Low Power Multicore Solutions for Approximation

Luca Benini  lbenini@iis.ee.ethz.ch,luca.benini@unibo.it
ML Processors from Tiny to Huge

Legend

Computation Precision
- analog
- int1
- int2
- int4.8
- int8
- int8.32
- int16
- int12.16
- int32
- fp16
- fp16.32
- fp32
- fp64

Form Factor
- Chip
- Card
- System

Computation Type
- Inference
- Training
TinyML challenge
AI capabilities in the power envelope of an MCU: **10-mW peak (1mW avg)**
TinyML Workloads – DNNs (and More)

90.2%, 480M-param, many GOPS

70%, “Tiny” DNNs

5M-param

High OP/B ratio
Massive Parallelism
MAC-dominated
Low precision OK

“Model zoo”: very fast evolution → need programmable solutions
ML on MCUs?

High performance MCUs

Low-Power MCUs

1TOPS/W=1pJ/OP → TinyML (1 GOPs/Inf) @10fps in 10mW

Courtesy of J Pineda, NXP + Updates
Energy efficiency @ GOPS is the Challenge

ARM Cortex-M MCUs: M0+, M4, M7 (40LP, typ, 1.1V)*

High performance MCUs

*data from ARMs web
“Classical” core performance scaling trajectory

- Faster CLK → deeper pipeline → IPC drops
- Recover IPC → superscalar → ILP bottleneck (dependencies)
- Mitigate ILP bottlenecks → OOO → huge power, area cost!
A way Out: Processor Specialization

3-cycle ALU-OP, 4-cyle MEM-OP $\Rightarrow$ only IPC loss: LD-use, Branch

**Baseline RISC** (not good for ML)

**V2**
- Data motion (e.g. auto-increment)
- Data processing (e.g. MAC)

**V3**
- Domain specific data processing
- Narrow bitwidth
- HW support for special arithmetic

ISA extension cost 25 kGE $\rightarrow$ 40 kGE (1.6x), energy efficient if $0.6T_{exec}$

[Gautschi et al. TVLSI 2017]
RISC-V Instruction Set Architecture

- Started by UC-Berkeley in 2010
- Contract between SW and HW
  - Partitioned into user and privileged spec
  - External Debug
- Standard governed by RISC-V foundation
  - ETHZ is a founding member of the foundation
  - Necessary for the continuity
- Defines 32, 64 and 128 bit ISA
  - No implementation, just the ISA
  - Different implementations (both open and close source)
- At ETHZ+UNIBO we specialize in efficient implementations of RISC-V cores
RISC-V Foundation Members

A modern, open, free ISA, extensible by construction
Endorsed and Supported by 1000+ Companies
RISC-V ISA Baseline and Extensions

- Kept very simple and extendable
  - Wide range of applications from IoT to HPC

- RV + word-width + extensions
  - RV32IMC: 32bit, integer, multiplication, compressed

- User specification:
  - Separated into extensions, only I is mandatory

- Privileged Specification (WIP):
  - Governs OS functionality: Exceptions, Interrupts
  - Virtual Addressing
  - Privilege Levels

<table>
<thead>
<tr>
<th>I</th>
<th>Integer instructions (frozen)</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>Reduced number of registers</td>
</tr>
<tr>
<td>M</td>
<td>Multiplication and Division (frozen)</td>
</tr>
<tr>
<td>A</td>
<td>Atomic instructions (frozen)</td>
</tr>
<tr>
<td>F</td>
<td>Single-Precision Floating-Point (frozen)</td>
</tr>
<tr>
<td>D</td>
<td>Double-Precision Floating-Point (frozen)</td>
</tr>
<tr>
<td>C</td>
<td>Compressed Instructions (frozen)</td>
</tr>
<tr>
<td>X</td>
<td>Non Standard Extensions</td>
</tr>
</tbody>
</table>
### Basic Instructions (I)

- Load Rd, Imm32
- Load Rd, Mem
- Load Rd, Mem indexed
- Load Rd, Mem base
- Add Rd, Rs,_imm
- Add Rd, Rs, Rs
- Add Rd, Rs, Rs_16
- Add Rd, Rs, Rs_32
- Add Rd, Rs, Rs_64
- Sub Rs, Rs, imm
- Sub Rs, Rs, Rs
- Sub Rs, Rs, Rs_16
- Sub Rs, Rs, Rs_32
- Sub Rs, Rs, Rs_64
- Nop
- Store Rd, Mem
- Store Rd, Mem indexed
- Store Rd, Mem base
- Branch
- Branch beq
- Branch bne
- Branch bgt
- Branch bge
- Branch bhi
- Branch bhlt

### Privilege Mode

- RISC-V supports 3 privilege levels: User, Supervisor, and Hypervisor.
- User level is for normal program execution.
- Supervisor level is used for operating system functions.
- Hypervisor level is the highest level and is used for virtualization.

### Compressed Instructions (C)

-指令压缩

### Floating Point Extensions

- Floating Point Instructions
- Load FP
- Store FP
- Add FP
- Sub FP
- Multiply FP
- Divide FP
- Compare FP

### Atomic Extensions (A)

- Atomic Load/Store
- Atomic Compare/Exchange
- Atomic Increment
- Atomic Decrement

### Multiply/Divide (M)

- Multiply/Multiply
- Divide/Divide
- Multiply-accumulate

---

**Note:**

- This slide provides an overview of RISC-V instruction set architecture, focusing on various instruction categories and their functionalities.
- It highlights the support for different privilege modes and the benefits of compressed instructions for efficient memory usage.
- The floating point and atomic extensions are crucial for applications requiring high-performance computing and low-level operations.
- The multiply/divide instructions are essential for tasks involving mathematical calculations.
RISC-V Architectural State

- There are 32 registers, each 32 / 64 / 128 bits long
  - Named x0 to x31
  - x0 is hard wired to zero
  - There is a standard ‘E’ extension that uses only 16 registers (RV32E)

- In addition one program counter (PC)
  - Byte based addressing, program counter increments by 4/8/16

- For floating point operation 32 additional FP registers

- Additional Control Status Registers (CSRs)
  - Encoding for up to 4’096 registers are reserved. Not all are used.
RISC-V Instructions four basic types

- **R** register to register operations
- **I** operations with immediate/constant values
- **S / SB** operations with two source registers
- **U / UJ** operations with large immediate/constant value

<table>
<thead>
<tr>
<th>Field</th>
<th>R-type</th>
<th>I-type</th>
<th>S-type</th>
<th>U-type</th>
</tr>
</thead>
<tbody>
<tr>
<td>funct7</td>
<td>rs2</td>
<td>rs1</td>
<td>funct3</td>
<td>rd</td>
</tr>
<tr>
<td>rs2</td>
<td>rs1</td>
<td>rs1</td>
<td>funct3</td>
<td>rd</td>
</tr>
<tr>
<td>imm[31:12]</td>
<td></td>
<td></td>
<td></td>
<td>rd</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>opcode</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>opcode</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>opcode</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>opcode</td>
</tr>
</tbody>
</table>
RISC-V is a load/store architecture

- All operations are on internal registers
  - Can not manipulate data in memory directly
- Load instructions to copy from memory to registers
- R-type or I-type instructions to operate on them
- Store instructions to copy from registers back to memory
- Branch and Jump instructions
- 1/3 ALU utilization if operands are from/to memory (LD, ALU, ST)
### Encoding of the instructions, main groups

- **Reserved** opcodes for standard extensions
- Rest of opcodes free for custom implementations
- Standard extensions will be frozen/not change in the future

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>LOAD</td>
<td>LOAD-FP</td>
<td><strong>custom-0</strong></td>
<td>MISC-MEM</td>
<td>OP-IMM</td>
<td>AUIPC</td>
<td>OP-IMM-32</td>
<td>48b</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>STORE</td>
<td>STORE-FP</td>
<td><strong>custom-1</strong></td>
<td>AMO</td>
<td>OP</td>
<td>LUI</td>
<td>OP-32</td>
<td>64b</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>MADD</td>
<td>MSUB</td>
<td>NMSUB</td>
<td>NMADD</td>
<td>OP-FP</td>
<td><strong>reserved</strong></td>
<td>custom-2/rv128</td>
<td>48b</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>BRANCH</td>
<td>JALR</td>
<td><strong>reserved</strong></td>
<td>JAL</td>
<td>SYSTEM</td>
<td><strong>reserved</strong></td>
<td>custom-3/rv128</td>
<td>≥ 80b</td>
<td></td>
</tr>
</tbody>
</table>

**Extensibility is integral to RISC-V ISA design!**
How to get efficiency: ISA extensions

Only one MACs every 4 cycles!

1. Post modified LD/ST
2. MAC
3. HW loop
4. Packed-SIMD operations with dot product
5. Shuffle operations for vectors
6. Mac-load

Post increment LD/ST

- **Automatic address update**
  - Update base register with computed address after the memory access
  - Save instructions to update address register
  - Post-increment:
    - Base address serves as memory address

- **Offset can be stored in:**
  - Register
  - Immediate

\[ c = 0; \]
\[ \text{for}(i=0;i<100;i++) \]
\[ c = c + a[i]*b[i]; \]

**Original RISC-V**

\[
\begin{align*}
\text{addi} & \ \text{x4, x0, 64} \\
\text{Lstart :} & \\
\text{lb} & \ \text{x2, 0(x10)} \\
\text{lb} & \ \text{x3, 0(x12)} \\
\text{addi} & \ \text{x10, x10, 1} \\
\text{addi} & \ \text{x12, x12, 1} \\
\text{.....} & \\
\text{bne} & \ \text{x2,x3, Lstart}
\end{align*}
\]

**Auto-incr load/store**

\[
\begin{align*}
\text{addi} & \ \text{x4, x0, 64} \\
\text{Lstart :} & \\
\text{lb} & \ \text{x2, 0(x10!)} \\
\text{lb} & \ \text{x3, 0(x12!)} \\
\text{.....} & \\
\text{bne} & \ \text{x2,x3, Lstart}
\end{align*}
\]

\[ \Rightarrow \text{save 2 additional instructions to update the read addresses of the operands!} \]
Hardware loops

- **Hardware loop setup with:**
  - 3 separate instructions
    - `lp.start`, `lp.end`, `lp.count`, `lp.counti`
    - No restriction on start/end address
  - **Fast setup instructions**
    - `lp.setup`, `lp.setupi`
    - Start address = PC + 4
    - End address = start address + offset
    - Counter from immediate/register

---

Original RISC-V

```
// initialize counter
mv   x4, 100
// init accumulator
mv   x5, 0
Lstart:
  // decrement counter
  addi x4, x4, -1
  // load elements from mem
  lw   x8, 0(x9)
  lw   x10, 0(x11)
  // update memory pointers
  add  x9, x9, 4
  add  x11, x11, 4
  // mac
  mul  x8, x8, x10
  add  x5, x5, x8
bne  x4, x0, Lstart
```

HW Loop Ext

```
// init accumulator
mv   x5, 0
// set number iterations, start and end of the loop
lp.setupi 100, Lend
  // load elements from mem
  lw   x8, 0(x9)
  lw   x10, 0(x11)
  // update memory pointers
  add  x9, x9, 4
  add  x11, x11, 4
  // mac
  mul  x8, x8, x10
Lend:  add  x5, x5, x8
```

No counter and branch overhead!

```c
\[ c = 0; \\
    for(i=0; i<100; i++) \\
    c = c + a[i]*b[i]; \\
\]```
Multiply Accumulate

- Accumulation on 32 bit data p.mac
  - Directly on the register file
  - Pro:
    - Faster access to mac accumulation
    - Single cycle mult/mac
  - Cons:
    - Additional read port on the register file
    - used for pre/post increment with register

```c
int acc=0, coeff[N], inp[N];
for(int i=0; i<N; i++)
    acc += coeff[i] * inp[i];
```

```c
acc = __builtin_pulp_mac (inp[i], coeff[i], acc);
```

**Intrinsics**: special functions that map directly to inlined DSP instructions.
However, the compiler can already place the p.mac instruction into the above code!
Xpulp Extentions: packed-SIMD

Remember: DNN inference is OK with low-bitwidth operands

- packed-SIMD extensions
  - Make usage of resources the best in performance with little overhead
  - Target for embedded systems, RVV is for high performance
  - pSIMD in 32bit machines
  - Vectors are either 4 8bits-elements or 2 16bits-elements
  - pSIMD instructions

<table>
<thead>
<tr>
<th>Computation</th>
<th>add, sub, shift, avg, abs, dot product</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compare</td>
<td>min, max, compare</td>
</tr>
<tr>
<td>Manipulate</td>
<td>extract, pack, shuffle</td>
</tr>
</tbody>
</table>
Xpulp Extensions: packed-SIMD

- **Same Register-file**
  - The instruction encode how to interpret the content of the register

<table>
<thead>
<tr>
<th>add rD, rs1, rs2</th>
<th>rD = 0x03020100 + 0x0D0C0B0A</th>
</tr>
</thead>
</table>
| add.h rD, rs1, rs2 | rD[0] = 0x0100 + 0x0B0A  
                      rD[1] = 0x0302 + 0x0D0C |
| add.b rD, rs1, rs2 | rD[0] = 0x00 + 0x0A  
                      rD[1] = 0x01 + 0x0B  
                      rD[2] = 0x02 + 0x0C  
                      rD[3] = 0x03 + 0x0D |
Advanced ALU for Xpulp extensions

- Optimized datapath to reduce resources
- Multiple-adders for round
- Adder followed by shifter for fixed point normalization
- Clip unit uses one adder as comparator and the main comparator
Expanding SIMD Dot Product

- Dot Product: (half word example)

  \[ \text{32 bit} \quad \text{32 bit} \quad \text{32 bit} \]

  → 2 multiplications, 1 addition, 1 accumulation in 1 cycle (2x for bytes)
MUL architecture

16x16b with sign selection for short multiplications [with round and normalization]. 5 cycles FSM for higher 64-bits (mulh* instructions)

32x32b single cycle MAC/MUL unit

16x16b short parallel dot product

8x8b byte parallel dot product

clock gating to reduce switching activity between the scalar and SIMD multipliers
Reference & Examples on Compiler Builtins

SIMD Instructions of the Xpulp ISA extension

<table>
<thead>
<tr>
<th>Computation</th>
<th>add, sub, shift, avg, abs, dot product</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compare</td>
<td>min, max, compare</td>
</tr>
<tr>
<td>Manipulate</td>
<td>extract, pack, shuffle</td>
</tr>
</tbody>
</table>

Dot-product without accumulation between unsigned char vectors (v4u):


Dot-product without accumulation between signed char vectors (v4s):


Also with mixed signs:


Similar builtins without accumulation for short vectors:

\[ S = \_\_builtin\_dotup2(A, B); \quad S = \_\_builtin\_doptsp2(A, B); \quad S = \_\_builtin\_doptusp2(A, B); \]

All of these are also available with accumulation (over accumulator S):

\[ S = \_\_builtin\_sdotup4(A, B, S); \quad S = \_\_builtin\_sdoptsp4(A, B, S); \quad S = \_\_builtin\_sdoptusp4(A, B, S); \]

\[ S = \_\_builtin\_sdotup2(A, B, S); \quad S = \_\_builtin\_sdoptsp2(A, B, S); \quad S = \_\_builtin\_sdoptusp2(A, B, S); \]
ISA Extensions at Work

- The innermost loop has 4x less iterations
  - 4 bytes per matrix are loaded as a 32b word
  - Dot product with accumulation performs in 1 cycle 4 macs

...  ... //iterate #COL/4
lpsetup x1,a4,stop1 lpsetup x1,a6,stop1
plbu a0,1(a3!) plw a1,4(t1!) //load 4-bytes with post inc
plbu a1,32(a2!) plw a5,4(t3!)
stop1: p.mac a5,a0,a1 stop1: pv.sdotsp.b a7,a1,a5 //4 mac
.... ........
for (i = 0; i < 100; i++)
    d[i] = a[i] + b[i];

<table>
<thead>
<tr>
<th>Baseline</th>
<th>Auto-incr load/store</th>
<th>HW Loop</th>
<th>Packed-SIMD</th>
</tr>
</thead>
<tbody>
<tr>
<td>mv   x5, 0</td>
<td>mv   x5, 0</td>
<td>lp.setupi 100, Lend</td>
<td>lp.setupi 25, Lend</td>
</tr>
<tr>
<td>mv   x4, 100</td>
<td>mv   x4, 100</td>
<td>lb   x2, 0(x10)</td>
<td>lw  x2, 0(x10!)</td>
</tr>
<tr>
<td>Lstart:</td>
<td>Lstart:</td>
<td>lb   x3, 0(x11!)</td>
<td>lw  x3, 0(x11!)</td>
</tr>
<tr>
<td>lb   x2, 0(x10)</td>
<td>lb   x2, 0(x10!)</td>
<td>addi x4, x4, -1</td>
<td>pv.add.b x2, x3,  x2</td>
</tr>
<tr>
<td>lb   x3, 0(x11)</td>
<td>lb   x3, 0(x11!)</td>
<td>add i x4, x4, -1</td>
<td></td>
</tr>
<tr>
<td>addi x10,x10, 1</td>
<td>add  x2, x3, x2</td>
<td>add  x2, x3, x2</td>
<td></td>
</tr>
<tr>
<td>addi x11,x11, 1</td>
<td>sb   x2, 0(x12!)</td>
<td>sb   x2, 0(x12!)</td>
<td>Lend: sb x2, 0(x12!)</td>
</tr>
<tr>
<td>add  x2, x3, x2</td>
<td>bne   x4, x5,</td>
<td>bne   x4, x5,</td>
<td></td>
</tr>
<tr>
<td>sb   x2, 0(x12!)</td>
<td>Lstart</td>
<td>Lstart</td>
<td>Lend: sw x2, 0(x12!)</td>
</tr>
<tr>
<td>bne   x4, x5,</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lstart</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

11 cycles/output  8 cycles/output  5 cycles/output  1,25 cycles/output

Note: All-int computation – hard to quantize. What about FP? See Luca Bertaccini’s lecture!
Advanced SIMD: Shuffle Instruction

- In order to use the vector unit the elements have to be aligned in the register file
- Shuffle allows to recombine bytes into 1 register:
  
  \[
  \text{pv.shuffle2.b } rD, rA, rB
  \]

  \[
  \begin{align*}
  rD\{3\} &= (rB[26]==0) \ ? \ rA: rD \{rB[25:24]\} \\
  rD\{2\} &= (rB[18]==0) \ ? \ rA: rD \{rB[17:16]\} \\
  rD\{1\} &= (rB[10]==0) \ ? \ rA: rD \{rB[ 9: 8]\} \\
  rD\{0\} &= (rB[ 2]==0) \ ? \ rA: rD \{rB[ 1: 0]\}
  \end{align*}
  \]

- With \( rX[i] = rX[(i+1)*8-1:i*8] \)
Shuffle for Direct SIMD Convolution

Convolution in registers
5x5 convolutional filter

- 7 Sum-of-dot-product
- 4 move
- 1 shuffle
- 3 lw/sw
- ~ 5 control instructions

Significant benefit in reuse of registers and less LD/ST
GEMM-based Convolution

8-bit Convolution example

CMSIS-NN based Matrix Multiplication Layout: 2x2

\[
\begin{array}{ccc}
\times & x1 & x2 \\
\end{array}
\]

PULP-NN Matrix Multiplication Layout: 4x2

\[
\begin{array}{ccc}
\times & x1 & x2 \\
\end{array}
\]

RegisterFile of the RISCY core: 32 general purpose registers

2x2: 43% utilization

4x2: 69% utilization

More Data Reuse & Higher utilization of the RF

Peak Performance (8 cores)

- 2x2: 12.8 MAC/cyc
- 4x2: 15.5 MAC/cyc

Never underestimate the importance of registers! How to get “more”?
Achieving 100% dotp Unit Utilization

8-bit Convolution

**RV32IMC**

- `addi a0, a0, 1`
- `addi t1, t1, 1`
- `addi t3, t3, 1`
- `addi t4, t4, 1`
- `lbu a7, -1(a0)`
- `lbu a6, -1(t4)`
- `lbu a5, -1(t3)`
- `lbu t5, -1(t1)`
- `mul s1, a7, a6`
- `mul a7, a7, a5`
- `add s0, s0, s1`
- `mul a6, a6, t5`
- `add s0, s0, s1`
- `mul a6, a6, t5`
- `add s0, s0, s1`
- `mul a6, a6, t5`
- `add s0, s0, s1`
- `bne s5, a0, 1c000bc`

**RV32IMCXpulp**

- `addi a0, a0, 1`
- `addi t1, t1, 1`
- `addi t3, t3, 1`
- `addi t4, t4, 1`
- `lbu a7, -1(a0)`
- `lbu a6, -1(t4)`
- `lbu a5, -1(t3)`
- `lbu t5, -1(t1)`
- `mul s1, a7, a6`
- `mul a7, a7, a5`
- `add s0, s0, s1`
- `mul a6, a6, t5`
- `add s0, s0, s1`
- `mul a6, a6, t5`
- `add s0, s0, s1`
- `mul a6, a6, t5`
- `add s0, s0, s1`
- `bne s5, a0, 1c000bc`

**8-bit SIMD sdotp**

- `lp.setup`
- `p.lw w1, 4(a0)`
- `p.lw x1, 4(a1)`
- `pv.sdotsp.b s1, w1, x1`
- `pv.sdotsp.b s2, x1, w1`
- `pv.sdotsp.b s3, w2, x1`
- `pv.sdotsp.b s4, x2, w2`

**8-bit sdotp + LD**

- `lp.setup`
- `p.lw w2, 4(a1)`
- `pv.sdotsp.b s1, w1, x1`
- `pv.sdotsp.b s2, x1, w1`
- `pv.sdotsp.b s3, w2, x1`
- `pv.sdotsp.b s4, x2, w2`

Yes! dotp+ld

- `N/4`
- `Can we remove?`
- `Init NN-RF (outside of the loop)`
- `lp.setup`
- `pv.nnsdotup.h s0, ax1, 9`
- `pv.nnsdotsp.b s1, aw2, 0`
- `pv.nnsdotsp.b s2, aw4, 2`
- `pv.nnsdotsp.b s3, aw3, 4`
- `pv.nnsdotsp.b s4, ax1, 14`
- `end`

9x less instructions than RV32IMC

14.5x less instructions at an extra 3% area cost (~600GEs)
Hardware for dotp+ld

NN Register File: 6 32-bit registers (weights and input activations)

Special-purpose registers
Not only RISC-V: Armv8.1-M

- New embedded vector ISA **Helium (MVE)**
  - Uses 8 128-bit vector registers (reuses the 32 FP registers)
  - ISA enhancements for loops, branches (Low Overhead Branch Extension)
  - Instructions for half-precision floating-point support
  - Enhancements in debug including performance monitoring unit (PMU) and additional debug support to focus on signal processing application developments.

- Being able to set a breakpoint which triggers (halts code execution and passes control to the debugger) when a certain count value is reached and being able to set a data watchpoint with a bit mask for data value comparison (for example, for looking for a signal value to be within a certain range).
ARM MVE Vectors

- Helium provides a SIMD capability for Cortex-M CPUs: a set of 128-bit registers are provided which can be used to hold, e.g. 16 separate 8-bit values. A single instruction can operate on each value independently (with predication).
- Extension of Arm Thumb
- Helium instructions operate on vectors of elements of the same data type: Int/FP
  - Integer elements may be signed or unsigned 8-, 16-, 32-bit, fixed-point saturating (Q7, Q15, Q31)
  - Floating-point elements may be single (32-bit) or half precision (16-bit).
- The position of an element in a vector is called lane
ARM MVE Vector Execution Model

- MVE permits instruction execution to be interleaved
  - Multiple instructions may overlap in the pipeline execute stage. For example, a Vector Load (VLDR) instruction which reads multiple words from memory into a vector register may execute at the same time as a Vector Multiply (VMUL) instruction which uses that data
  - It is up to the CPU hardware designer to decide how many “beats” are executed on each clock cycle (eg. 32-bit datapath vs 64-bit datapath)

- Complicates exception handling
  - D of the VLDR happens after beat A of the VMLA has completed. If memory for beat D triggers a fault, the processor needs to remember that the following instruction was part executed (storing a value which shows which beats have already been executed). If after exception handling, the program returns to this location, the hardware already knows which beats should not be re-executed
Cortex-M55 Performance

- Performance relative to Cortex-M4
- Major improvements for Q7, FP16 (new datatypes in HW)
- ML benchmark (KWS): MFCC, DNN (2 conv, 3 FC layers), 8-bit (w, act), 80-500KB, accuracy 90%-95%
Quantized Neural Networks (QNNs) are a natural target for execution on constrained extreme edge platforms.

SoA Quantization Results

Quantization of a MobilenetV1_224_1.0 (*)

<table>
<thead>
<tr>
<th>Quantization Method</th>
<th>Top1 Accuracy</th>
<th>Weight Memory Footprint</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full-Precision</td>
<td>70.9%</td>
<td>16.27 MB</td>
</tr>
<tr>
<td>INT-8</td>
<td>70.1%</td>
<td>0.8% 4.06 MB</td>
</tr>
<tr>
<td>INT-4</td>
<td>66.46%</td>
<td>4.4% 2.35 MB</td>
</tr>
<tr>
<td>Mixed-Precision</td>
<td>68%</td>
<td>2.9% 2.09 MB</td>
</tr>
</tbody>
</table>

Mixed-precision approach key to meet the memory constraints of tiny devices

Sub-byte operands manipulation

32-bit data load with post increment (one cycle)

To MAC units

Disassembled Pseudocode

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>p.lw Src, 0(a0);</td>
<td>// vectorial load</td>
</tr>
<tr>
<td>p.bextract w1e, Src, 4, 0;</td>
<td>//bextract built-in</td>
</tr>
<tr>
<td>p.bextract w2e, Src, 4, 4;</td>
<td></td>
</tr>
<tr>
<td>p.bextract w3e, Src, 4, 8;</td>
<td></td>
</tr>
<tr>
<td>p.bextract w4e, Src, 4, 12;</td>
<td></td>
</tr>
<tr>
<td>p.packhi.b Res, w3e, w4e;</td>
<td>//pack built-in</td>
</tr>
<tr>
<td>p.packlo.b Res, w1e, w2e;</td>
<td>//two assembly insns</td>
</tr>
</tbody>
</table>
Mixed Precision SIMD Processor

- Can support all variants:
  - 16x16, 16x8, 16x4, 16x2
  - 8x8, 8x4, 8x2
  - 4x4, 4x2
  - 2x2

- Avoids Pack/unpack Overheads
- Maximized performance (SIMD)
- Maximizes RF use (Data Locality)

How to encode all these instructions?
Mixed-Precision Core: New Formats Required

- dotp variants
- add variants
- sub variants
- avg variants
- shift variants
- max variants
- min variants
- abs variants

... > 500 instructions
Virtual SIMD Instructions

- Encode operation as a virtual SIMD in the ISA (e.g. `sdotsp.v`)
- Format specified at runtime by a Control Register (e.g. 4x4)
- 180 → 18 Instructions needed for SIMD DOTP
- Potential to avoid code replication for different formats
- Tiny Overhead on QNN for Switching format

Format switch not frequent in DNN, e.g. every layer.
Processor HW extension

- **Goal**
  - HW support for mixed-precision SIMD instructions;

- **Challenge**
  - Enormous number of instructions to be encoded in the ISA;

- **Solution**
  - Status-based execution.
Extended Dot-Product Unit

Multi-Precision Integer Dotp-Unit

OpC
(32b scalar)

OpA
(32b SIMD vector)

OpB
(32b SIMD vector)

SLICER AND ROUTER

2x32b → 32b
Adder Tree

4x18b → 32b
Adder Tree

8x10b → 32b
Adder Tree

16x6b → 32b
Adder Tree

Output Result Mux

Dotp Result
(32b scalar)
Xpulp Extensions Performance (Single Core)

up to 11x

Bottom line: pJ/OP is achievable on single core for ML workloads

Nice – But what about the GOPS? Faster+Superscalar is not efficient!

M7: 5.01 CoreMark/MHz-58.5 µW/MHz
M4: 3.42 CoreMark/MHz-12.26 µW/MHz
ML & Parallel Near Threshold → PULP

- As VDD decreases, operating speed decreases
- However efficiency increases → more work done per Joule
- Until leakage effects start to dominate
- Put more units in parallel to get performance up and keep them busy with a parallel workload

ML is massively parallel and scales well (P/S ↑ with NN size)

Efficiency vs VDD chip01

- Better to have N× PEs running at lower voltage than one PE at nominal voltage!

Efficiency (mac/mW) vs. VDD (V)

- As VDD decreases, operating speed decreases.
- However, efficiency increases → more work done per Joule.
- Until leakage effects start to dominate.
- Put more units in parallel to get performance up and keep them busy with a parallel workload.

ML is massively parallel and scales well (P/S ↑ with NN size).
Multiple RI5CY Cores (1-16)
Low-Latency Shared TCDM

Tightly Coupled Data Memory

Mem Mem Mem Mem Mem

Mem Mem Mem Mem Mem

Logarithmic Interconnect

RISC-V core RISC-V core RISC-V core RISC-V core

CLUSTER
High speed single clock logarithmic interconnect

Ultra-low latency $\rightarrow$ short wires + 1 clock cycle latency

World-level bank interleaving «emulates» multiported mem

Fast synchronization and Atomics

Synchronization & Events

- event signalling
- execution synchronization
- execution control
- exclusive resources manag.

Avoid busy waiting!
Minimize sw synchro. overhead
Efficient fine-grain parallelization

Private, per core port
→ single cycle latency
→ no contention

external cluster
Results: Barrier

- Fully parallel access to SCU: Barrier cost constant
- Primitive energy cost: Down by up to 30x
- Minimum parallel section for 10% overhead in terms of ...
  - ... cycles: ~100 instead of > 1000 cycles
  - ... energy: ~70 instead of > 2000 cycles
PULP for ML (DNNs) Speedup

- 8-bit convolution
  - Open source DNN library
- 10x through xPULP
  - Extensions bring real speedup
- Near-linear speedup
  - Scales well for regular workloads
- 75x overall gain

[Garofalo et al. Philos. Trans. R. Soc 20]
8-Cores Cluster + XpulpNN + M&L (22nm)
Addressing Multicore Inefficiencies

Power analysis of a parallel 8-bit x 4-bit convolution

- Reduce unnecessary power consumption (not spent in computation)
- Exploit convolution’s instruction and memory data access pattern regularity
- Increase energy efficiency at low extra-area cost
  - reconfigurable MIMD/SIMD architecture

Unnecessary power consumption

- CORE_0: 4%
- CORES_ID_EX: 43.3%
- CORES_IF: 10%
- I$: 20%
- TCDM.: 22%
The Power of SIMD

- Cores enter in SIMD (VLEM) mode when executing regular kernels (in two clock cycles)
- In SIMD, instruction flow orchestrated only by the MAIN core → Less energy
- Cores resume in MIMD mode on divergent branches (..or control tasks) → Flexibility

[Garofalo et al. ESSCIRC 2021]
Broadcasting Share Data

- Overhead: many clk cycles to unlock execution in case of concurrent accesses
  - Eliminate overhead to access at same address → BROADCAST UNIT
  - Misalign static data and stacks to avoid accesses to the same mem bank

Convolution exec. kernels

~44% energy saving w.r.t. MIMD mode

<10% area overhead w.r.t. MIMD only cluster
Data memory Hierarchy: DMA-based, SW managed

- Tightly Coupled Data Memory
- Logarithmic Interconnect
- RISC-V cores
- DMA
- L2 Mem
An additional I/O controller for IO, off-chip Memory

PULPissimo

Ext. Mem

Mem Cont

L2 Mem

RISC-V core

I/O

interconnect

Tightly Coupled Data Memory

Mem

Mem

Mem

Mem

Mem

Mem

DMA

Logarithmic Interconnect

Event Unit

RISC-V core

RISC-V core

RISC-V core

RISC-V core

I$

I$

I$

I$
All together in VEGA: Extreme Edge IoT Processor

- RISC-V cluster (8 cores + 1)
  614GOPS/W @ 7.6GOPS (8 bit DNNs), 79GFLOPS/W @ 1GFLOP (32 bit FP appl)
- Multi-precision HWCE(4b/8b/16b)
  3×3×3 MACs with normalization / activation: 32.2GOPS and 1.3TOPS/W (8 bit)
- 1.7 μW cognitive unit for autonomous wake-up from retentive sleep mode

In cooperation with

All together in VEGA: Extreme Edge IoT Processor

- RISC-V cluster (8cores +1)
  614GOPS/W @ 7.6GOPS (8bit DNNs), 79GFLOPS/W @ 1GFLOP (32bit FP appl)
- Multi-precision HWCE(4b/8b/16b)
  3×3×3 MACs with normalization / activation: 32.2GOPS and 1.3TOPS/W (8bit)
- 1.7 µW cognitive unit for autonomous wake-up from retentive sleep mode
- **Fully-on chip DNN inference with 4MB MRAM**

<table>
<thead>
<tr>
<th>Technology</th>
<th>22nm FDSOI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chip Area</td>
<td>12mm²</td>
</tr>
<tr>
<td>SRAM</td>
<td>1.7 MB</td>
</tr>
<tr>
<td>MRAM</td>
<td>4 MB</td>
</tr>
<tr>
<td>VDD range</td>
<td>0.5V - 0.8V</td>
</tr>
<tr>
<td>VBB range</td>
<td>0V - 1.1V</td>
</tr>
<tr>
<td>Fr. Range</td>
<td>32 kHz - 450 MHz</td>
</tr>
<tr>
<td>Pow. Range</td>
<td>1.7 µW - 49.4 mW</td>
</tr>
</tbody>
</table>
Full DNN Energy (MobileNetV2)

Bandwidth [MB/s]

Energy per byte [pJ/B]

end-to-end on-chip computation 3.5x less energy

weights on MRAM

weights on HyperRAM

1.19 mJ

4.16 mJ

Eth zurich
System Scalability… How many processors?

- Performance bottleneck at the core’s boundaries (single 32-bit data port)
- Energy efficiency bounded by limited scalability of low-latency local interco
- Custom datapaths to improve throughput and efficiency.

Note: ..but at which cost? See Gianna Paulin’s Lecture
Luca Benini, Alessandro Capotondi, Alessandro Ottaviano, Alessio Burrello, Alfio Di Mauro, Andrea Borghesi, Andrea Cossettini, Andreas Kurth, Angelo Garofalo, Antonio Pullini, Arpan Prasad, Bjoern Forsberg, Corrado Bonfanti, Cristian Cioflan, Daniele Palossi, Davide Rossi, Fabio Montagna, Florian Glaser, Florian Zaruba, Francesco Conti, Georg Rutishauser, Germain Haugou, Gianna Paulin, Giuseppe Tagliavini, Hanna Müller, Luca Bertaccini, Luca Valente, Manuel Eggimann, Manuele Rusci, Marco Guermandi, Matheus Cavalcante, Matteo Perotti, Matteo Spallanzani, Michael Rogenmoser, Moritz Scherer, Moritz Schneider, Nazareno Bruschi, Nils Wistoff, Pasquale Davide Schiavone, Paul Scheffler, Philipp Mayer, Robert Balas, Samuel Riedel, Segio Mazzola, Sergei Vostrikov, Simone Benatti, Stefan Mach, Thomas Benz, Thorir Ingolfsson, Tim Fischer, Victor Javier Kartsch Morinigo, Vlad Niculescu, Xiaying Wang, Yichao Zhang, Frank K. Gürkaynak, all our past collaborators and many more that we forgot to mention.