AI-RISC - Scalable RISC-V Processor with Tightly Integrated AI Accelerators and Custom Instruction Extensions

Vaubhav Verma and Mircea R. Stan
High-Performance Low-Power Lab (HPLP)
University of Virginia
vv8dn@virginia.edu
Need for Edge AI

• **Latency** – enable real-time AI systems
• **QoS** – cannot rely on connectivity in remote areas
• **Security** – sending data over network not secure
• **Privacy** – keep private data locally on device
• **Bandwidth** – send “information” to cloud rather than “data”
• **Cost** – data communication is costly
But Edge AI is different!

- Smaller neural network models
- Smaller batch sizes ($\approx 1$)
- Edge devices are cost, area and size limited
- Edge devices need to support both AI and non-AI applications
- Edge processors lack support for Keras, PyTorch, MXNet etc.
AI-RISC

Custom RISC-V processor with ISA extensions targeting AI applications

Tightly integrated AI accelerators for fine-grained offloading of AI tasks

End-to-end hardware/software co-design solution

Support for AI and non-AI applications on the same processor

vv8dn@virginia.edu
AI Functional Unit (AFU)

**APU** = Coarse-grained AI Processing Unit

**AFU** = Fine-grained AI Functional Unit

Image adapted from PRIME, ISCA 2016
Tightly Integrated AFU

Fetch
- Instruction Memory
  - Fetched Instruction

Decode
- Register File
  - Decode

Execute
- ALU
  - MAC AFU
  - PIM/Matrix AFU
  - Activation AFU

Memory
- Data Memory
  - LD/ST
  - PIM Memory

Writeback
- Writeback

vv8dn@virginia.edu
AI extensions

Grouped by corresponding AI Functional Units (AFU)

• Instructions in **bold** are already implemented
• All instructions utilize RISC-V CUSTOM-2 opcode space
AI extensions – General

• MAC
• Packed SIMD MAC
• Quantize
  • Accelerates cast and recast
  • Accelerate conditional clipping
• Transpose
  • Accelerates data layout changes
• Post increment ld/st
  • Already implemented in ASIP Designer RISC-V example
• Hardware loops
  • Already implemented in ASIP Designer RISC-V example
Preliminary Results with MAC Instruction

ResNet-8 TinyMLPerf Benchmark network on CIFAR-10 dataset

Quantized to int8

Instruction Count Improvement with MAC

Scalar MAC

SIMD MAC (4 lanes)
AI extensions – Matrix AFU

• Matrix data type
  • Vector of 8 elements with each element of 8 bits → 64 bits wide

• GEMM instructions
  • 2x4 * 4x2 → 2x2 result, 16 bits
  • Planned – 4x2 * 2x4 – require double/quad register for result

• GEMV instructions
  • 1x4 * 4x2
  • 2x4 * 4x1

• Vector-Vector multiplication
  • 1x8 * 8x1
AI extensions – PIM VMM AFU

• vmm.ld
  • Load results from PIM memory

• vmm.sd
  • Store weights to PIM memory

• vmm
  • Perform in-memory VMM operation
    • 1x8 * 8x8
    • 1x16 * 16x16
Preliminary Results with PIM VMM Instruction

Baseline – Simple C program implementing VMM as nested ‘for’ loops

Custom VMM instructions exposed to C via compiler intrinsics

1.6x better performance than writing Assembly

---

Preliminary Results with PIM VMM Instruction

<table>
<thead>
<tr>
<th>Size</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>8x8</td>
<td>10.1x</td>
</tr>
<tr>
<td>16x16</td>
<td>17.4x</td>
</tr>
</tbody>
</table>

Manuscript under preparation

vv8dn@virginia.edu
Planned AI extensions – Activation AFU

Hardware support for activation functions

• ReLU
• Sigmoid
• Tanh
• Swish
Planned AI extensions – Systolic AFU

• Load weights into the systolic array
• Store output activations – from systolic array to processor registers/scratchpad
• Prepare input activations – pre-processing instructions
• Systolic MAC
• Flush weights / stop
2-step Compiler & Issues

Solved by hardware designers
Hardware/Software Co-design

✓ End-to-end design methodology
  • TVM on the frontend
  • Synopsys ASIP Designer on the backend

✓ Support for multiple Domain Specific Language (DSL) frontends - Pytorch, MXNet, TFLite, Tensorflow, DarkNet etc.

✓ Verified with both 32 and 64-bit RISC-V

✓ Quantization support
ASIP Designer Enhancements

ISA extensions + microarchitecture AFU addition

TVM C codegen

Optionally replace generated AFU RTL with custom RTL

Hardware generation

RTL generator

Synthesizable RTL

RTL synthesizer

ASIC/FPGA

Processor model

Instruction set

Refinement

1. SDK Generation
2. Architectural optimization
3. Hardware generation
4. Verification

Instruction set simulator

Debugger and profiler

Binary

1. Optimizing C/C++ Compiler
2. Asm
3. Link

Optimizing C/C++ Compiler

Algorithm

C/C++

Architectural optimization and software development

1. Processor model
2. Instruction set

ISA extensions + microarchitecture AFU addition

PyTorch CoreML TensorFlow DarkNet Keras MXNet

vv8dn@virginia.edu
Compilers are hard!

- 2-step compilation – TVM + ASIP Designer generated C compiler
- Goal is to have no/minimum interaction with TVM generated C code

**Problem 1** – ASIP Compiler is not smart enough to detect opportunities for complex instructions like VMM, GEMM etc.
  - Works well for simple instructions like MAC.

**Solution** - Expose new instructions to TVM via compiler intrinsics.

- Split loops for convolution schedule in TVM
- Replace inner loops with PIM VMM instructions
- TVM calls AI-RISC VMM instructions in C code
Compiler issues with custom instructions

- **Problem 2** – Breaking the convolution schedule leads to accuracy issues.

- **Issue 1** – Wrong allocation of operands
  - Solved by TVM buffer and strided access of operands from bigger matrix

- **Issue 2** – Wrong data type in quantized NN
  - Defined input/output data types in TVM hardware intrinsic call
  - 8-bit inputs and 16/32-bit accumulated result.

- **Issue 3** – Kernel and Input data layout
  - TVM supports only a few Input/Filter layouts with specific ISA.
  - Adding support for required Input-Filter layout combinations in TVM for C hardware target used in AI-RISC.
More Compiler issues

• **Problem 3** – TVM support for breaking the convolution computation to match custom extension kernel size is limited.
  • TVM throws random errors when breaking the computation schedule using “tensorize” schedule pass.
  • Exact same convolution works with tensorize as a standalone kernel but not as a part of neural network.

• **Solution** – Trying to debug the exact issue but till we find a reliable solution we work with what we have.
  • Breaking the computation schedule into error-free parts and adapting the custom instructions accordingly for testing purposes.
TVM generated C code

```c
for (int32 t k outer = 0; k outer < 16; ++k outer) {
    for (int32 t y inner = 0; y inner < 8; ++y inner) {
        for (int32 t k inner = 0; k inner < 8; ++k inner) {
            ((int32 t*)compute1)[((z outer_y outer_fused * 8) + y inner)] = (((int32 t*)compute1)[((z outer_y outer_fused * 8) + y inner)] + (((int32 t*)compute1)[((k outer * 8) + k inner)]) * ((int32 t*)compute1)[((z outer_y outer_fused * 1024) + (y inner * 128))] + (k outer * 8)) + k inner));
        }
    }
}
```

```c
for (int32 t k outer = 0; k outer < 16; ++k outer) {
    (void)gemm 1x8x8 update ISQWNLHM(((int8 t *)placeholder + ((k outer * 8))), ((int8 t *)placeholder1 + (((z outer_y outer_fused * 1024) + (k outer * 8)))))
}
```

```c
void gemm 1x8x8 update ISQWNLHM(int8 t *aa, int8 t *bb, int32 t *cc, int A_stride, int B_stride, int C_stride) {
    for (int i = 0; i < 8; i++) {
        PIM_mem_store((char*)I, *((long*)(bb+i*B_stride)));
    }
    chess_memory_fence();
    long out0 = PIM_mac_0(*((long*) (aa)));
    long chess_storage[13] out1 = PIM_mac_1(*((long*) (aa)));
    long chess_storage[14] out2 = PIM_mac_2(*((long*) (aa)));
    long out3 = PIM_mac_3(*((long*) (aa)));
    long out[4] = { out0, out1, out2, out3);
    for (int i = 0; i < 8; i++) {
        cc[i] += *((int32 t*)Gout) + i;
    }
}
```

**Without Custom instructions**

**With Custom instructions**
Chess compiler intrinsic

promotion void PIM_mem_store ( char*, wchar8) = void PIM_mem_store (addr,w64);
promotion void PIM_mem_store ( char*, int) = void PIM_mem_store (addr,w64);
promotion void PIM_mem_store ( char*, long) = void PIM_mem_store (addr,w64);
promotion long PIM_mac_0 (long) = w64 PIM_vmm_0 (w64);
promotion long PIM_mac_1 (long) = w64 PIM_vmm_1 (w64);
promotion long PIM_mac_2 (long) = w64 PIM_vmm_2 (w64);
promotion long PIM_mac_3 (long) = w64 PIM_vmm_3 (w64);
Chess_view rule

```c
chess_view() {
    PIM_mem[pim_addr] = pim_wr;
} -> {
    PIM_mem_store(pim_addr,pim_wr);}

chess_view() {
    pim_rd_mac`EX` = PIM_mac[pim_addr=0`ID`]`EX`;
    pimOUT0`EX` = pimmac (pimIN`EX`, pim_rd_mac`EX`, pimOUT1`EX`, pimOUT2`EX`, pimOUT3`EX`);
} -> {
    pimOUT0`EX` = PIM_vmm_0(pimIN`EX`);
    pimOUT1`EX` = PIM_vmm_1(pimIN`EX`);
    pimOUT2`EX` = PIM_vmm_2(pimIN`EX`);
    pimOUT3`EX` = PIM_vmm_3(pimIN`EX`);}
```
Instruction definition nML

```c
opn pim_rr_instr(rdh: eX, rdl: eX, rs1: eX)
{
    action {
        stage ID..EX:
            pim_rd_mac`EX` = PIM_mac[pim_addr=0`ID`]`EX``;
        stage ID:
            pid_S1 = r1 = X[rs1];
        stage EX:
            //pimmac (pimIN, pim_rd_mac, pimOUTl, pimOUTh) @pim64x64;
            pimOUT0 = pimmac (pimIN, pim_rd_mac, pimOUT1, pimOUT2, pimOUT3) @pim64x64
            pex_D1 = tex_D1 = pimOUT0;
            pex_D2 = tex_D2 = pimOUT1;
            pex_D3 = tex_D3 = pimOUT2;
            pex_D4 = tex_D4 = pimOUT3;
        stage ME:
            pme_D1 = tme_D1 = pex_D1;
            pme_D2 = tme_D2 = pex_D2;
            pme_D3 = tme_D3 = pex_D3;
            pme_D4 = tme_D4 = pex_D4;
    }
    stage WB:
        if (rdl: x0) {
            w1 dead = w1 = pme_D1;
            w2 dead = w2 = pme_D2;
            w3 dead = w3 = pme_D3;
            w4 dead = w4 = pme_D4;
        }
        else {
            X[rdl] = w1 = pme_D1;
            X[13] = w2 = pme_D2;
            X[14] = w3 = pme_D3;
            X[rdh] = w4 = pme_D4;
        }
} syntax : "vmm" PADMMN " " rdh ", " PADMMN " " rdl ", " PADOP1 rs1 ;
image : "00000000"::rdh::rs1::"001"::rdl, class(pim_rrr);
```
Compiled assembly with new instructions

```
6970 0x01 0x50 0x30 0x5b vmm.sd x21, 0(x0)
6974 0x01 0x40 0x30 0xdb vmm.sd x20, 1(x0)
6978 0x08 0x01 0xba 0x0b ld x20, 128(x3!)
6982 0x01 0x40 0x31 0x5b vmm.sd x20, 2(x0)
6986 0x08 0x01 0xba 0x0b ld x20, 128(x3!)
6990 0x01 0x40 0x31 0xdb vmm.sd x20, 3(x0)
6994 0x08 0x01 0xba 0x0b ld x20, 128(x3!)
6998 0x01 0x40 0x32 0x5b vmm.sd x20, 4(x0)
7002 0x08 0x01 0xba 0x0b ld x20, 128(x3!)
7006 0x01 0x40 0x32 0xdb vmm.sd x20, 5(x0)
7010 0x08 0x01 0xba 0x0b ld x20, 128(x3!)
7014 0x01 0x40 0x33 0x5b vmm.sd x20, 6(x0)
7018 0xc8 0x81 0xba 0x0b ld x20, -888(x3!)
7022 0x01 0x40 0x33 0xdb vmm.sd x20, 7(x0)
7026 0x00 0x83 0x3a 0x0b ld x20, 8(x6!)
7030 0x41 0x1c c.lw x15, 0(x10)
7032 0x00 0xca 0x15 0xdb vmm x12, x11, x20
7036 0x00 0x05 0x84 0x1b sext.w x8, x11
vv8dn@virginia.edu
```
Emulation/Definition of AFU

```c
w64 pimmac(w64 a, v8w64 pim_rd_mac, w64 & out1, w64 & out2, w64 & out3) {
  v8w32 r = 0;
  v8w08 aa = a;
  w64 out0;
  for (int i = 0; i < 8; i++){
    for (int j = 0; j < 8; j++){
      v8w08 temp = pim_rd_mac[j];
      int32_t inter = 0;
      inter = aa[j] * temp[i];
      r[i] += inter;
    }
  }
  out0 = r[1]; r[0];
  out1 = r[3]; r[2];
  out2 = r[5]; r[4];
  out3 = r[7]; r[6];
  return out0;
}
```
Instructions through instruction viewer
Results

Performance improvements with AI-RISC
Evaluation Methodology

- **Benchmark**
  - ResNet-8 from TinyMLPerf
  - GEMV kernel
- **Baseline**
  - 5-stage in-order 64-bit RISC-V RV64IMC
- **Compilation**
  - TVM (TFLite to C) + Custom C compiler (C to binary)
- **Simulation Framework**
  - Cycle-accurate Simulator
Speedup on GEMV kernel

- A Matrix → 8x8
- B Vector → 1x8
- Input datatype → int8
- Output datatype → int16

Speedup on GEMV kernel

<table>
<thead>
<tr>
<th></th>
<th>Plain RISC-V</th>
<th>RISC-V + MAC</th>
<th>RISC-V + GEMM 1x8x1 + MAC</th>
<th>RISC-V + VMM + MAC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speedup</td>
<td>1.00</td>
<td>1.75</td>
<td>5.68</td>
<td>17.63</td>
</tr>
</tbody>
</table>
Speedup on single CONV2D kernel

- Input image → 7x7
- Input channels → 8
- Filter → 2x2
- Output channels → 2
- Input datatype → int8
- Output datatype → int16
- data_layout → NHWC
- kernel_layout → HWIO
Speedup on ResNet-8 network from TinyMLPerf

Number of processor Cycles

- Plain RISC-V
- RISC-V + MAC: 1.41x
- RISC-V + PIM VMM + MAC: 4.41x
- RISC-V + GEMM 1x8x1 + MAC: 1.95x
Design-Space Exploration for PIM VMM

vv8dn@virginia.edu
Area Overhead

- Plain RISC-V: 0.00%
- RISC-V + MAC: 14.93%
- RISC-V + PIM2x2 VMM + MAC: 18.23%
- RISC-V + PIM2x4 VMM + MAC: 19.56%
- RISC-V + PIM2x8 VMM + MAC: 22.79%
- RISC-V + PIM4x2 VMM + MAC: 18.57%
- RISC-V + PIM4x4 VMM + MAC: 20.03%
- RISC-V + PIM4x8 VMM + MAC: 22.79%
- RISC-V + PIM8x2 VMM + MAC: 19.24%
- RISC-V + PIM8x4 VMM + MAC: 20.94%
- RISC-V + PIM8x8 VMM + MAC: 26.78%
FoM for optimal PIM size

Figure of Merit = Performance gain / Area Overhead
Vector “V” extension addition to AI-RISC

- Undergrad project by Nate Hunter, Noah Mills, Greg Vavoso and Bill Yang.
- Subset of Vector extension instructions added to AI-RISC:
  - Addition and Subtraction
  - Logical AND/OR/XOR/MIN/MAX
  - Shift and Move
  - Multiplication

<table>
<thead>
<tr>
<th>Example</th>
<th>Vector instructions</th>
<th>Scalar processor cycles</th>
<th>Vector processor cycles</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subtract on 128-byte vectors</td>
<td>vsub.vv</td>
<td>128</td>
<td>1</td>
<td>128</td>
</tr>
<tr>
<td>Dot product on 32-word vectors</td>
<td>vmul.vv, vredsum.vs</td>
<td>33 + 31 = 64</td>
<td>2 + 5 = 7</td>
<td>9.1</td>
</tr>
</tbody>
</table>
Summary

AI-RISC = Custom RISC-V processor for Edge AI

Tightly integrated AI Functional Units (AFU)

Custom ISA extensions to RISC-V

Complete SDK generation including compiler support for PyTorch, TensorFlow etc.

Speedup compared to RV64IMC → 17.6x for GEMV, 4.4x for ResNet-8

Scalable, flexible and support for both AI and non-AI applications
Publications


- V. Verma, M. Stan, “AI-RISC: Scalable open source processor for AI applications at edge of IoT,” Design Automation Conference Young Student Fellow Program poster session (DAC YFP), July 2020. (Best Poster Award)
Questions?

- This work is funded by SRC under GRC AIHW task 2945.001.