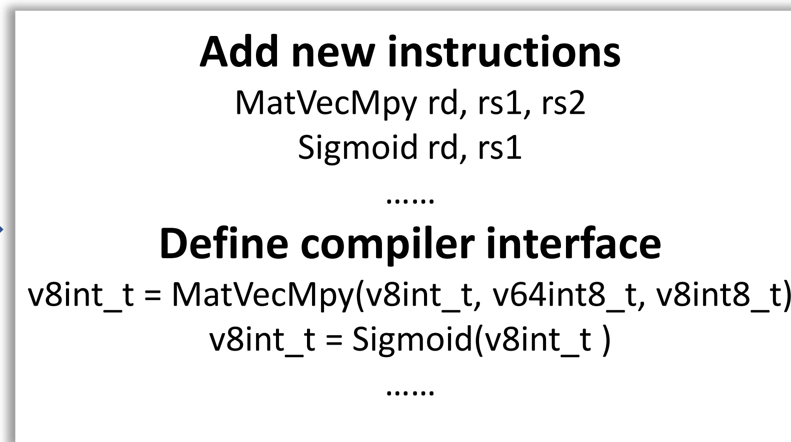
- Introduction and Motivation
- **Proposed FlexACC Design**
- Software Mapping
- Experimental Results

ASIP Designer: A Brief Overview

- Starting from a general-purpose RISC ISA template
- Extend it by adding new instructions and customized hardware units
- Codesign both hardware and compiler interface simultaneously
- Test/optimize hardware by running compiled C code on provided Instruction Set Simulator



RISC baseline hardware



Codesign of hardware/compiler

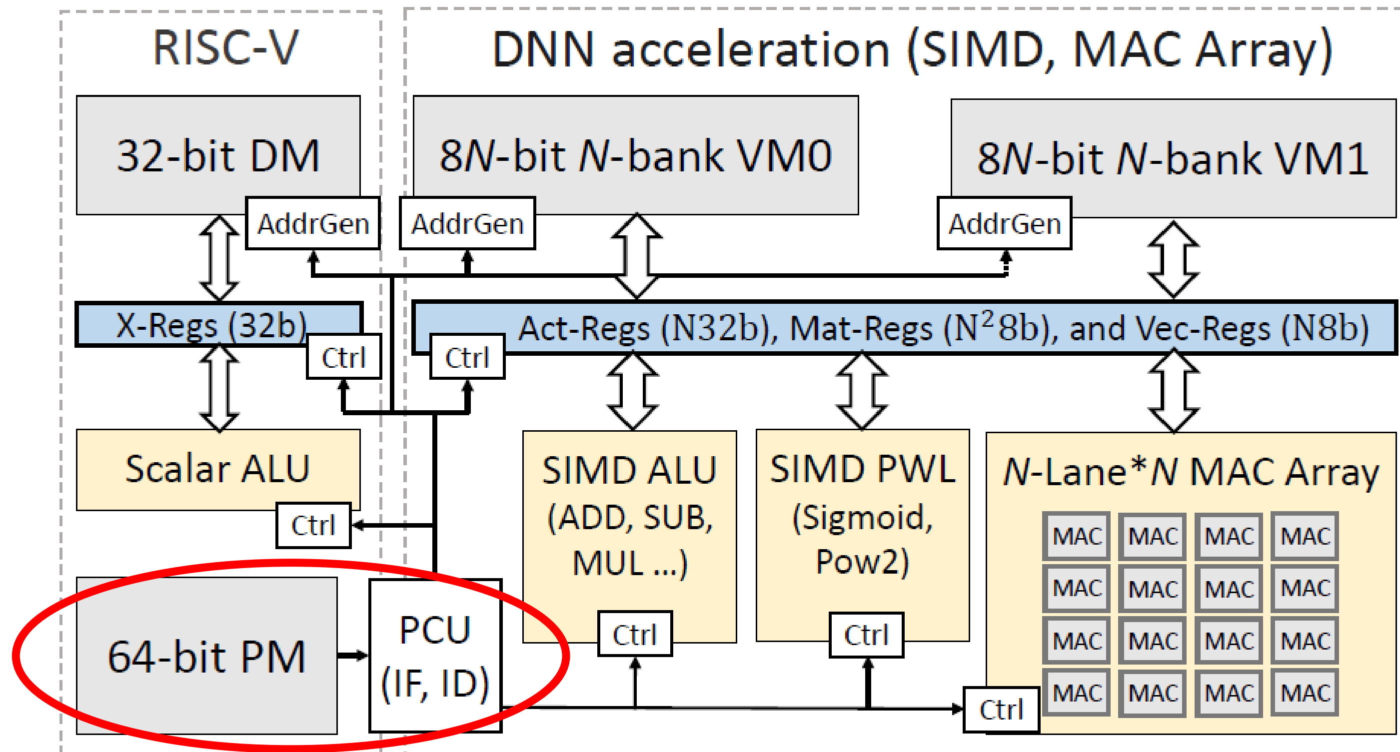
```
void sort(int A[], int len)
{
    for (int i = 0 ; i < len-1; i++) {
    int loc = find min location(A,i,len);
    int temp = A[i];
    A[i] = A[loc];
    A[loc] = temp;
    }
}
```

```
ISS command       = <PROCDIR>/../iss/tnlp ca dbg
ISS mode          = Cycle accurate
Core name         = ::iss

Cycle count       = 1401
Instruction count = 1199

PC =          11

SP =        2048
Stack area: DMb [       512 ..       2048] growing down
Minimum stack pointer value =        1960
```

Simulate with compiled C code

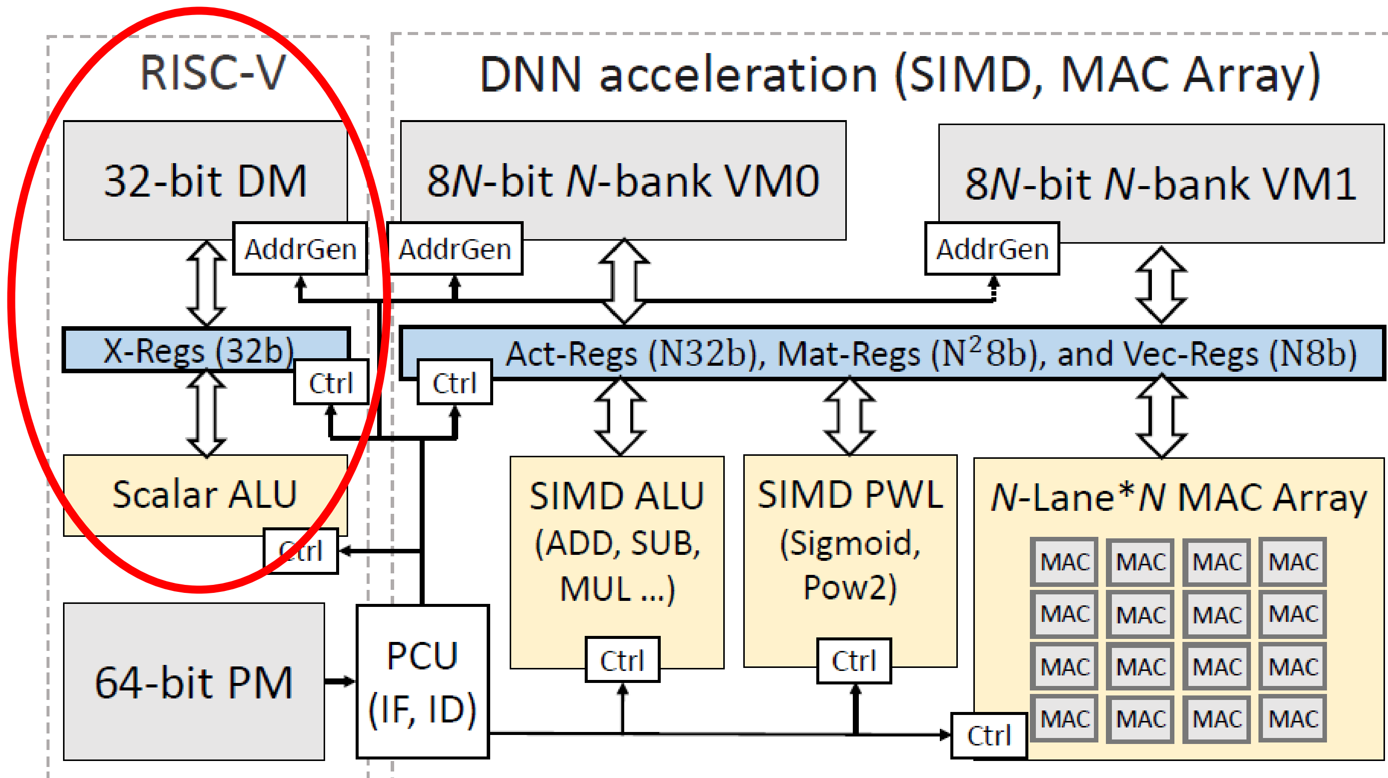
FlexACC Architecture (1)

- FlexACC architecture combines RISC-V pipeline and DNN acceleration units
- Program control unit (PCU) fetches instructions from program memory (PM) to issue control signals



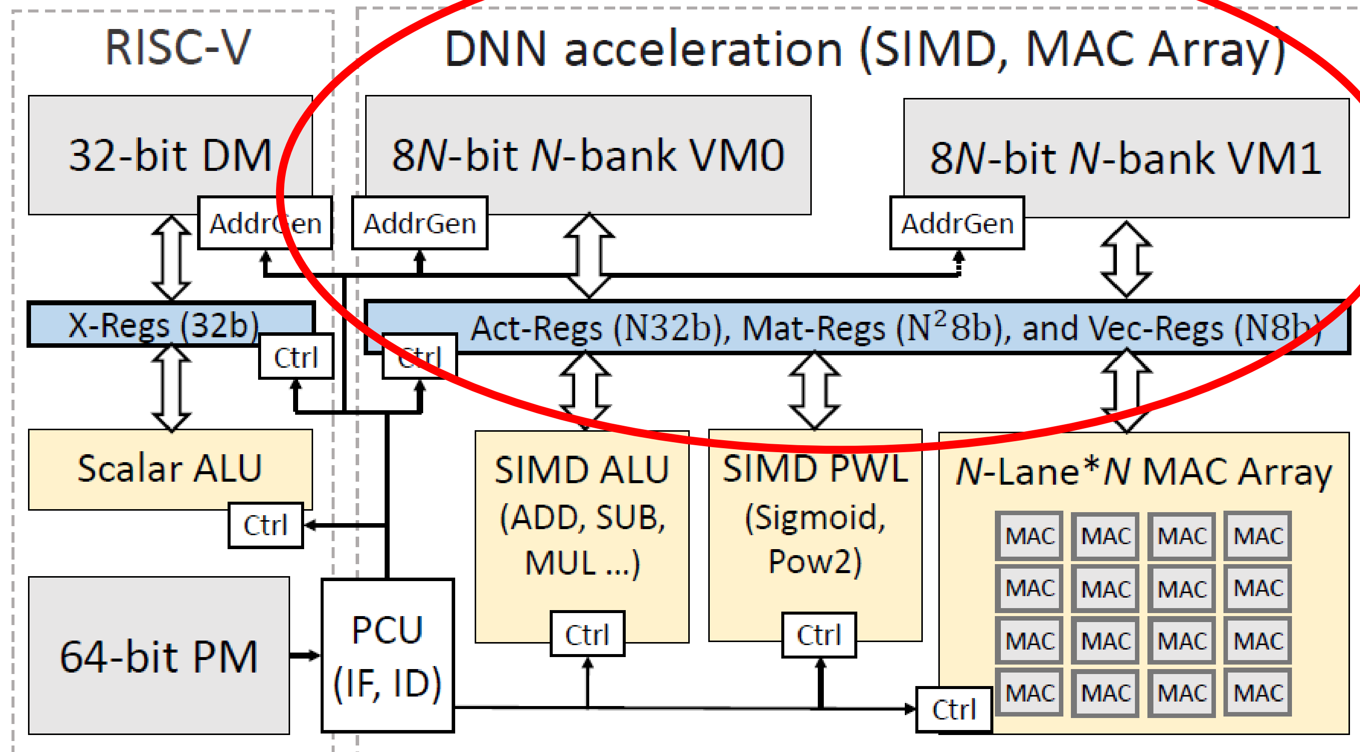
FlexACC Architecture (2)

- Scalar operations are performed on RISC-V pipeline with data memory (DM), general purpose registers (X-Regs), and scalar ALU



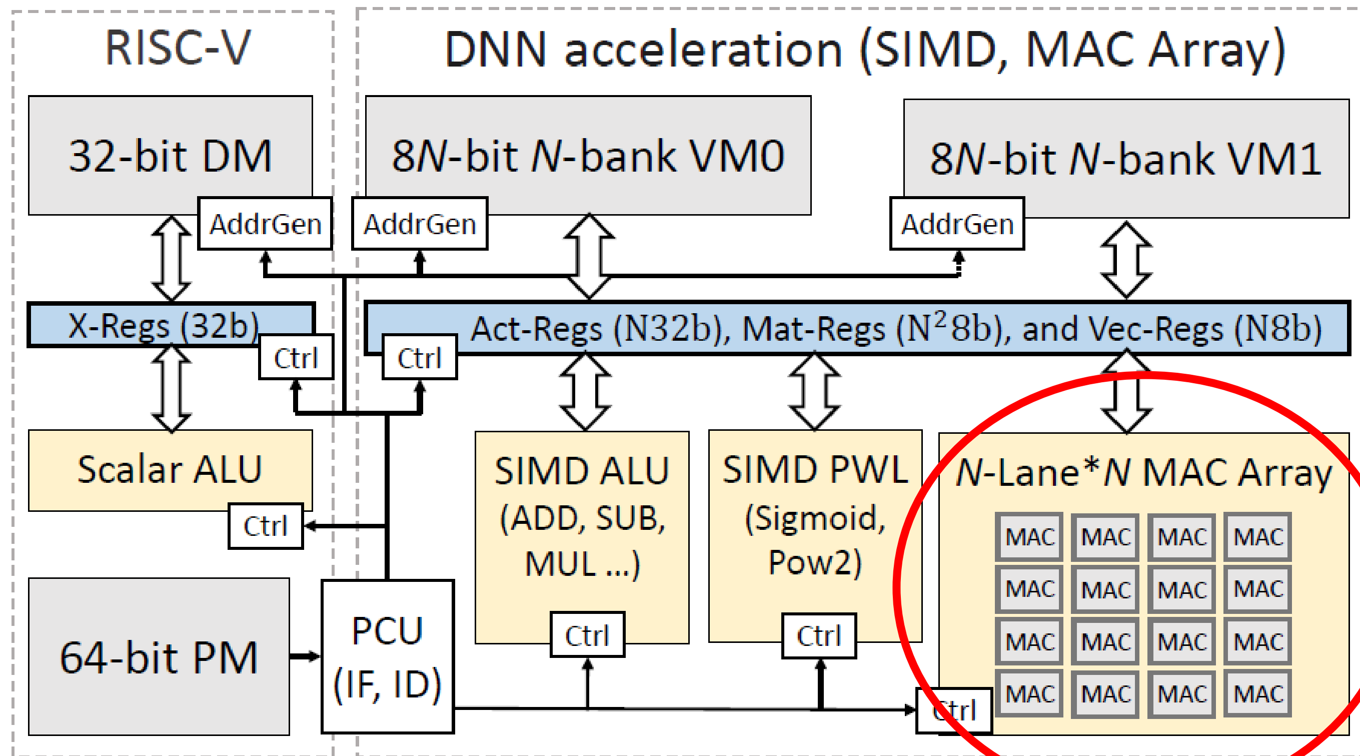
FlexACC Architecture (3)

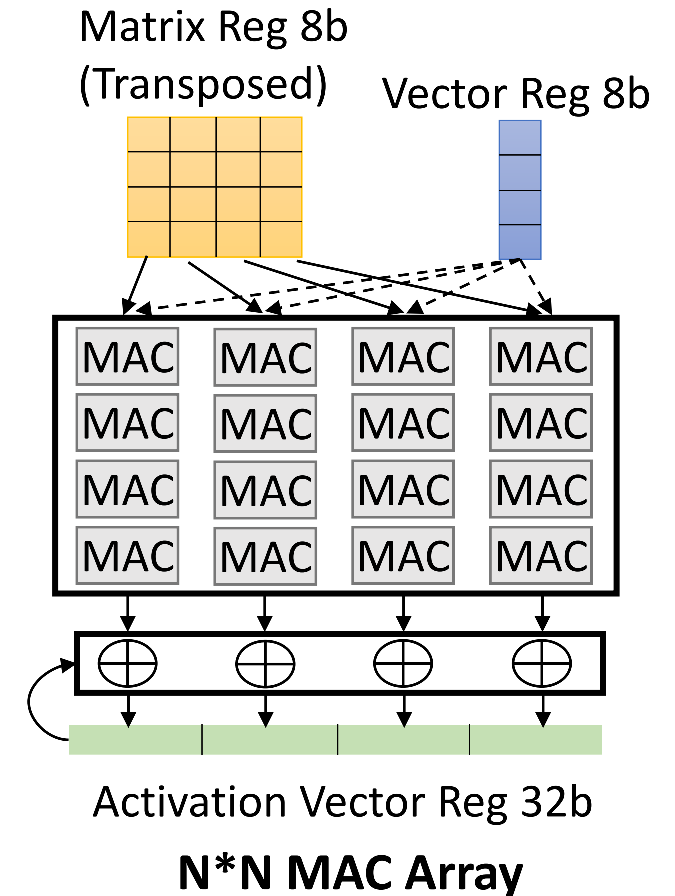
- DNN acceleration datapath includes customized vector memories (VM0, VM1) and different types of vector or matrix registers
- Note: vector size *N* is a configurable architectural parameter



FlexACC Architecture (4)

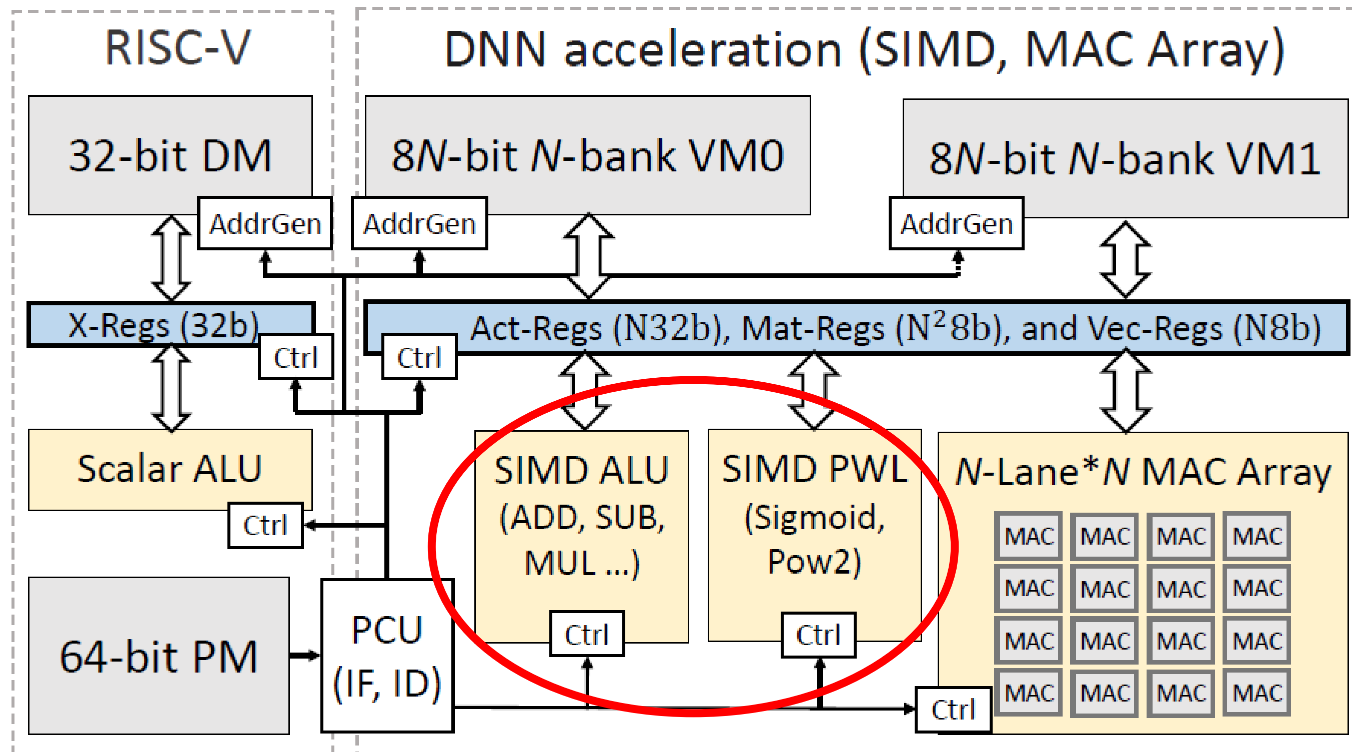
- MAC Array computes matrix-vector multiply with 8b multiplication and 32b addition
- Reuse of vector register via broadcasting

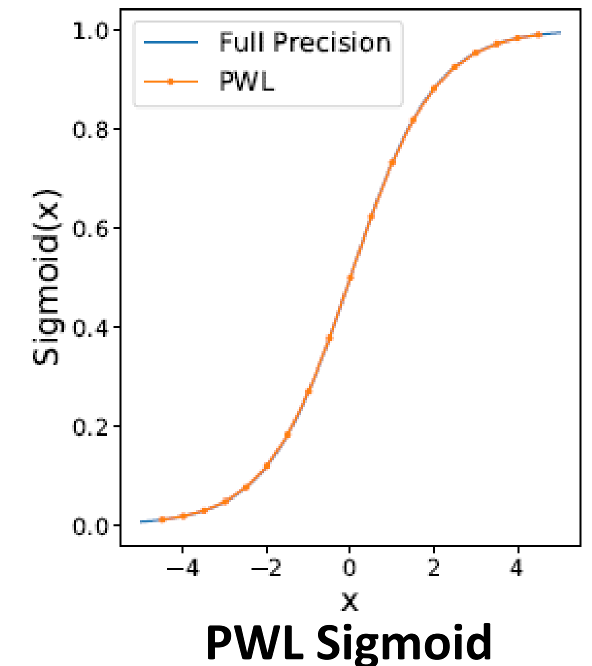




FlexACC Architecture (5)

- Scalar arithmetic is carried out in each lane of SIMD
- SIMD also includes piecewise linear functions (PWL) with lookup tables





PWL Sigmoid

Application-Specific Instruction Set (1)

- FlexACC ISA is a 64b VLIW with four instruction slots

RISC-V Slot (63–32)	
ALU, DIV	0110011
ALU immd.	0010011
Load	0000011
Store	0100011
Branch	1100011
Jump	1101111
Jump & link register	1100111
Load upper immd.	0110111
RISC-V Extensions	
Load w/ pm	0001011
Store w/ pm	0101011
Zero overhead loops Boolean (min/max)	1111011

Vector Slot (31–20)
Vector Move
X ⇔ Act
Vec ⇔ Vec
Mat ⇔ Mat
Act ⇔ Act
Act ⇔ Vec
……
MAC Instructions
MatVecMulAdd
MatVecMul
SIMD Instructions
Vector PWL
Vector ALU

VM 1 (19–10)	VM 0 (09–00)
Load Vec	Load Vec
Load Mat	Load Mat
Load Act	Load Act
Store Vec	Store Vec
Store Mat	Store Mat
Store Act	Store Act

Application-Specific Instruction Set (2)

- RISC-V Slot includes baseline 32b instructions with some extensions

63 – 32		31 – 20	19 – 10	09 – 00
RISC-V Slot		**Vector Slot**	**VM 1**	**VM 0**
ALU, DIV	0110011	Vector Move	Load Vec	Load Vec
ALU immd.	0010011	X ⇔ Act	Load Mat	Load Mat
Load	0000011	Vec ⇔ Vec	Load Act	Load Act
Store	0100011	Mat ⇔ Mat	Store Vec	Store Vec
Branch	1100011	Act ⇔ Act	Store Mat	Store Mat
Jump	1101111	Act ⇔ Vec	Store Act	Store Act
Jump & link register	1100111	......		
Load upper immd.	0110111	MAC Instructions		
RISC-V Extensions		MatVecMulAdd		
Load w/ pm	0001011	MatVecMul		
Store w/ pm	0101011	SIMD Instructions		
Zero overhead loops	1111011	Vector PWL		
Boolean (min/max)		Vector ALU		

Application-Specific Instruction Set (3)

- Vector Slot
  - Vector move instructions
  - MAC instructions
  - SIMD instructions

63 … 32		31 … 20	19 … 10	09 … 00
RISC-V Slot		**Vector Slot**	**VM 1**	**VM 0**
ALU, DIV	0110011	Vector Move	Load Vec	Load Vec
ALU immd.	0010011	X ⇔ Act	Load Mat	Load Mat
Load	0000011	Vec ⇔ Vec	Load Act	Load Act
Store	0100011	Mat ⇔ Mat	Store Vec	Store Vec
Branch	1100011	Act ⇔ Act	Store Mat	Store Mat
Jump	1101111	Act ⇔ Vec	Store Act	Store Act
Jump & link register	1100111	……		
Load upper immd.	0110111	MAC Instructions		
RISC-V Extensions		MatVecMulAdd		
Load w/ pm	0001011	MatVecMul		
Store w/ pm	0101011	SIMD Instructions		
Zero overhead loops	1111011	Vector PWL		
Boolean (min/max)		Vector ALU		

Application-Specific Instruction Set (4)

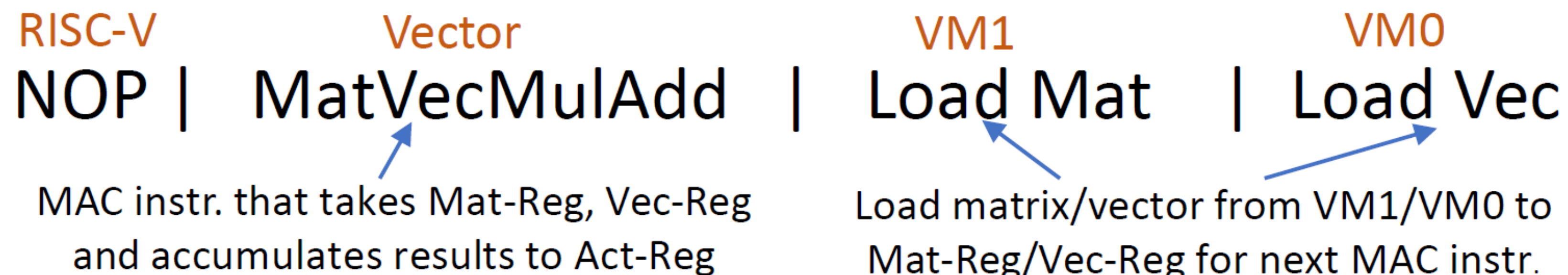
- Vector Memory Slots
  - VM0 and VM1
- Flexible load/store of vector variables from vector memories

63	32	31 – 20	19 – 10	09 – 00
RISC-V Slot		Vector Slot	VM 1	VM 0
ALU, DIV	0110011	Vector Move	Load Vec	Load Vec
ALU immd.	0010011	X ⇔ Act	Load Mat	Load Mat
Load	0000011	Vec ⇔ Vec	Load Act	Load Act
Store	0100011	Mat ⇔ Mat	Store Vec	Store Vec
Branch	1100011	Act ⇔ Act	Store Mat	Store Mat
Jump	1101111	Act ⇔ Vec	Store Act	Store Act
Jump & link register	1100111	......		
Load upper immd.	0110111	MAC Instructions		
RISC-V Extensions		MatVecMulAdd		
Load w/ pm	0001011	MatVecMul		
Store w/ pm	0101011	SIMD Instructions		
Zero overhead loops	1111011	Vector PWL		
Boolean (min/max)		Vector ALU		

Simultaneous Computation & Memory Access

- Several techniques are leveraged to improve the overall performance
  - Instruction level parallelism (ILP)
  - load/store with address postmodify (hardware-based address increment)
  - zero overhead loop (ZLP)

- The combination of ILP, address postmodify, and ZLP ensures continuous dataflow and zero delay during sequential MAC operations.

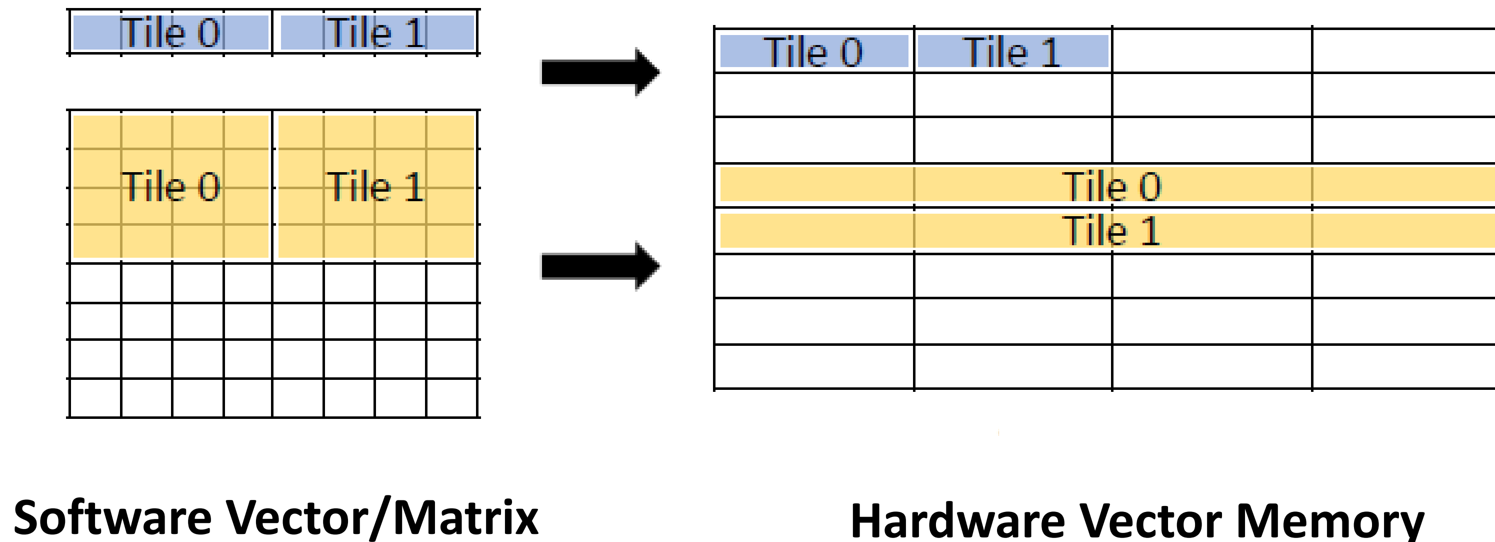


Outline

- Introduction and Motivation
- Proposed FlexACC Design
- **Software Mapping**
- Experimental Results

Tensor Tiling

- Tensor tiling is an essential step to map vectors and matrices to vector memories (VM1 and VM0)
- For example, software vector is tiled along one dimension and matrix is tiled along two dimensions



Computation Mapping

- C code is compiled into FlexACC instructions to utilize customized hardware units

Conv2D (Inner-loops)

```
for (h = h_st; h < h_ed; h++)
  for (w = w_st; w < w_ed; w++)
    for (ic = 0; ic < Cin/N; ic++)
      acc_vec = MatVecMulAdd(……);
```

```
do (loop) x19, 6
do (loop) x23, 3
vmac a0, m0, v0
```

Attention (Softmax)

```
for (j = 0; j < T/N; j++){
  act = pow2(act);
  sum += vsum(act);
}
inv_sum = 65536 / sum
```

```
do (loop) x26, 31
pow2 (exp) a0, a0
vsum (sum) x6, a0
add  x11, x9, x11
div  x11, x5, x11
```

GCN (Aggregation)

```
while (is_end != false) {
  // Aggregate next node
  for (k = 0; k < Cout/N; k++){
    ……………………
  }
}
```

```
do (loop) x24, 33
j (jump) 10
……………………
vadd a1, a0
……………………
bne (branch) x2,x6,-10
```

Sequential and Irregular Memory Access

- Sequential memory access leverages hardware-based address generators to increment addresses by constant offsets

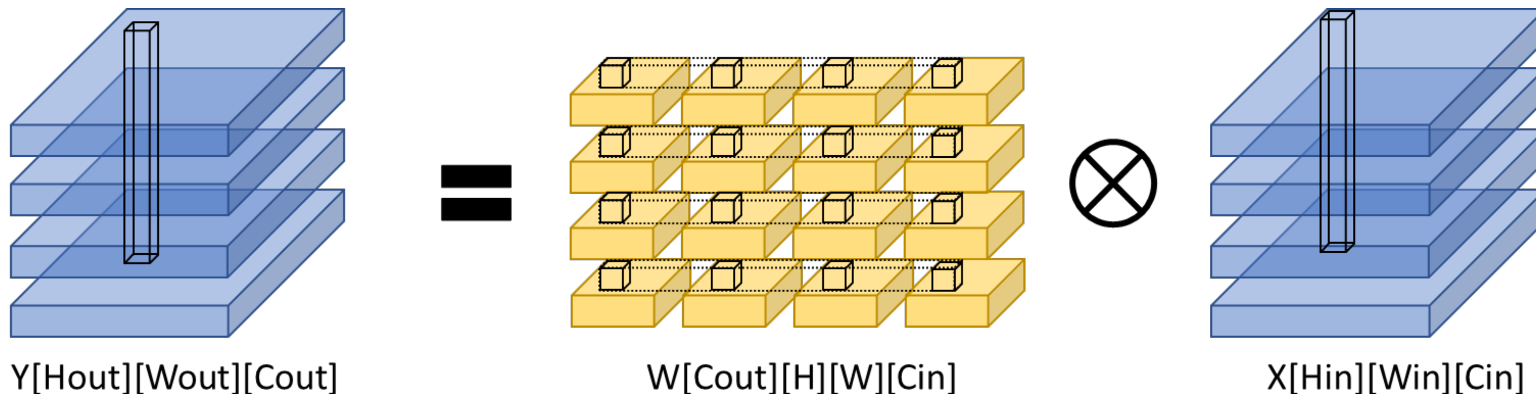
- Irregular access patterns can only be managed in a software-based approach with additional scalar or control instructions

- For efficient computations, loop structures of DNN should be arranged in a way that the memory access of the inner-most loop is sequential

CNN Example

- Conv2D operation involves convolution of
  - Input Image : X[Hin][Win][Cin]
  - Weight filters : W[Cout][H][W][Cin]
  - Output Image : Y[Hout][Wout][Cout]
- 2D tiling on Cin and Cout => MatVecMpyAdd
- Inner Loop:
  Hardware-based address increment is leveraged
- Outer Loops:
  Address is computed by software-based approach

```
1  // Tiled Weight Tensor: W[Cout/N][H][W][Cin/N][N][N]
2  // Tiled Input  Tensor: x[Hin ][Win ][Cin/N][N]
3  // Tiled Output Tensor: y[Hout][Wout][Cin/N][N]
4  for (ho = 0; ho < Hout; ho++) {
5    for (wo = 0; wo < Wout; wo++) {
6      for (co = 0; co < Cout/N; co++) {       // tiled Cout
7        acc_vec  = 0;
8        // CPU compute base address of output
9        //  and start/end input (hi, wi) and weight (h, w)
10       h_st = max(PAD - ho, 0);   w_st = max(PAD - wo, 0);
11       hi_st = -PAD + ho*STRIDE;  wi_st = -PAD + wo*STRIDE;
12       h_ed = min(Hin-hi_st, H);  w_ed = min(Win-wi_st, W);
13
14       for (h = h_st; h < h_ed; h++) {  weight height
15         for (w = w_st; w < w_ed; w++) {  weight width
16           // CPU compute base address of input/weight
17           hi = hi_st + h; wi = wi_st + w;
18           for (ci = 0; ci < Cin/N; ci++) { // tiled Cin
19             acc_vec = MatVecMpyAdd (acc_vec,
20                       W[co][h][w][ci], x[hi][wi][ci]);
21           }
22         }          Matrix-Vector Multiply and Add
23       }
24       y[ho][wo][co] =  acc_vec;
25     }
26   }
27 }
```



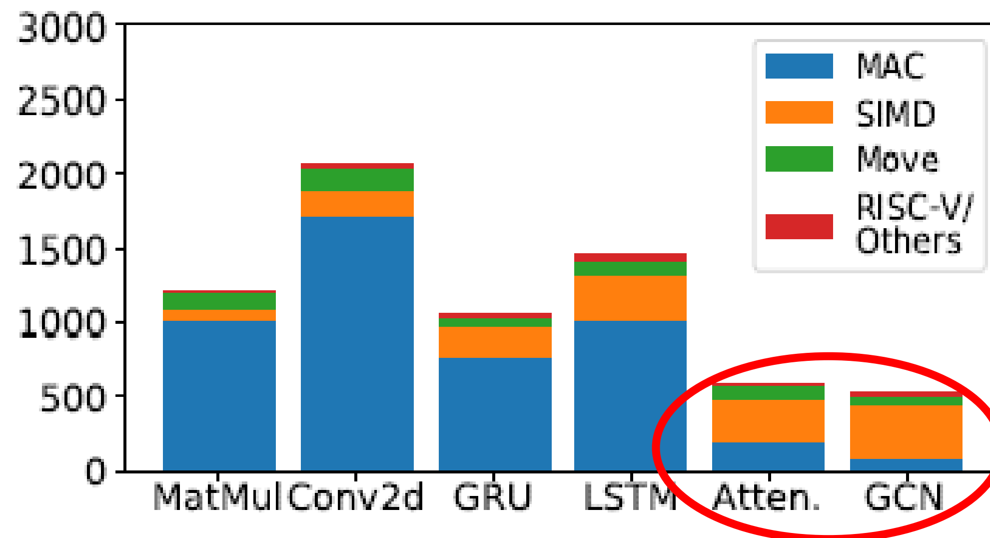
Outline

- Introduction and Motivation
- Proposed FlexACC Design
- Software Mapping
- **Experimental Results**

FlexACC Performance

- We simulate FlexACC (vector size N=8) on 6 selected DNN workloads

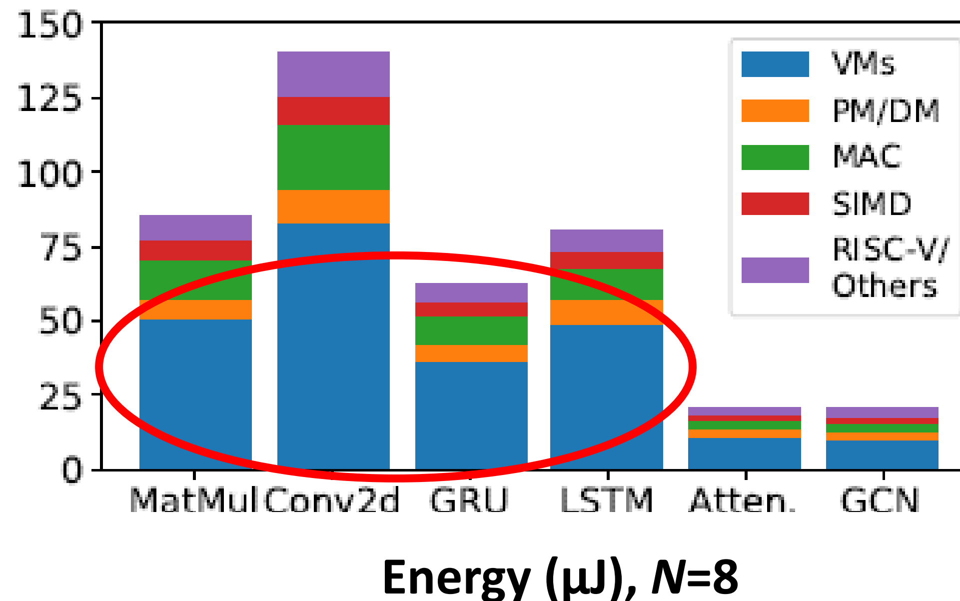
- Decent MAC utilization is achieved on Conv2D (84%) and LSTM (71%)
- Attention and GCN involves more non-MAC operations, resulting in SIMD bottlenecks



Performance (μs), N=8

FlexACC Energy

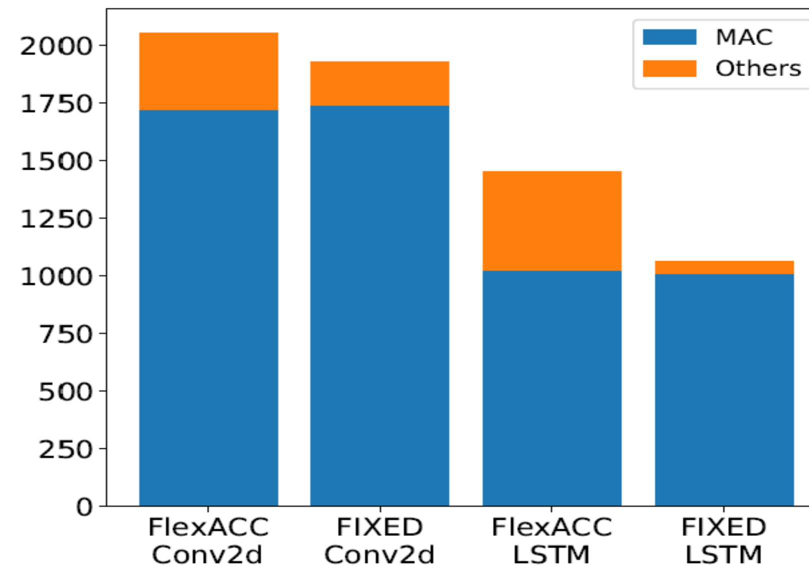
- Since output stationary dataflow is used, energy is dominated by load/store data from vector memories
- We further explore reducing memory access with weight stationary flow



Energy (μJ), *N*=8

Comparisons with Fixed-Datapath Designs (1)

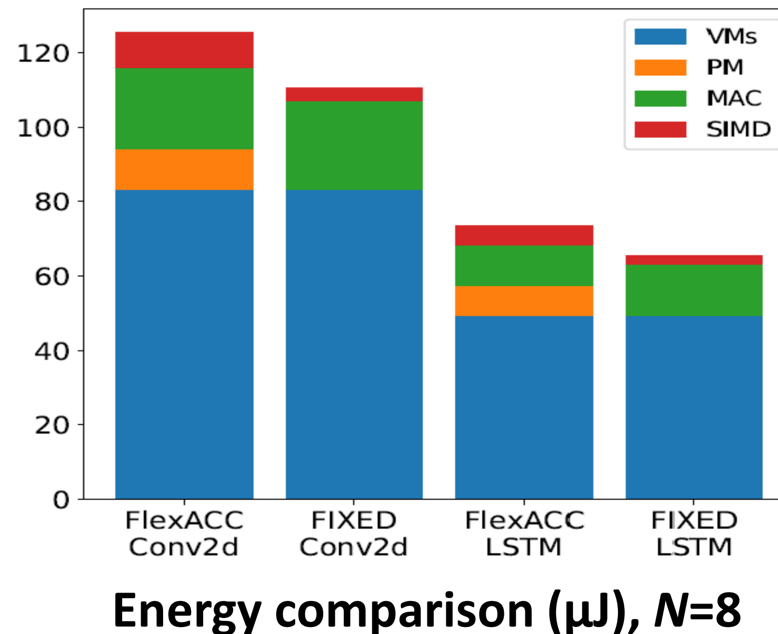
- FlexACC is compared with two standalone fixed-datapath (hardwired-datapath or ASIC) Conv2D and LSTM engines
- Performance comparisons
  - 10% or 30% latency increase than FIXED-Conv2D or FIXED-LSTM



Performance comparison (μs), *N*=8

Comparisons with Fixed-Datapath Designs (2)

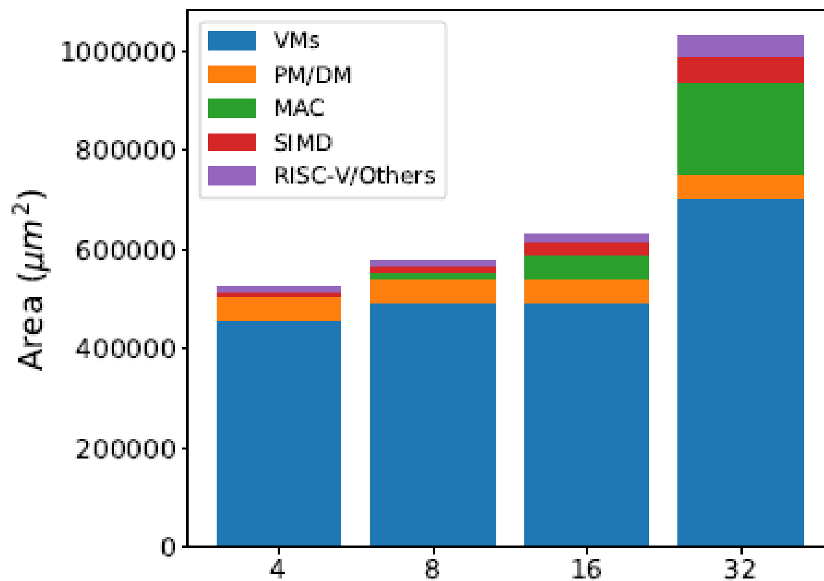
- Energy comparisons
  - 15% or 11% energy increase than FIXED-Conv2D or FIXED-LSTM
  - The energy gap is related to instruction fetch from program memory



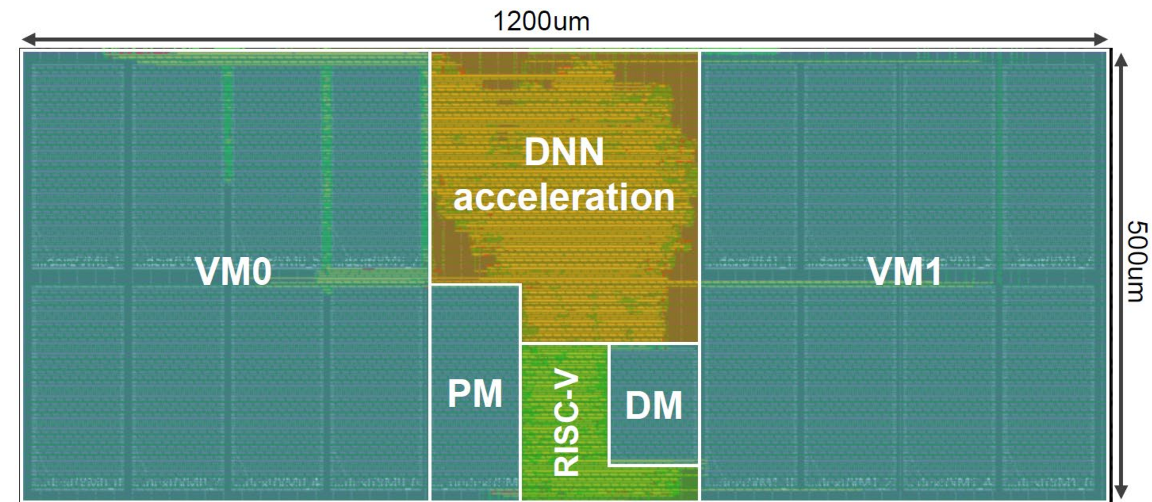
Energy comparison (μJ), *N*=8

FlexACC Design Space Explorations

- Vector size of FlexACC is configurable for *N* = 4, 8, 16 or 32
- More studies are provided to discuss how hardware performance can be affected by different design choices



Area vs Vector Size *N*

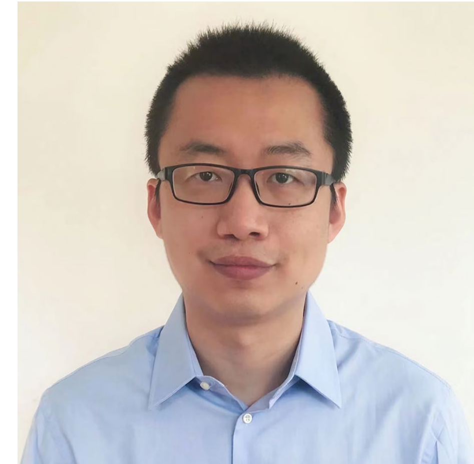


Layout of FlexACC with *N*=8

Conclusion

- We propose and implement FlexACC accelerator with an application-specific ISA for DNN inferences
- Experimental results affirm FlexACC can perform a wide range of DNN inferences with decent performance
- A head-to-head comparison to fixed-datapath baselines further reveals that FlexACC has moderate overhead of achieving high programmability

Acknowledgement



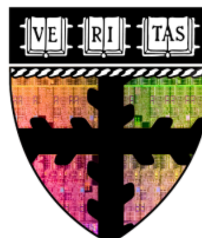
Prof. Gu-Yeon Wei
Harvard University

Prof. David Brooks
Harvard University

*Prof. Tianyu Jia
Carnegie Mellon University

- I would like to thank my advisors and colleagues for their guidance and support in this project
- I would like to thank David Florez (from Synopsys) for his advice in hardware implementation with ASIP Designer

* Prof. Tianyu Jia was a Postdoctoral Fellow at Harvard University during this project



Thank You