ARA: 64-BIT RISC-V VECTOR IMPLEMENTATION IN 22NM FDSOI

Matheus Cavalcante
PhD Student
ETH Zurich

Fabian Schuiki
PhD Student
ETH Zurich

https://tmt.knect365.com/risc-v-summit
Ariane
1GHz
2 DP GFLOPS
8 GB/s

I$, D$

Instruction
Data

Interconnect

64b
64b
64b
Ariane
1GHz
2 DP GFLOPS
8 GB/s

Instruction Queue

ARA
1GHz
8 DP GFLOPS
8 GB/s

Interconnect

Instruction

64b

Data

I$, D$

I$, D$

64b

ACK/TRAP

64b

MMU

64b
Memory Bandwidth and Performance: ARA and Ariane Rooflines

- Operational Intensity: operations per byte
  - Algorithm dependent
  - One FMA = 2 operations
- Memory- and compute-boundness
- Memory bandwidth
- Number of FMAs

We want to be here!
ARA: High-performance vector processor

- Global Foundries’ 22FDX process
  - Master’s Thesis (+ a few months of ongoing PhD studies!)
  - Planning to open-source it within the PULP platform (as usual!)

- **Snapshot** of the current development
  - Challenges we faced
  - Results we achieved
  - Insights we gained
Vector processors background

- Vector processing: SIMD
  - Less instruction BW, simpler control, less energy per operation
- Packed-SIMD vs. Vector “Cray-like” SIMD
- CRAY-I (1977)

- Fujitsu A64FX
  - Based on ARM’s v8-A SVE
  - 512-bit wide packed-SIMD
  - Peak-performance at 2.7 TFLOPS
- Hwacha
  - Vector-fetch architecture
  - More complex: vector unit fetches its instructions and threads can diverge
RISC-V “V” Extension

- RISC-V “V” Extension: “Cray-like” vector-SIMD approach
- ARA: based on version 0.4-DRAFT

- No full compliance
  - No support to fixed-point and vector atomics – not our focus
  - Limited support to type promotions (e.g., 64b ← 8b + 8b) – hardware cost
    - Eventually dropped in later versions of the Extension
ARA Microarchitecture
Main datapath element: FMA Units

- We support masked FMA instructions (four operands):
  \[ \text{vmadd } vd, \text{ vsa, vsb, vsc, vmask} \]
  \[ vd[i] = \text{lsb(vmask[i]) ? vsa[i] + vsb[i] + vsc[i] : 0}; \]

- The four lanes operate in lockstep
  - Low control overhead

- FMA is pipelined (5 cycles) to meet \( f_{\text{min}} \) constraint

- Each lane gets 64b operands from four 256b input FIFO buffers (A, B, C, VMASK)
  - Number of lanes determines buffer width
Operand FIFO queues

- One input FIFO buffer provides one operand to all the (four) lanes
  - 256b (4x64b) wide entries
  - One FIFO buffer per operand per multi-lane datapath unit → 10 FIFO buffers

- Output FIFO buffers for output operands, one per multi-lane datapath unit

- Needed to sustain maximum throughput for the lock-step operation of the FUs, while hiding the latency caused by banking conflicts in the VRF (next slides)
Vector register file

256b banks $\rightarrow$ one bank stores 4 operands consumed in parallel by the 4 lanes
8 banks $\rightarrow$ BF of 1,6 for the worst-case read BW (FMA is 4R+1W/cycle)
Vector Register File and Operand-Deliver Interconnect

- All-to-all input log-interconnect
  - 256b wide (64bx4)
  - 8-source (VRF banks) x 10-dest (FIFO buffers)
  - Registered boundaries (for timing)
- All-to-all output log-interconnect
  - 256b wide (64bx4)
  - 4-source (out FIFO buffers) x 8 dest (RF banks)
- Fixed-priority arbiter
  - $P_M > P_A > P_B > P_C$
  - VRF is built as 1RW SRAM banks
  - Writes have lower priority than reads – unless output queue is full
Execution of a FMA instruction

- Consider the execution of the following instruction
  \[
  \text{vmadd } vd, \text{ vsa, vsb, vsc, vsmask}
  \]

- Worst case in terms of banking factor
  - 4 reads + 1 write per cycle
  - Banking factor = 1.75

- We take a vector of length 256 (ideally 64 cycles to run)
Execution of a FMA instruction

The first 4 elements of all 4 operands are in Bank 0. 3 access stalled due to banking conflicts.
Execution of a FMA instruction

Cycle count: 2
FMA Utilization: 0/2 = 0%

<table>
<thead>
<tr>
<th>Bank 7</th>
<th>Bank 6</th>
<th>Bank 5</th>
<th>Bank 4</th>
<th>Bank 3</th>
<th>Bank 2</th>
<th>Bank 1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mask</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
</tr>
<tr>
<td>B</td>
</tr>
<tr>
<td>C</td>
</tr>
</tbody>
</table>

FMA

Operand Request

VRF Priority Arbiter

M A B C

1 0 0 0
Execution of a FMA instruction

Cycle count: 3
FMA Utilization: 0/3 = 0%

Operand Request

VRF Priority Arbiter

FMA

Bank 7
Bank 6
Bank 5
Bank 4
Bank 3
Bank 2
Bank 1
Bank 0

Mask

A
B
C

Cycle count: 3
FMA Utilization: 0/3 = 0%
Execution of a FMA instruction

Cycle count: 7
FMA Utilization: 1/7 = 14%
Execution of a FMA instruction

Cycle count: 8
FMA Utilization: 2/8 = 25%
Execution of a FMA instruction

Cycle count: 12
FMA Utilization: 6/12 = 50%
Execution of a FMA instruction

Cycle count: 13
FMA Utilization: 7/13 = 54%
Execution of a FMA instruction

Cycle count: 14
FMA Utilization: 8/14 = 57%
Execution of a FMA instruction

Cycle count: 15
FMA Utilization: 9/15 = 60%

Operand Request
M A B C

VRF Priority Arbiter

Cycle count: 15
FMA Utilization: 9/15 = 60%

Bank 7
Bank 6
Bank 5
Bank 4
Bank 3
Bank 2
Bank 1
Bank 0

Mask
A
B
C

FMA

6 5 4 3

2
Execution of a FMA instruction

Cycle count: 69
FMA Utilization: 63/69 = 91%
Execution of a FMA instruction

Cycle count: 70
FMA Utilization: 64/70 = 91%
Hardware Support for Vector Reductions

- Triggered by VMADD instruction with scalar result register
- Executed on FMA units by feeding results back in as operand C
- E.g. Reduction of 64-element vector:
- Avg. utilization in this case 36% \(\frac{N}{N+4\cdot29}, \ N = 64\)

\[ y = \sum_{i} A_i \cdot B_i \]

Lane 1
Lane 2
Lane 3
Lane 4

Accumulation Phase
16 cycles
20 partial sums (5 latency x 4 lanes)

Merge Results in each Lane
17 cycles
4 partial sums (4 lanes)

Merge Lanes
12 cycles
1 final sum

Final result written to regfile

Matheus Cavalcante and Fabian Schuiki  12.4.2018  25
Ways to Improve Reductions

- Current Implementation (constant 29 cycle tail):
  \[ \frac{N}{N + 4.29} \] (36% for 64-element vector)

- Future Improvement A:
  - Schedule FPU operations of next instruction in gaps of the reduction
  - Utilization improves to \( \frac{N}{N + 19} \) (77% for 64-element vector)

- Future Improvement B:
  - Add separate reduction adder
  - Utilization improves up to 100%, since reduction tail not eating away performance
Benchmarks
ARA and Ariane – Peak performance

Performances we must achieve!

ATTAINABLE PERFORMANCE

Operational Intensity (OP/B)

Performance (OP/Cycle)
Benchmarks

- Can we achieve 8 GFLOPs peak performance?
  - Upper-bound: four FMAs working at 100%

- Three key kernels:
  - Multiply-add (DAXPY): heavily memory-bound
  - Convolution (DCONV): compute-bound
  - Matrix-multiplication (DGEMM): compute-bound

- Cycle-accurate simulation from the RTL
  - We ignore the initial set-up cycles (around 40 cycles)
  - Startup, instruction fetch, decoding, vector unit configuration…
DAXPY: $Y \leftarrow aX + Y$

- Strip-mined loop over the $n$ elements of vectors $x$ and $y$

- Memory-bound
  - We'll be far from 8 GFLOPs!
  - But are we close to the performance limit?

```c
// Read scalar a
vins va, a, zero;

while (n != 0) {
    // Stripmined loop
    size_t vl = setvl(n);

    // Read x and y
    vld vx, x;
    vld vy, y;

    // vy = va . vx + vy
    vmadd vy, va, vx, vy;

    // Store y
    vst vy, y;

    // Bump pointers
    x += vl; y += vl; n -= vl;
}
```
DAXPY: Performance

<table>
<thead>
<tr>
<th>Vector Length</th>
<th>FPU Utilization (%)</th>
<th>Performance (op/cycle)</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>5.6%</td>
<td>0.45</td>
</tr>
<tr>
<td>64</td>
<td>6.6%</td>
<td>0.53</td>
</tr>
<tr>
<td>128</td>
<td>7.3%</td>
<td>0.59</td>
</tr>
<tr>
<td>256</td>
<td>7.7%</td>
<td>0.62</td>
</tr>
<tr>
<td>512</td>
<td>8.0%</td>
<td>0.64</td>
</tr>
<tr>
<td>1024</td>
<td>8.1%</td>
<td>0.65</td>
</tr>
</tbody>
</table>

- We achieve what we could in terms of perf
  - Can’t expect 8 GFLOPs from a memory-bound kernel
  - Ops/cycle grows to 8 if we increase memory port width (e.g. 128b → 2x perf)
**DCONV:** \( Y = K \ast X \)

- Kernel particular for CNNs
  - Convolution kernel size: 7 channels, each \( 3 \times 3 \)
  - Image size: 7 channels, each \( n \times 1 \)

- Operational intensity
  - \( 3 \times 3 \times 8 \times 7n = 504n \) bytes of memory transfers
  - \( 882n \) operations (multiply-adds)
  - 1,75 operations per byte

- Compute-bound kernel
  - It should be possible to achieve 8 ops/cycle
  - Scheduling is key
DCONV: Performance

<table>
<thead>
<tr>
<th>Vector Length</th>
<th>FPU Utilization (%)</th>
<th>Performance (op/cycle)</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>19.8%</td>
<td>1.58</td>
</tr>
<tr>
<td>64</td>
<td>36.1%</td>
<td>2.89</td>
</tr>
<tr>
<td>128</td>
<td>61.5%</td>
<td>4.92</td>
</tr>
<tr>
<td>256</td>
<td>82.1%</td>
<td>6.57</td>
</tr>
<tr>
<td>512</td>
<td>82.3%</td>
<td>6.59</td>
</tr>
<tr>
<td>1024</td>
<td>82.5%</td>
<td>6.60</td>
</tr>
</tbody>
</table>

- Initial banking conflicts limit performance
- Performance goes up until strip-mining loop comes to play
  - Unroll strip-mining: programmability?
  - Hard to hide all the memory transfers (initial loads and final stores)
DGEMM: $C \leftarrow \alpha AB + \beta C$

- BLAS-3 routine
  - Common kernel in several applications

- High data reuse
  - When the kernel is compute-bound, it should be possible to achieve 8 ops/cycle

- Operational intensity
  - $8 \times 3n^2 = 24n^2$ bytes of memory transfers
  - $2n^3$ operations (multiply-adds)
  - $\frac{n}{12}$ operations per byte
  - If $n \leq 12$, kernel is memory-bound by ARA's VLSU unit
DGEMM: Performance

<table>
<thead>
<tr>
<th>Vector Length</th>
<th>FPU Utilization (%)</th>
<th>Performance (op/cycle)</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>19.2%</td>
<td>1.54</td>
</tr>
<tr>
<td>64</td>
<td>37.8%</td>
<td>3.02</td>
</tr>
<tr>
<td>128</td>
<td>70.3%</td>
<td>5.62</td>
</tr>
<tr>
<td>256</td>
<td>84.7%</td>
<td>6.77</td>
</tr>
<tr>
<td>512</td>
<td>85.5%</td>
<td>6.84</td>
</tr>
<tr>
<td>1024</td>
<td>86.3%</td>
<td>6.91</td>
</tr>
</tbody>
</table>

- We see the same phenomena seen with DCONV
  - Initial banking conflicts limiting performance with shorter vectors
  - Strip-mining and unmaskable memory transfers limiting steady performance
So.. can we achieve 8 GFLOPs?
Implementation results
ARA: GF FDX22 1GHz implementation (SS, 0.72V, 125 ºC)

Critical path
45 gates
ARA is $1.8 \times$ bigger than Ariane…
and has $4 \times$ its computational power

Operation density:
- Ariane: 7,27 GFLOPS/mm$^2$
- ARA: 16,23 GFLOPS/mm$^2$
Conclusions
Shuffling instructions

- Higher operational intensity → minimize data transfers
  - By shuffling and reordering data inside vector registers
- Only two* instructions available
  - \texttt{vslide}: \(vd[i] = vs1[i + rs2]\)
  - \texttt{vrgather}: \(vd[i] = vs1[vs2[i]]\)
- Register-gather is too general → hard to optimize!
- Dedicated instructions to more specific shuffling: permutations, rotations?

*(three, more recently, as \texttt{vslide} was split into \texttt{vslideup} and \texttt{vslidedown})*
Decoupling between scalar and vector units

- We did benefit from decoupling the scalar and the vector unit

- Different “worlds”
  - Scalar unit: speculative, one instruction issued per cycle, several in-flight instructions
  - Vector unit: non-speculative, latency-tolerant, high throughput, a few in-flight vector instructions

- We see with apprehension ISA decisions that push towards their recoupling
  - E.g., the recent decision of mapping the vector registers over the floating-point registers
ARA supports mixed-precision to a certain extent

Previous versions allowed for a mixed-precision instruction as $64b \leftarrow 8b + 8b$
- $8b$, $16b$ and $32b$ operands could be promoted to $64b$ operands
- High hardware cost!

We now allow for a more restricted set of type promotions
- $8 \rightarrow 16b$, $16 \rightarrow 32b$ and $32 \rightarrow 64b$
- Aligned with newer revisions of the V Extension
Wrapping up...

- ARA: 64-bit vector processor
ARA: 64-BIT RISC-V VECTOR IMPLEMENTATION IN 22NM FDSOI

Matheus Cavalcante  
PhD Student  
ETH Zurich

Fabian Schuiki  
PhD Student  
ETH Zurich

https://tmt.knect365.com/risc-v-summit