Scaling an Augmented RISC-V Processor Design with High-Level Synthesis

Fourth International workshop on RISC-V for HPC

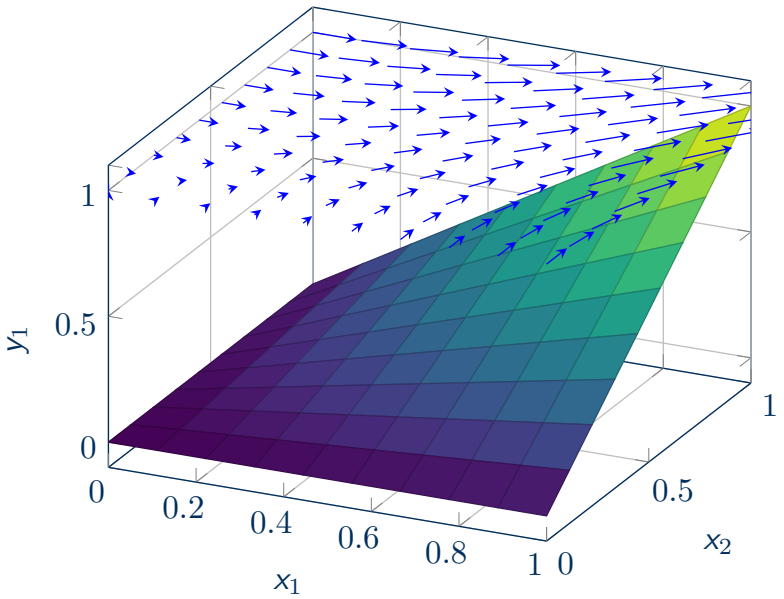
Johannes Schoder, Martin Bücker

Friedrich Schiller University Jena, Department of Computer Science

Hamburg, 16th May 2024

Motivation

What we want in the long run:



$$y = f(x_1, x_2) = x_1 \cdot x_2$$

$$\nabla f(x_1, x_2) = \nabla y = \begin{pmatrix} \frac{\partial y}{\partial x_1} \\ \frac{\partial y}{\partial x_2} \end{pmatrix} = \begin{pmatrix} x_2 \\ x_1 \end{pmatrix}$$

Accelerate and Embed Automatic Differentiation into a Domain Specific RISC-V Processor. Work in progress..

We want: The gradient of a differentiable function without having to work for it.

- RISC-V - open source, expandable ISA
- HLS - High-Level Synthesis: fast prototyping
- FPGA - scalable and reconfigurable
- AD - Automatic Differentiation - program transformation in hardware

Why do we want that?

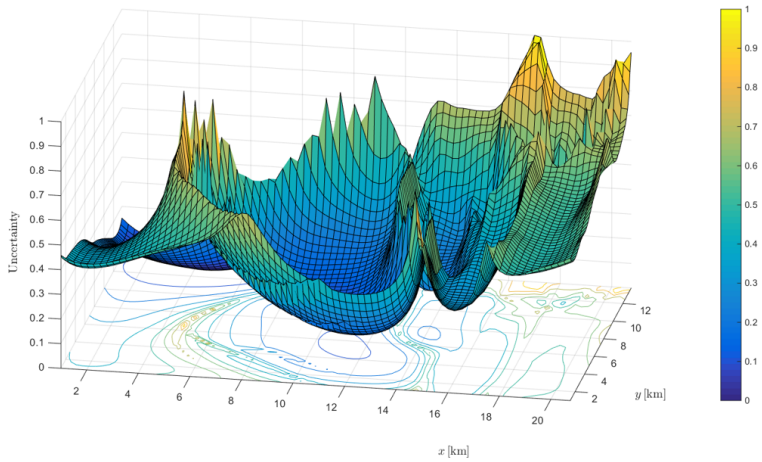


Figure: MeProRisk II: Optimization strategies and risk analysis for deep geothermal reservoirs

`https://www.ac.uni-jena.de/research/project/meprorisk`

We need the information about the gradient for, e.g.,

- Simulations
  - solve ODEs
  - solve nonlinear systems (newton method)
- Optimization
  - parameter estimation
  - minimize cost functions
  - train neural networks
- Examples:
  - physics simulation (shape optimization)

Automatic differentiation

- Typically done with software tools
- Two distinct modes:
  - ”reverse mode automatic differentiation (AD)”
  - ”forward mode AD”
- This study focuses only on forward mode AD

Example $f: \mathbb{R}^3 \rightarrow \mathbb{R}^2$:

$$y_1(x_1, x_2, x_3) = \frac{\sin(x_1 \cdot x_2)}{\exp(x_2 - x_3)}$$

$$y_2(x_2, x_3) = \exp(x_2 - x_3) - 2 \cdot x_3$$

- Forward mode: $f: \mathbb{R}^n \rightarrow \mathbb{R}^m$ preferred if $n \ll m$
- Reverse mode: $f: \mathbb{R}^n \rightarrow \mathbb{R}^m$ preferred if $n \gg m$

Forward Mode AD

C Code

$$y_1 = f(x_1, x_2) = (a - x_1)^2 + b(x_2 - x_1^2)^2$$

```c
int Rosenbrock (int x_1, int x_2){
    int a = 1;
    int b = 10;
    int result=((a-x_1)*(a-x_1))+(b*(x_2-x_1*x_1)*(x_2-x_1*x_1));
   return result;
}
```

Listing 1: C code example for the Rosenbrock function

ASM Code

```
1  Rosenbrock: #a0 = x_1; a1 = x_2 (procedure call standard)
2  li      a2, 1       #a2 = a
3  sub     a2, a2, a0  #a2 = a-x_1
4  mul     a2, a2, a2  #a2 = (a-x_1)^2
5  mul     a0, a0, a0  #a0 = x_1^2
6  sub     a0, a1, a0  #a0 = (x_2-x_1^2)
7  mul     a0, a0, a0  #a0 = (x_2-x_1^2)^2
8  li      a1, 10      #a1 = b
9  mul     a0, a0, a1  #a0 = b*(x_2-x_1^2)^2
10 add     a0, a0, a2  #a0 = (a-x_1)^2+b*(x_2-x_1^2)^2
11 ret                 #return a0 (procedure call standard)
```

Listing 2: RV32IM assembly code example for the Rosenbrock function

Step 1: Decomposing complex mathematical functions in ”elementary functions”
- (SUB, ADD, MUL)

Extended Evaluation Procedure

$$
\begin{array}{l|l}
v_{-1} = x_1 & \overrightarrow{\Delta v_{-1}} = \overrightarrow{\Delta x_1} \\
v_0 = x_2 & \overrightarrow{\Delta v_0} = \overrightarrow{\Delta x_2} \\
\hline
v_1 = a - v_{-1} & \overrightarrow{\Delta v_1} = \overrightarrow{\Delta v_{-1}} \cdot \frac{\partial v_1}{\partial v_{-1}} = -\overrightarrow{\Delta v_{-1}} \\
v_2 = v_1 \cdot v_1 = {v_1}^2 & \overrightarrow{\Delta v_2} = \overrightarrow{\Delta v_1} \cdot \frac{\partial v_2}{\partial v_1} = 2 \cdot \overrightarrow{\Delta v_1} \cdot v_1 \\
v_3 = v_{-1} \cdot v_{-1} = {v_{-1}}^2 & \overrightarrow{\Delta v_3} = \overrightarrow{\Delta v_{-1}} \cdot \frac{\partial v_3}{\partial v_{-1}} = 2 \cdot \overrightarrow{\Delta v_{-1}} \cdot v_{-1} \\
v_4 = v_0 - v_3 & \overrightarrow{\Delta v_4} = \overrightarrow{\Delta v_0} \cdot \frac{\partial v_4}{\partial v_0} + \overrightarrow{\Delta v_3} \cdot \frac{\partial v_4}{\partial v_3} = \overrightarrow{\Delta v_0} - \overrightarrow{\Delta v_3} \\
v_5 = v_4 \cdot v_4 = {v_4}^2 & \overrightarrow{\Delta v_5} = \overrightarrow{\Delta v_4} \cdot \frac{\partial v_5}{\partial v_4} = 2 \cdot \overrightarrow{\Delta v_4} \cdot v_4 \\
v_6 = b \cdot v_5 & \overrightarrow{\Delta v_6} = \overrightarrow{\Delta v_5} \cdot \frac{\partial v_6}{\partial v_5} = b \cdot \overrightarrow{\Delta v_5} \\
v_7 = v_2 + v_6 & \overrightarrow{\Delta v_7} = \overrightarrow{\Delta v_2} \cdot \frac{\partial v_7}{\partial v_2} + \overrightarrow{\Delta v_6} \cdot \frac{\partial v_7}{\partial v_6} = \overrightarrow{\Delta v_2} + \overrightarrow{\Delta v_6} \\
\hline
y_1 = v_7 & \overrightarrow{\Delta y_1} = \overrightarrow{\Delta v_7}
\end{array}
\tag{1}
$$

Extended Evaluation Procedure

$$
\begin{aligned}
v_{i-n} &= x_i && i = 1 \ldots n \\
\overrightarrow{\Delta v_{i-n}} &= \overrightarrow{\Delta x_i} && i = 1 \ldots n \\
\hline
v_i &= \phi_i(v_j)_{j \prec i} && i = 1 \ldots l \\
\overrightarrow{\Delta v_i} &= \sum_{j \prec i} \frac{\partial}{\partial v_j} \phi_i(v_j) \cdot \overrightarrow{\Delta v_j} && i = 1 \ldots l \\
\hline
y_{m-i} &= v_{l-i} && i = m-1 \ldots 0 \\
\overrightarrow{\Delta y_{m-i}} &= \overrightarrow{\Delta v_{l-i}} && i = m-1 \ldots 0
\end{aligned}
\tag{2}
$$

Automatic Differentiation [1]:

[1] Andreas Griewank and Andrea Walther: Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation, https://doi.org/10.1137/1.9780898717761

Extended Evaluation Procedure

$$
\begin{array}{r|l}
v_{-1} = x_1 = 0 & \overrightarrow{\Delta v_{-1}} = \overrightarrow{\Delta x_1} = 1 \\
v_0 = x_2 = -0.5 & \overrightarrow{\Delta v_0} = \overrightarrow{\Delta x_2} = 0 \\
\hline
\dots & \dots \\
\hline
y_1 = 3.5 & \overrightarrow{\Delta y_1} = -2
\end{array}
\tag{3}
$$

with $\overrightarrow{\Delta x_1}, \overrightarrow{\Delta x_2} \in \mathbb{R}^1$

results in a single scalar entry of the gradient: $\overrightarrow{\Delta y_1} = \frac{\partial y_1}{\partial x_1}$

$\overrightarrow{\Delta y_1} = \frac{\partial y_1}{\partial x_1} = -2$ partial derivative of y_1 wrt. x_1

– because x_1, x_2, $\overrightarrow{\Delta x_1}$, and $\overrightarrow{\Delta x_2}$ are independent of each other, and these values aren't overwritten in the extended evaluation procedure, we can also say, that $\overrightarrow{\Delta y_1}$ contains the total derivative of y_1 wrt. x_1; $\frac{dy_1}{dx_1}$

Seed matrix S

$$
\begin{array}{l|l}
v_{-1} = x_1 = 0 & \overrightarrow{\Delta v_{-1}} = \overrightarrow{\Delta x_1} = \begin{pmatrix} 1 & 0 \end{pmatrix} \\
v_0 = x_2 = -0.5 & \overrightarrow{\Delta v_0} = \overrightarrow{\Delta x_2} = \begin{pmatrix} 0 & 1 \end{pmatrix} \\
\hline
\dots & \dots \\
\hline
y_1 = 3.5 & \overrightarrow{\Delta y_1} = \begin{pmatrix} -2 & -10 \end{pmatrix}
\end{array}
\tag{4}
$$

results in $\overrightarrow{\Delta y_1} = \left(\frac{\partial y_1}{\partial x_1} \quad \frac{\partial y_1}{\partial x_2} \right) \Big|_{0,-0.5}$

$p = 2$

Seeding - Gradient

Seeding $S \in \mathbb{R}^{(n \times p)}$ with $p = n$, and $S = I_n$ (Identity)

$$S = \begin{pmatrix} \overrightarrow{\Delta x_1}(1) & \overrightarrow{\Delta x_1}(2) & \dots & \overrightarrow{\Delta x_1}(p) \\ \overrightarrow{\Delta x_2}(1) & \overrightarrow{\Delta x_2}(2) & \dots & \overrightarrow{\Delta x_2}(p) \\ \vdots & & \ddots & \vdots \\ \overrightarrow{\Delta x_n}(1) & \dots & \dots & \overrightarrow{\Delta x_n}(p) \end{pmatrix} \Bigg| \quad S = \begin{pmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & & \ddots & \vdots \\ 0 & \dots & \dots & 1 \end{pmatrix} \in \mathbb{R}^{(n \times n)} \tag{5}$$

will result in the gradient at a specific position:

$$\overrightarrow{\Delta y_1}\Big|_{(x_1, x_2, \dots, x_n)} = \nabla f(x_1, x_2, \dots, x_n) = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} & \dots & \frac{\partial y_1}{\partial x_n} \end{pmatrix}\Big|_{(x_1, x_2, \dots, x_n)}$$

HLS Processor Design

Starting Point

Reminder: what we want?

We want information on the gradient for free by performing the program transformation in hardware

How can we get it?

Using an augmented RISC-V processor design based on existing RISC-V processor designs written for HLS.

Thanks to the DALI-LIRMM team and Prof. Goossens[2]

[2] Bernard Goossens - Guide to computer processor architecture : a RISC-V approach, with high-level synthesis, https://doi.org/10.1007/978-3-031-18023-1

$\overline{\text{AD}}$ operations RV32I ISA - Changes to ALU

RISC-V Inst.	RISC-V ISA	Arithmetic Operation $v_i = \phi_i(v_j)_{j \prec i}$	Simultaneous Transformed Operation $\overrightarrow{\Delta v_i} = \sum_{j \prec i} \frac{\partial}{\partial v_j} \phi_i(v_j) \cdot \overrightarrow{\Delta v_j}$
`ADD`, `ADDI`	I	$v_i = v_{j_1} + v_{j_2}$	$\overrightarrow{\Delta v_i} = \overrightarrow{\Delta v_{j_1}} + \overrightarrow{\Delta v_{j_2}}$
`SUB`, `SUBI`	I	$v_i = v_{j_1} - v_{j_2}$	$\overrightarrow{\Delta v_i} = \overrightarrow{\Delta v_{j_1}} - \overrightarrow{\Delta v_{j_2}}$
`SLL`, `SLLI`	I	$v_i = v_{j_1} \cdot 2^{v_{j_2}}$	$\overrightarrow{\Delta v_i} = \overrightarrow{\Delta v_{j_1}} \cdot 2^{v_{j_2}} + \overrightarrow{\Delta v_{j_2}} \cdot v_{j_1} \cdot 2^{v_{j_2}} \cdot \log_e 2$
`SRA`, `SRAI`, (`SRL`, `SRLI`)	I	$v_i = \frac{v_{j_1}}{2^{v_{j_2}}}$	$\overrightarrow{\Delta v_i} = \overrightarrow{\Delta v_{j_1}} \cdot \frac{1}{2^{v_{j_2}}} - \overrightarrow{\Delta v_{j_2}} \cdot v_{j_1} \cdot \frac{1}{(2^{v_{j_2}})} \cdot \log_e 2$

Augmented Processor Design

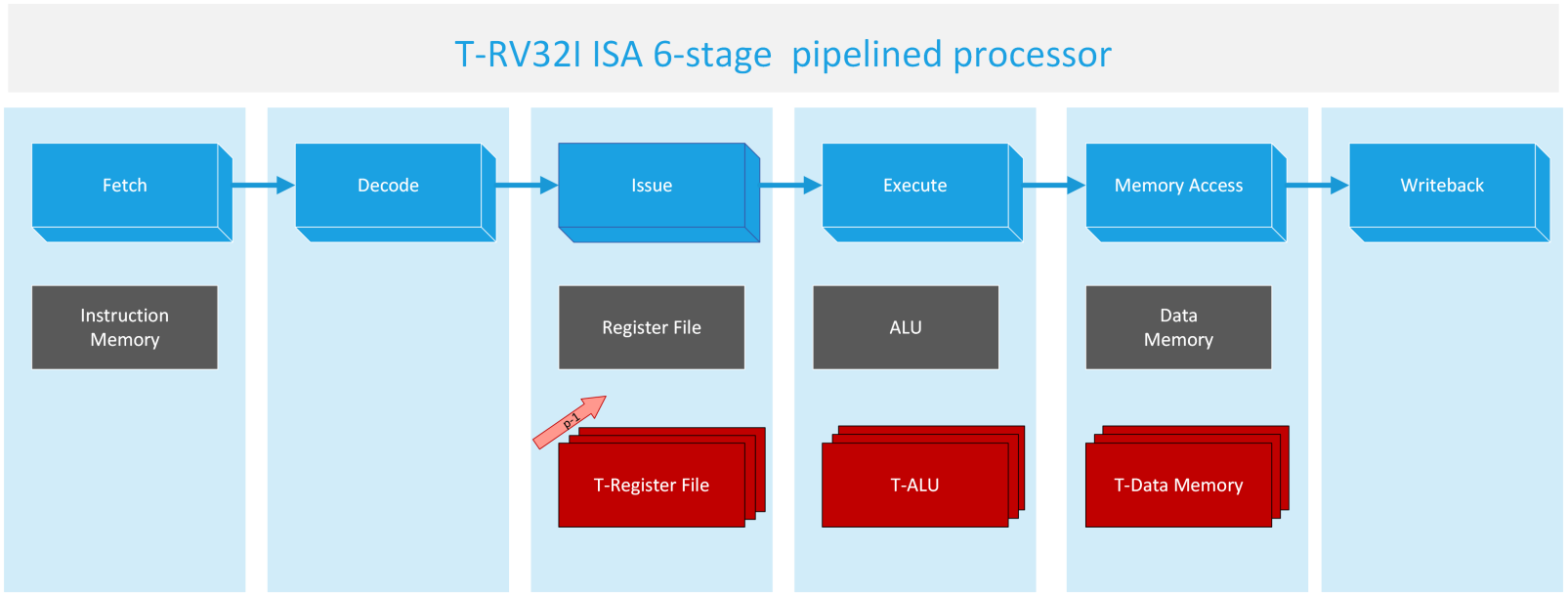
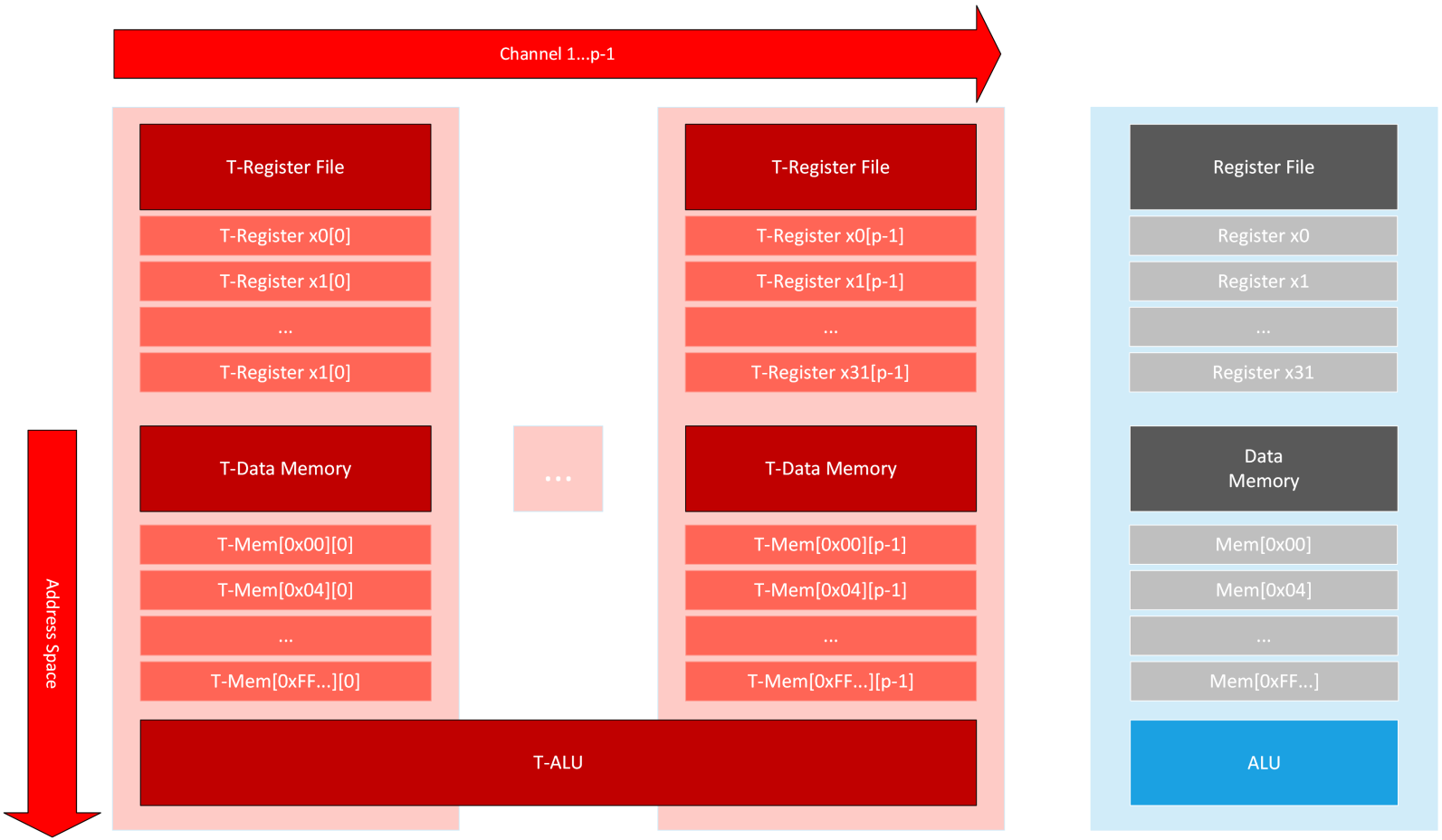


Figure: Augmented Processor Design

Memory Layout - "Channels"



Custom instructions

Added custom RV32-Instructions for:

1. Setting seed in Memory `TSW`
2. Loading gradient information from AD-Memory to Register `TLW`
3. Copying gradient information from AD-Register to Register `TMV`

Alveo U50DD

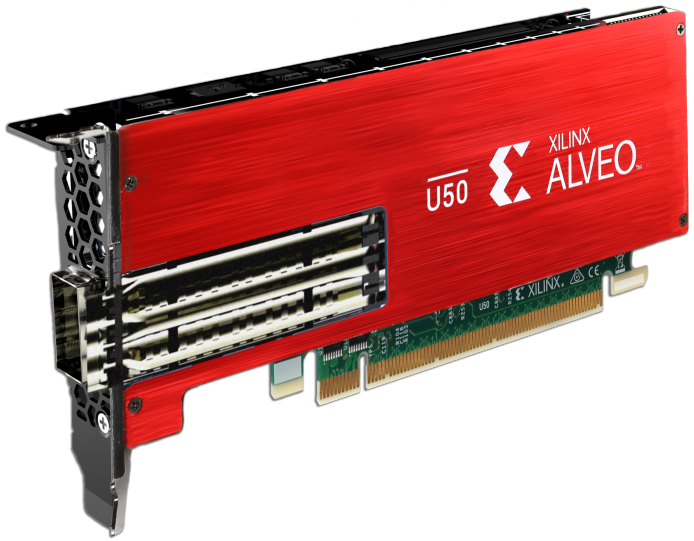


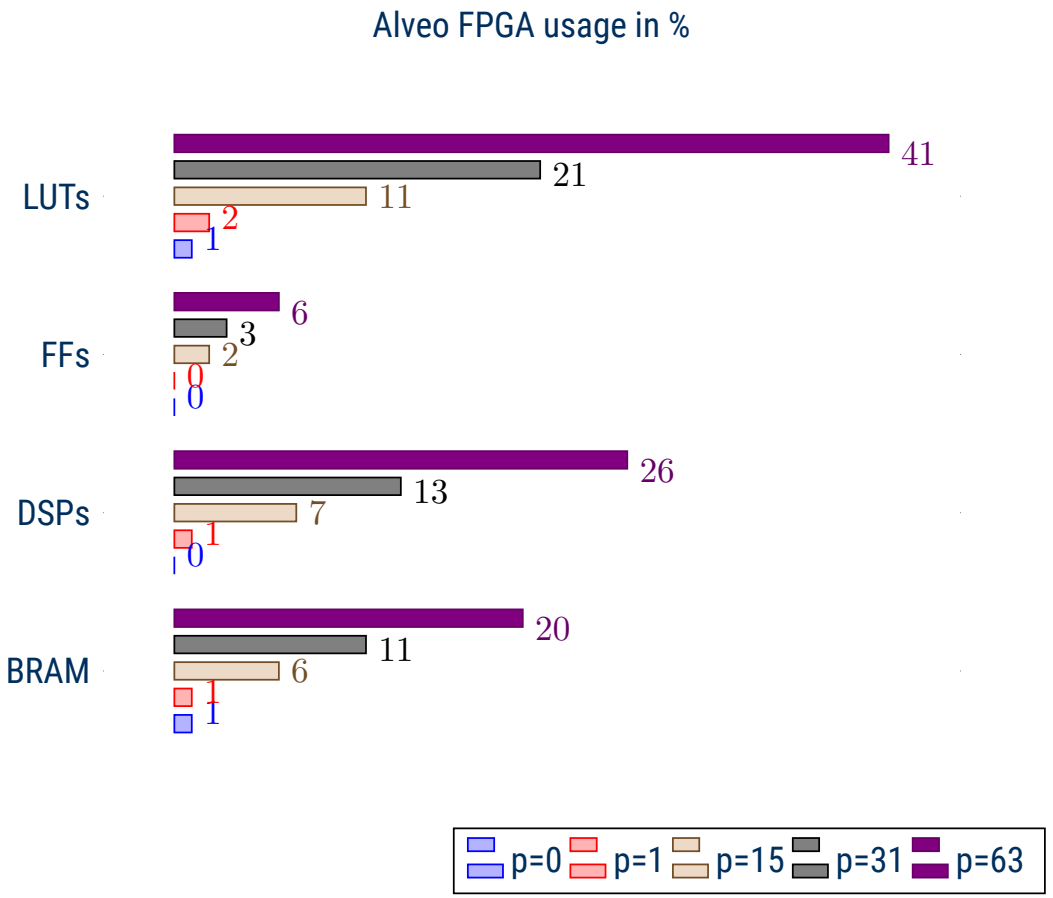
Figure: Alveo U50 [a]

UltraScale+™ FPGA

- 8 GB of high-bandwidth memory (HBM2)
- LUTs: 872K
- FFs (Registers): 1,743K
- DSPs: 5,952
- BRAM (36Kb): 1344 (47.3 Mb)
- UltraRAM (288 Kb): 640 (180.0 Mb)

[a] https://www.xilinx.com/products/boards-and-kits/alveo/u50.html

HLS (T-RV32I) for Alveo U50DD



Memory size:

- I-RAM: 2^{13} Bytes = 8KB
- D-RAM: 2^{14} Bytes = 16KB (+16KB*p)

Estimated Clock:

- 0: 2.97ns
- 1: 8.677ns
- 15: 11.626ns
- 31: 14.481ns
- 63: 14.590ns

Code size reduction:

unaugmented RISC-V processor

```
#ad_v2_0 = ad_v1_0 + ad_v0_0;
lw   a4,-44(s0)
lw   a5,-36(s0)
add  a5,a4,a5
sw   a5,-52(s0)
#ad_v2_1 = ad_v1_1 + ad_v0_1;
lw   a4,-48(s0)
lw   a5,-40(s0)
add  a5,a4,a5
sw   a5,-56(s0)
#v2 = v1 + v0;
lw   a4,-28(s0)
lw   a5,-24(s0)
add  a5,a4,a5
sw   a5,-32(s0)
```

augmented RISC-V processor

```
#v2 = v1 + v0;
lw   a4,-32(s0)
lw   a5,-28(s0)
add  a5,a4,a5
sw   a5,-36(s0)
```

Speedup for one arithmetic operation

$$T^{\text{orig}}(p) = (c \cdot p + 1) \cdot \text{CPI} \cdot {T_c}^{\text{orig}}$$

$$T^{\text{aug}}(p) = 1 \cdot \text{CPI} \cdot {T_c}^{\text{aug}}(p)$$

$$S(p) := \frac{T^{\text{orig}}(p)}{T^{\text{aug}}(p)} = (c \cdot p + 1) \cdot \frac{{T_c}^{\text{orig}}}{{T_c}^{\text{aug}}(p)}$$

$$S(63) \approx (c \cdot 63 + 1) \cdot \frac{2.97ns}{14.590ns} \approx (c \cdot 63 + 1) \cdot 0.2 \geq 12.8$$

p = channels, c = software "overhead" of non-augmented design,
${T_c}^{\text{orig}}$ = clock cycle time of the original processor design,
${T_c}^{\text{aug}}(p)$ = clock cycle time of the augmented processor design with p channels,
CPI = equals the clock cycles per instruction

$\overline{\text{AD}}$ operations RV32I ISA - Changes to ALU

RISC-V Inst.	RISC-V ISA	Arithmetic Operation $v_i = \phi_i(v_j)_{j \prec i}$	Simultaneous AD-Operation $\overrightarrow{\Delta v_i} = \sum_{j \prec i} \frac{\partial}{\partial v_j} \phi_i(v_j) \cdot \overrightarrow{\Delta v_j}$
`ADD`, `ADDI`	I	$v_i = v_{j_1} + v_{j_2}$	$\overrightarrow{\Delta v_i} = \overrightarrow{\Delta v_{j_1}} + \overrightarrow{\Delta v_{j_2}}$
`SUB`, `SUBI`	I	$v_i = v_{j_1} - v_{j_2}$	$\overrightarrow{\Delta v_i} = \overrightarrow{\Delta v_{j_1}} - \overrightarrow{\Delta v_{j_2}}$
`SLL`, `SLLI`	I	$v_i = v_{j_1} \cdot 2^{v_{j_2}}$	$\overrightarrow{\Delta v_i} = \overrightarrow{\Delta v_{j_1}} \cdot 2^{v_{j_2}} + \overrightarrow{\Delta v_{j_2}} \cdot v_{j_1} \cdot 2^{v_{j_2}} \cdot \log_e 2$
`SRA`, `SRAI`, (`SRL`, `SRLI`)	I	$v_i = \frac{v_{j_1}}{2^{v_{j_2}}}$	$\overrightarrow{\Delta v_i} = \overrightarrow{\Delta v_{j_1}} \cdot \frac{1}{2^{v_{j_2}}} - \overrightarrow{\Delta v_{j_2}} \cdot v_{j_1} \cdot \frac{1}{(2^{v_{j_2}})} \cdot \log_e 2$

Speedup

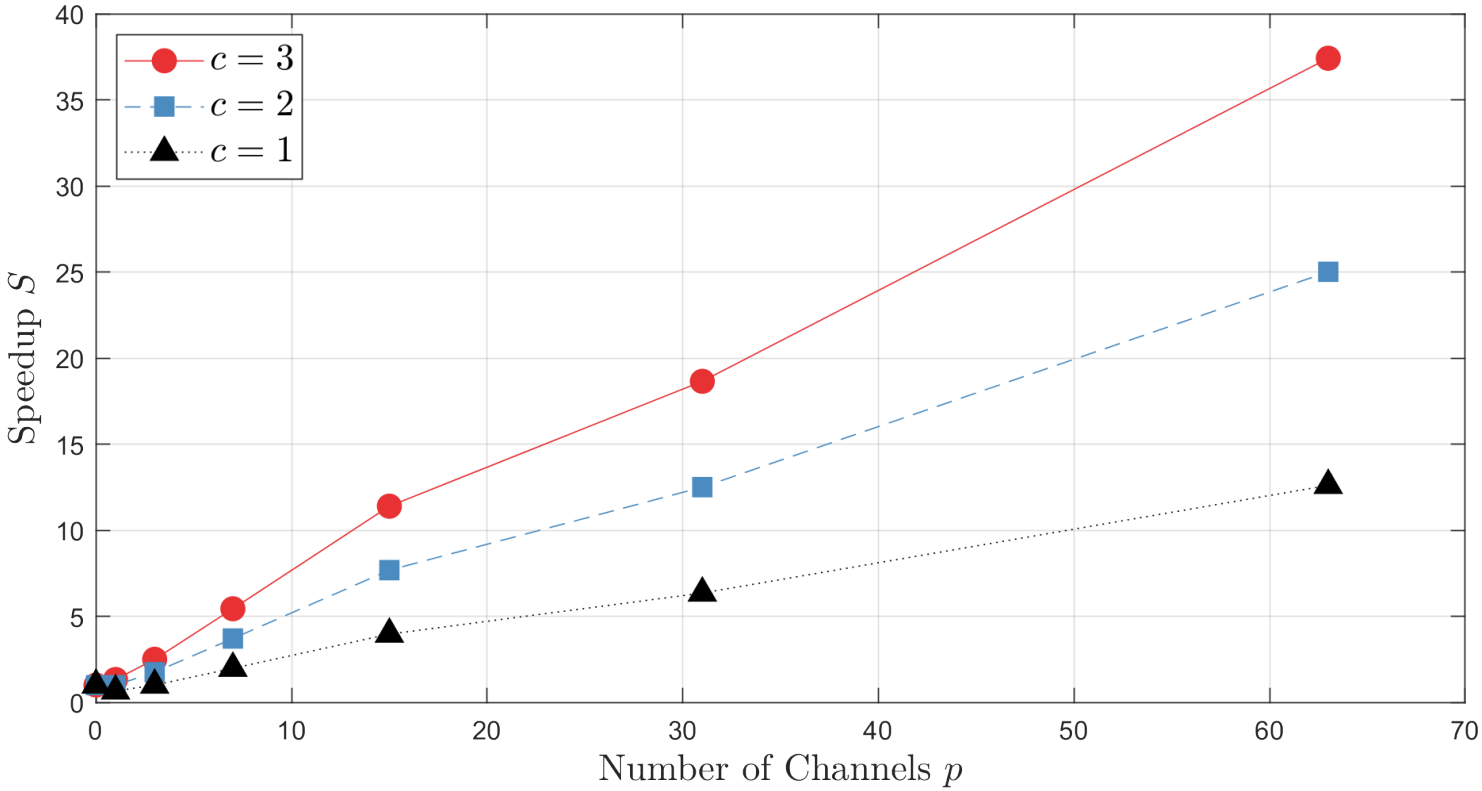


Figure: Exptected Speedup

What works quite well

Advantages

1. ”Easy” to use, user doesn’t have to think much about AD
2. AD adaptions don’t alter the ”normal” RISC-V instructions/ operations
3. Performance looks promising on paper
4. Scalable/ configurable design (FPGA)

Next steps:

Todo

1. Extend to T-RV32IMF (Mostly done)
2. Memory Hierarchy; so far BRAM instead of DRAM
   - Quantify effects on cache misses?
   - Applicability to larger workloads?
3. Ideally: Extend to T-RV32IMFV

Limitations

Todo

1. Clock frequency reduced
2. Processor size increased (FPGA usage)
3. Needs p- times more memory by design
4. Limited amount of arithmetic operations of the ISA
5. Useful larger applications?
6. Extensive testing needed
7. Usability of the prototype
8. Bit-Tricks

Thanks!