TU Graz
SCIENCE
PASSION
TECHNOLOGY

Accelerating Hash-Based Polynomial Commitment Schemes with Linear Prover Time

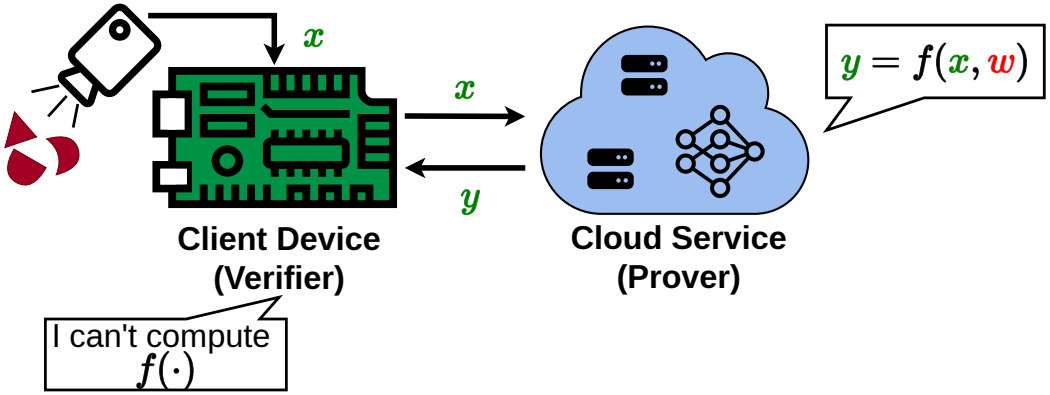
Florian Hirner **Florian Krieger** Constantin Piber Sujoy Sinha Roy

Graz University of Technology

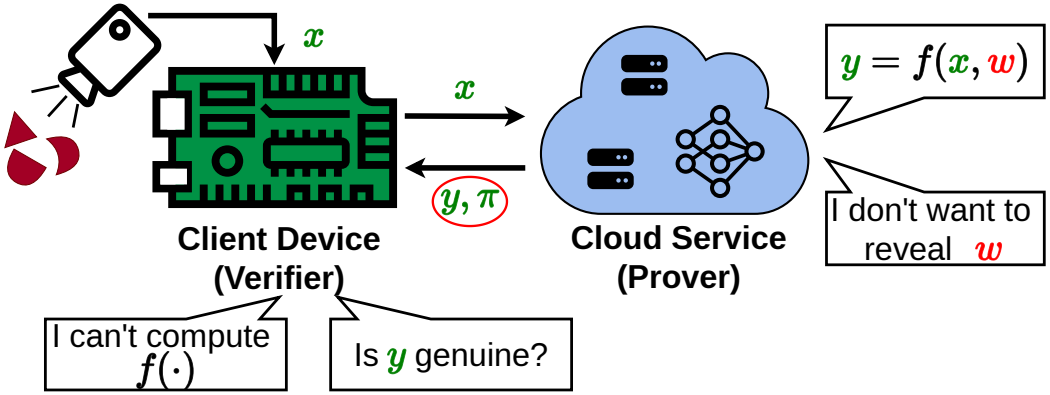
CHES 2025 Conference

› isec.tugraz.at

Verifiable Cloud Computing



Verifiable Cloud Computing



Polynomial Commitment Schemes

Polynomial Commitment Schemes (PCS)

- Central primitives in proof systems
- Prove genuine evaluation of a polynomial
  - $f(x) = c_0x^0 + c_1x^1 + c_2x^2 + \ldots + c_{n-1}x^{n-1}$

Polynomial Commitment Schemes (PCS)

- Central primitives in proof systems
- Prove genuine evaluation of a polynomial
  - $f(x) = c_0x^0 + c_1x^1 + c_2x^2 + \ldots + c_{n-1}x^{n-1}$

3-phase commitment in linear-prover-time PCS:

1. Linear Encoding
2. Leaf Hashing
3. Merkle Tree

Polynomial Commitment Schemes (PCS)

- Central primitives in proof systems
- Prove genuine evaluation of a polynomial
  - $f(x) = c_0x^0 + c_1x^1 + c_2x^2 + \ldots + c_{n-1}x^{n-1}$

3-phase commitment in linear-prover-time PCS:

1. Linear Encoding
2. Leaf Hashing
3. Merkle Tree

→**Cause substantial latency**

Phase 1: Graph-Based Linear Encoding

Phase 1: Graph-Based Linear Encoding

$$f(x) = c_0x^0 + c_1x^1 + c_2x^2 + \ldots + c_{n-1}x^{n-1} \longrightarrow \boldsymbol{C}$$

Phase 1: Graph-Based Linear Encoding

$$f(x) = c_0x^0 + c_1x^1 + c_2x^2 + \ldots + c_{n-1}x^{n-1} \longrightarrow \boldsymbol{C}$$

$$\boldsymbol{C}$$

$$\begin{bmatrix} c_{0,0} & c_{0,1} & \cdots & c_{0,n-1} \\ c_{1,0} & c_{1,1} & \cdots & c_{1,n-1} \\ c_{2,0} & c_{2,1} & \cdots & c_{2,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ c_{k-1,0} & c_{k-1,1} & \cdots & c_{k-1,n-1} \end{bmatrix}$$

Phase 1: Graph-Based Linear Encoding

$$f(x) = c_0x^0 + c_1x^1 + c_2x^2 + \ldots + c_{n-1}x^{n-1} \longrightarrow \boldsymbol{C}$$

$$\boldsymbol{C}$$

$$\begin{bmatrix} c_{0,0} & c_{0,1} & \cdots & c_{0,n-1} \\ c_{1,0} & c_{1,1} & \cdots & c_{1,n-1} \\ c_{2,0} & c_{2,1} & \cdots & c_{2,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ c_{k-1,0} & c_{k-1,1} & \cdots & c_{k-1,n-1} \end{bmatrix}$$

$$\boldsymbol{W} = Enc(\boldsymbol{C})$$

$$\begin{bmatrix} w_{0,0} & w_{0,1} & \cdots & w_{0,m-1} \\ w_{1,0} & w_{1,1} & \cdots & w_{1,m-1} \\ w_{2,0} & w_{2,1} & \cdots & w_{2,m-1} \\ \vdots & \vdots & \ddots & \vdots \\ w_{k-1,0} & w_{k-1,1} & \cdots & w_{k-1,m-1} \end{bmatrix}$$

Phase 1: Graph-Based Linear Encoding

$$f(x) = c_0x^0 + c_1x^1 + c_2x^2 + \ldots + c_{n-1}x^{n-1} \longrightarrow \boldsymbol{C}$$

$$\boldsymbol{C}$$

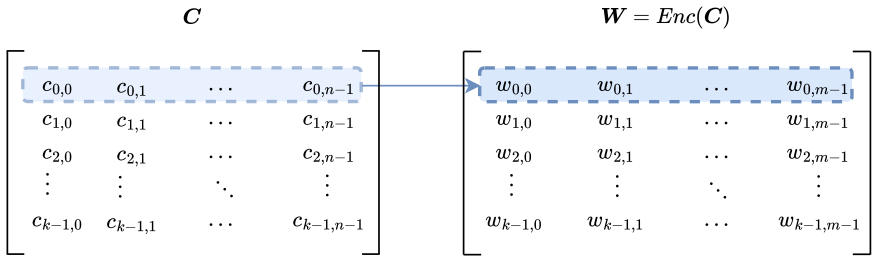
$$\begin{bmatrix} c_{0,0} & c_{0,1} & \cdots & c_{0,n-1} \\ c_{1,0} & c_{1,1} & \cdots & c_{1,n-1} \\ c_{2,0} & c_{2,1} & \cdots & c_{2,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ c_{k-1,0} & c_{k-1,1} & \cdots & c_{k-1,n-1} \end{bmatrix}$$

$$\boldsymbol{W} = Enc(\boldsymbol{C})$$

$$\begin{bmatrix} w_{0,0} & w_{0,1} & \cdots & w_{0,m-1} \\ w_{1,0} & w_{1,1} & \cdots & w_{1,m-1} \\ w_{2,0} & w_{2,1} & \cdots & w_{2,m-1} \\ \vdots & \vdots & \ddots & \vdots \\ w_{k-1,0} & w_{k-1,1} & \cdots & w_{k-1,m-1} \end{bmatrix}$$

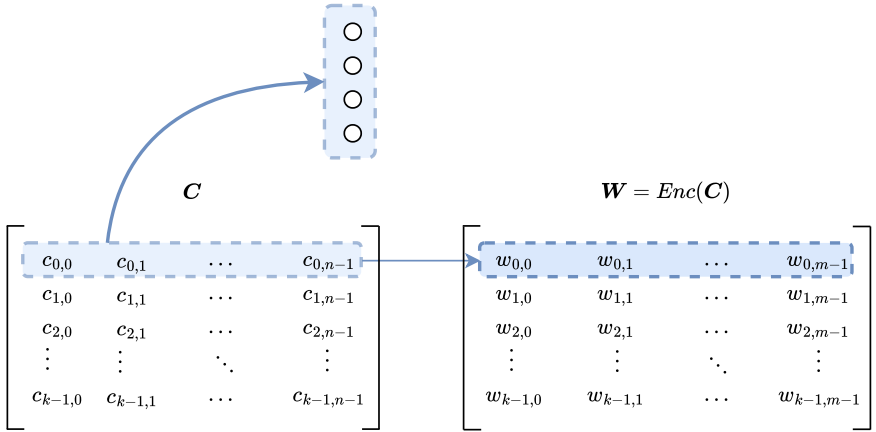
Phase 1: Graph-Based Linear Encoding

$$f(x) = c_0x^0 + c_1x^1 + c_2x^2 + \ldots + c_{n-1}x^{n-1} \longrightarrow \boldsymbol{C}$$



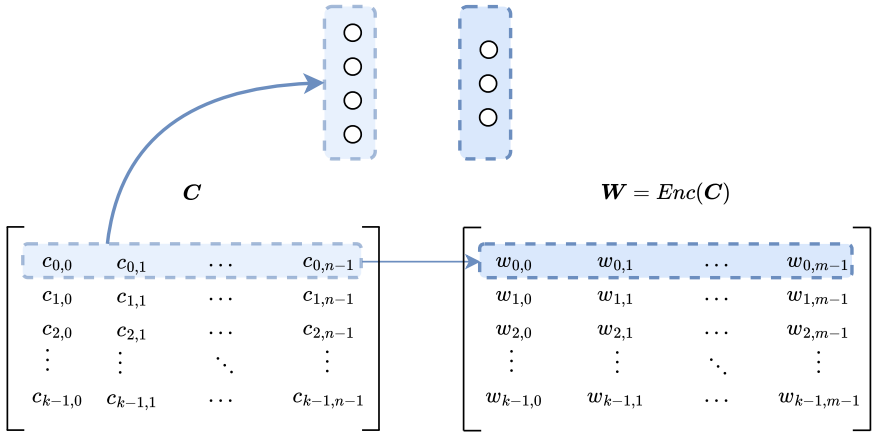
Phase 1: Graph-Based Linear Encoding

$$f(x) = c_0x^0 + c_1x^1 + c_2x^2 + \ldots + c_{n-1}x^{n-1} \longrightarrow \boldsymbol{C}$$



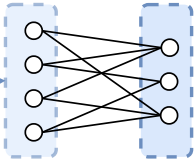
Phase 1: Graph-Based Linear Encoding

$$f(x) = c_0x^0 + c_1x^1 + c_2x^2 + \ldots + c_{n-1}x^{n-1} \longrightarrow \boldsymbol{C}$$



Phase 1: Graph-Based Linear Encoding

$$f(x) = c_0x^0 + c_1x^1 + c_2x^2 + \ldots + c_{n-1}x^{n-1} \longrightarrow \boldsymbol{C}$$



$\boldsymbol{C}$

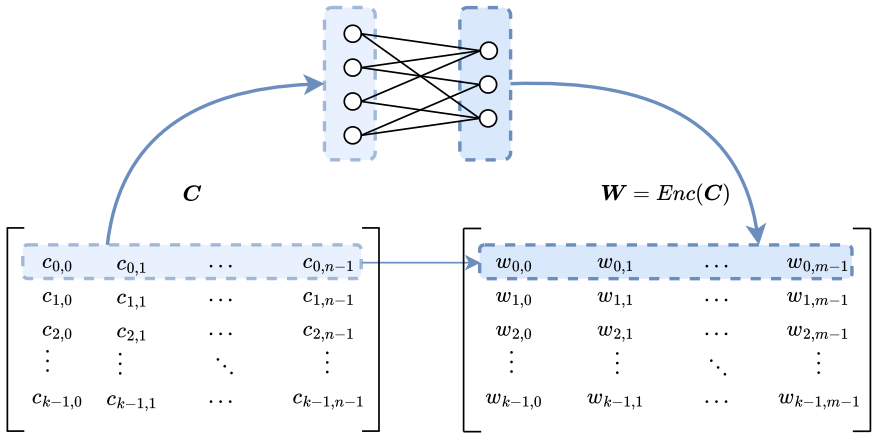
$$\begin{bmatrix} c_{0,0} & c_{0,1} & \cdots & c_{0,n-1} \\ c_{1,0} & c_{1,1} & \cdots & c_{1,n-1} \\ c_{2,0} & c_{2,1} & \cdots & c_{2,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ c_{k-1,0} & c_{k-1,1} & \cdots & c_{k-1,n-1} \end{bmatrix}$$

$\boldsymbol{W} = Enc(\boldsymbol{C})$

$$\begin{bmatrix} w_{0,0} & w_{0,1} & \cdots & w_{0,m-1} \\ w_{1,0} & w_{1,1} & \cdots & w_{1,m-1} \\ w_{2,0} & w_{2,1} & \cdots & w_{2,m-1} \\ \vdots & \vdots & \ddots & \vdots \\ w_{k-1,0} & w_{k-1,1} & \cdots & w_{k-1,m-1} \end{bmatrix}$$

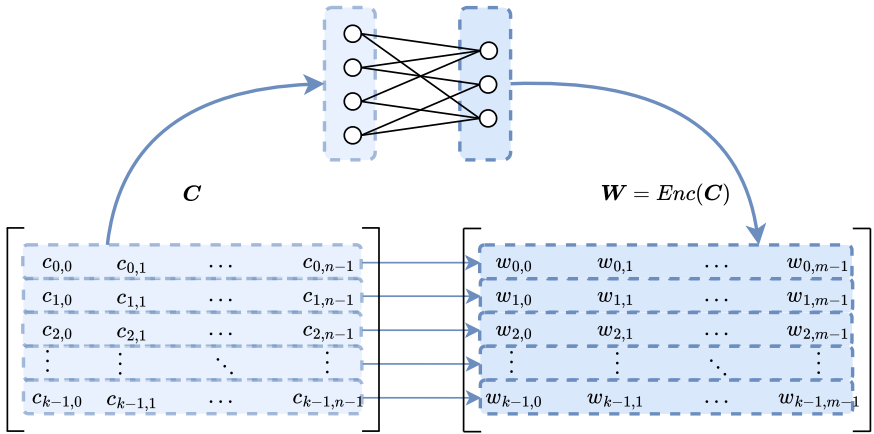
Phase 1: Graph-Based Linear Encoding

$$f(x) = c_0x^0 + c_1x^1 + c_2x^2 + \ldots + c_{n-1}x^{n-1} \longrightarrow \boldsymbol{C}$$



Phase 1: Graph-Based Linear Encoding

$$f(x) = c_0x^0 + c_1x^1 + c_2x^2 + \ldots + c_{n-1}x^{n-1} \longrightarrow \boldsymbol{C}$$



Phase 2+3: Leaf Hashing + Merkle Tree Generation

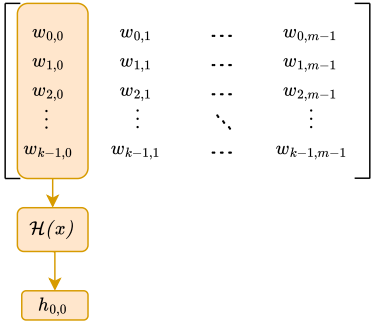
Phase 2+3: Leaf Hashing + Merkle Tree Generation

$$\begin{bmatrix} w_{0,0} & w_{0,1} & \cdots & w_{0,m-1} \\ w_{1,0} & w_{1,1} & \cdots & w_{1,m-1} \\ w_{2,0} & w_{2,1} & \cdots & w_{2,m-1} \\ \vdots & \vdots & \ddots & \vdots \\ w_{k-1,0} & w_{k-1,1} & \cdots & w_{k-1,m-1} \end{bmatrix}$$

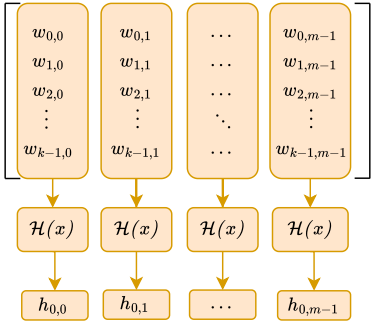
Phase 2+3: Leaf Hashing + Merkle Tree Generation

$$\begin{bmatrix} w_{0,0} & w_{0,1} & \cdots & w_{0,m-1} \\ w_{1,0} & w_{1,1} & \cdots & w_{1,m-1} \\ w_{2,0} & w_{2,1} & \cdots & w_{2,m-1} \\ \vdots & \vdots & \ddots & \vdots \\ w_{k-1,0} & w_{k-1,1} & \cdots & w_{k-1,m-1} \end{bmatrix}$$

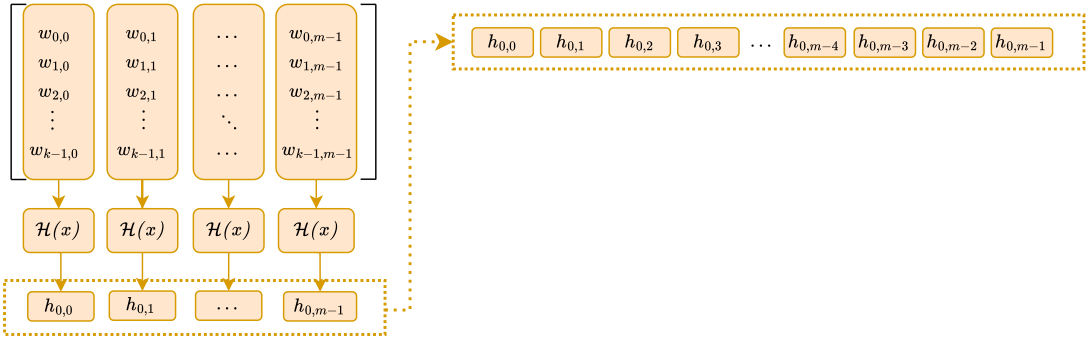
Phase 2+3: Leaf Hashing + Merkle Tree Generation



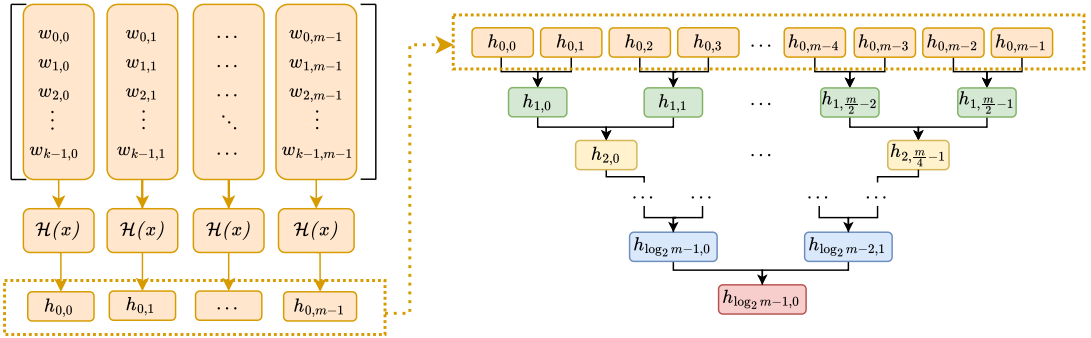
Phase 2+3: Leaf Hashing + Merkle Tree Generation



Phase 2+3: Leaf Hashing + Merkle Tree Generation



Phase 2+3: Leaf Hashing + Merkle Tree Generation



Hardware Limitation Factors

- Commitment and Proving is data-heavy

  → Polynomial size $N^{16..28}$ ($\rightarrow$ up to several GBs memory)

Hardware Limitation Factors

- Commitment and Proving is data-heavy

  → Polynomial size $N^{16..28}$ ($\rightarrow$ up to several GBs memory)

Observations:

- Linear Encoding $\rightarrow$ **Memory Bounded**
- Merkle Hashing $\rightarrow$ **Resource Bounded**

Hardware Limitation Factors

- Commitment and Proving is data-heavy

  → Polynomial size $N^{16..28}$ ($\rightarrow$ up to several GBs memory)

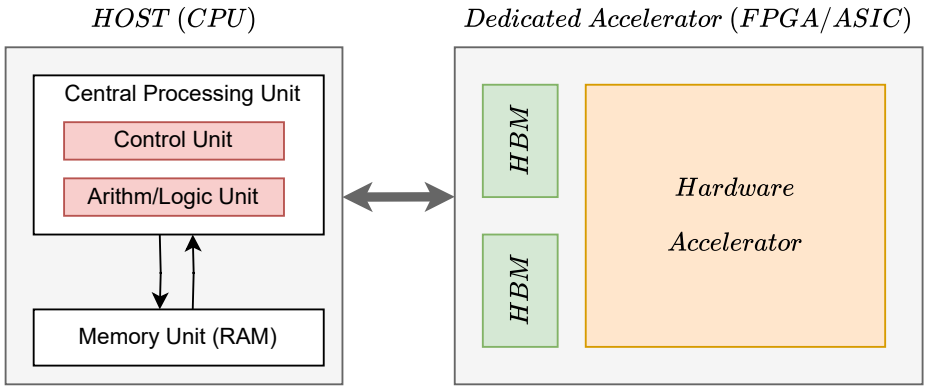
Observations:

- Linear Encoding $\rightarrow$ **Memory Bounded**
- Merkle Hashing $\rightarrow$ **Resource Bounded**

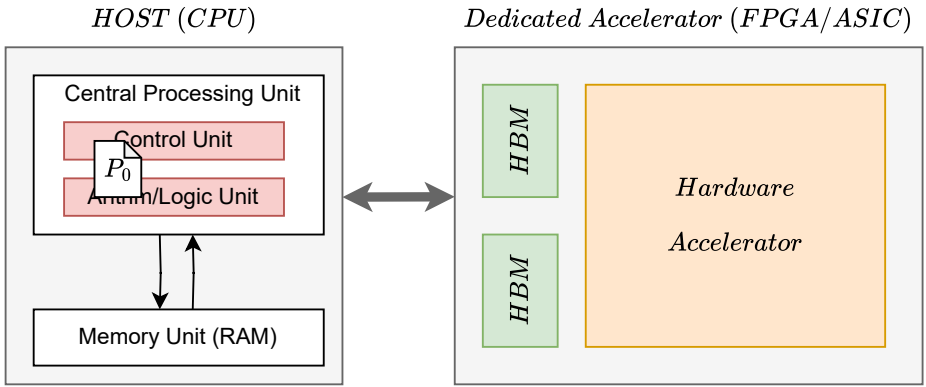
We address these limiting factors

→ Memory- and Platform-Aware Hardware Design!

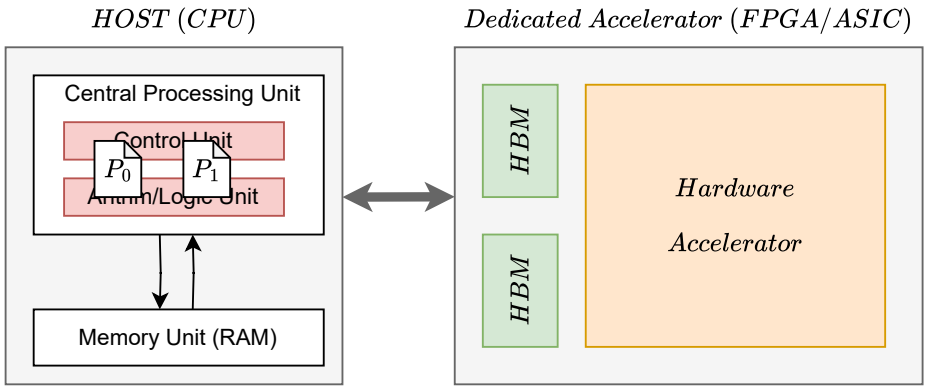
Computational Data Flow of PCS



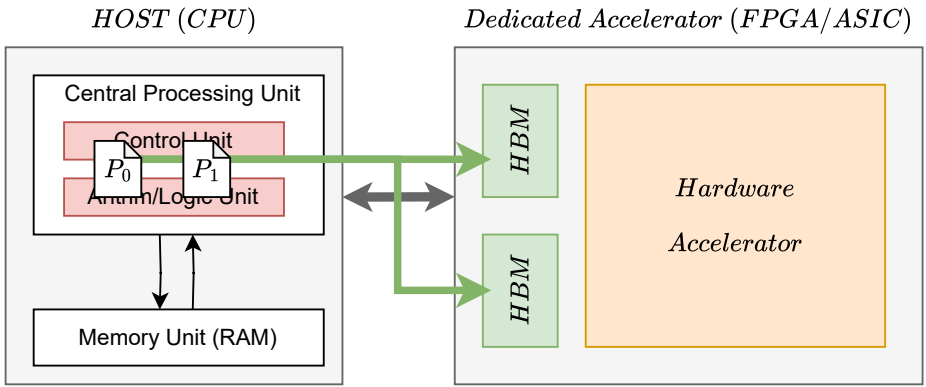
Computational Data Flow of PCS



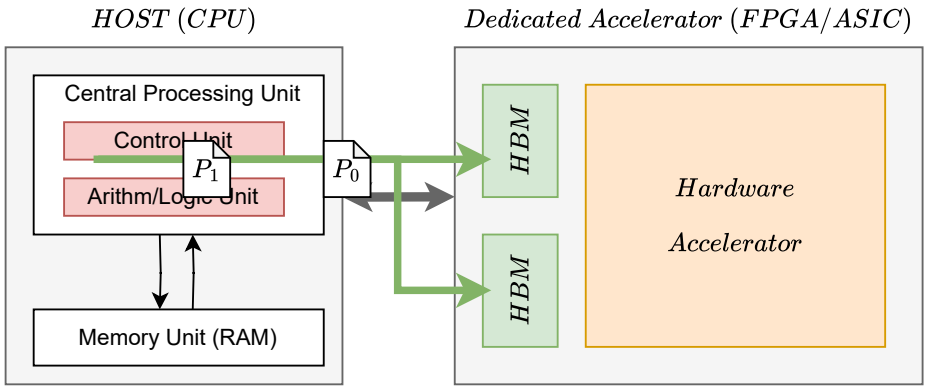
Computational Data Flow of PCS



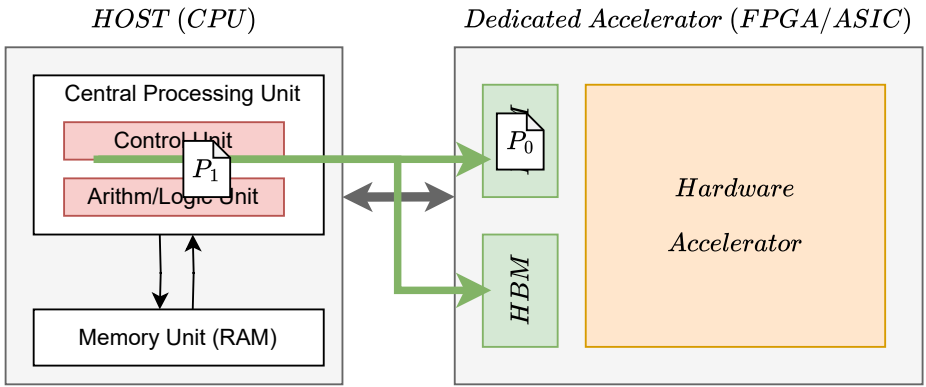
Computational Data Flow of PCS



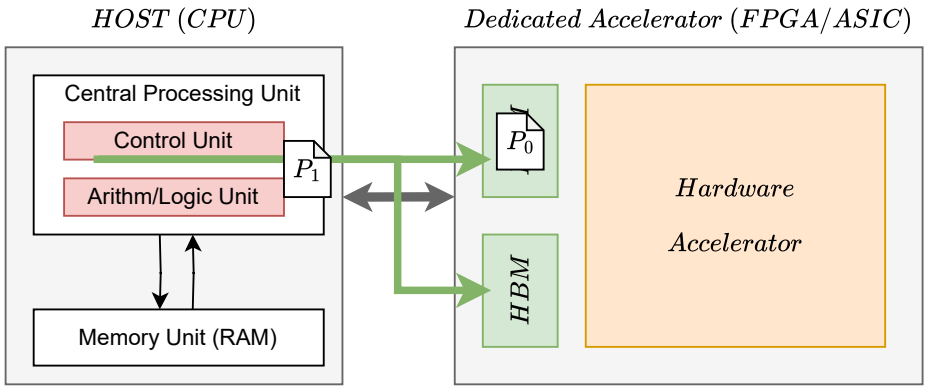
Computational Data Flow of PCS



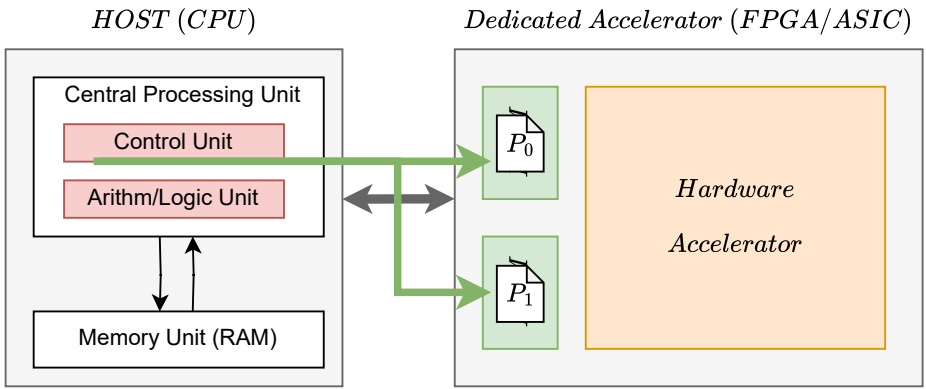
Computational Data Flow of PCS



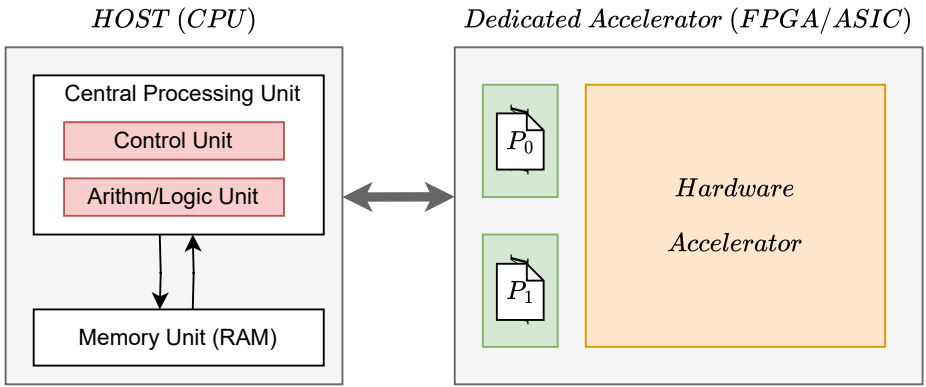
Computational Data Flow of PCS



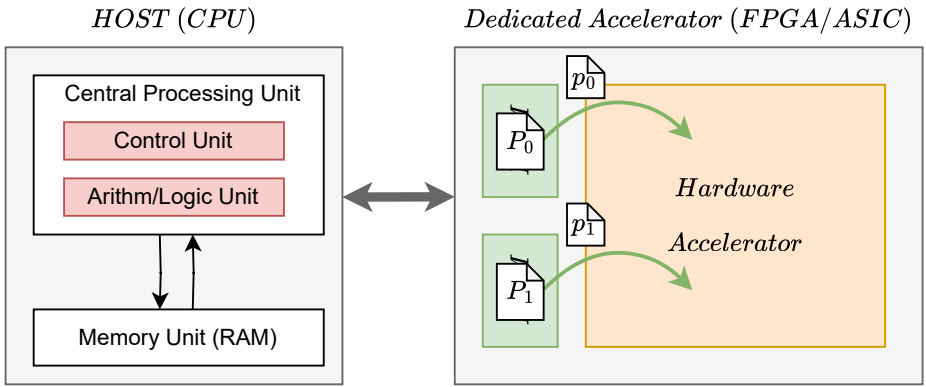
Computational Data Flow of PCS



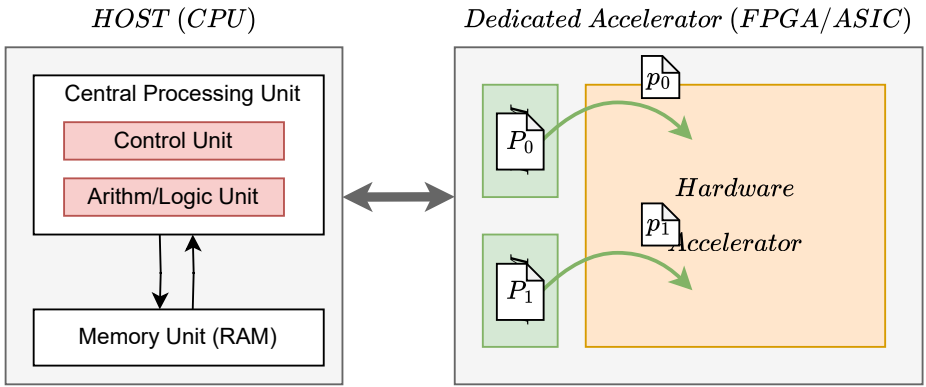
Computational Data Flow of PCS



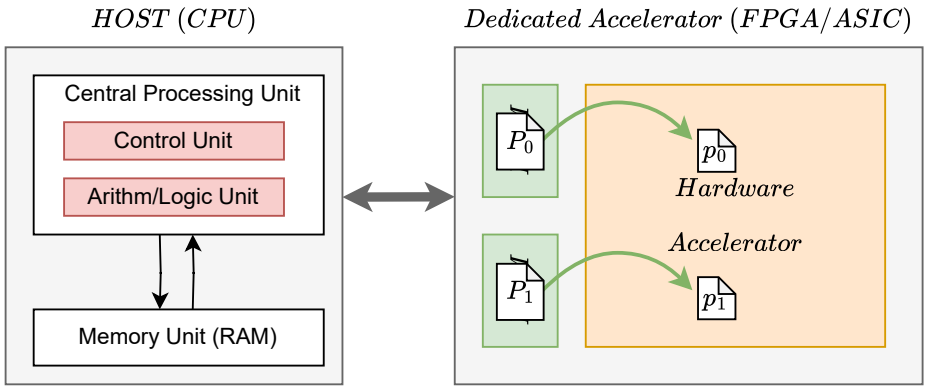
Computational Data Flow of PCS



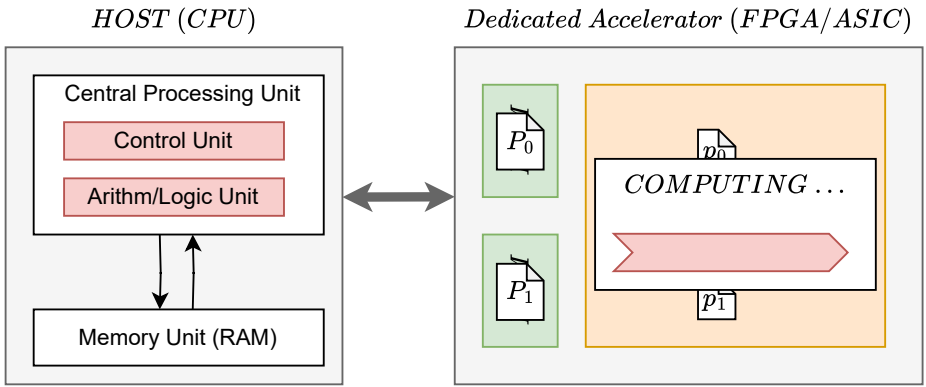
Computational Data Flow of PCS



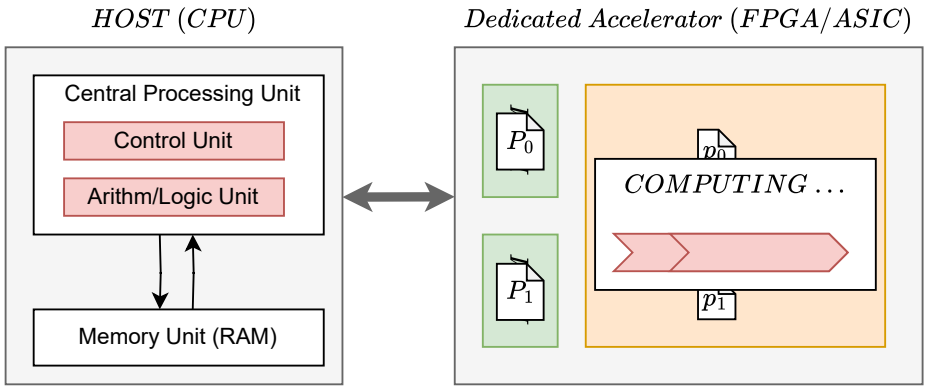
Computational Data Flow of PCS



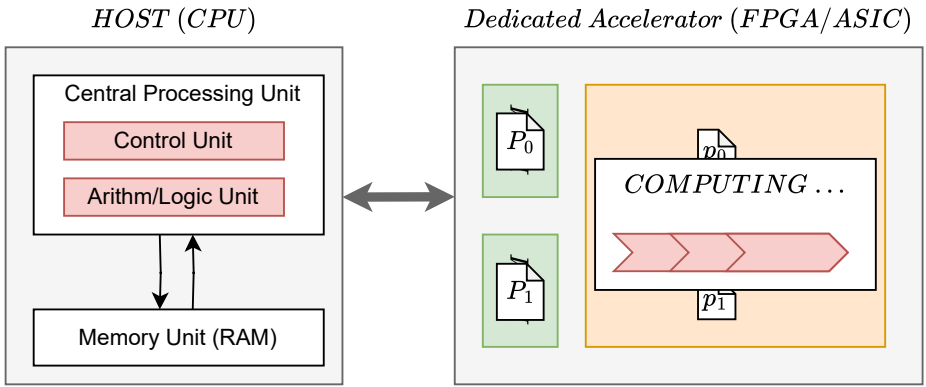
Computational Data Flow of PCS



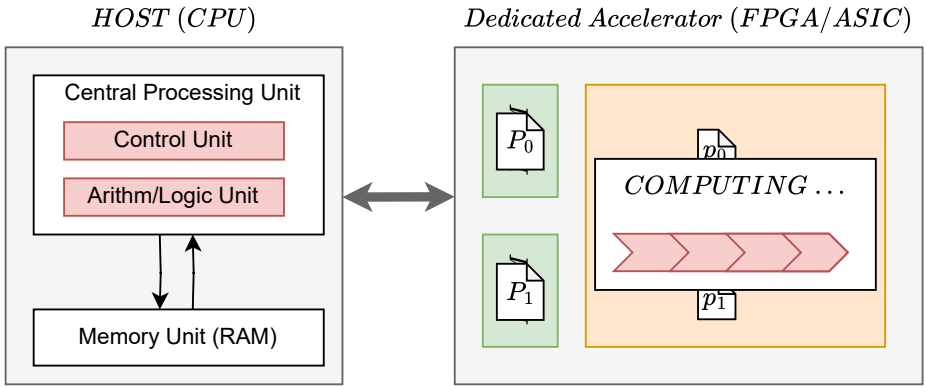
Computational Data Flow of PCS



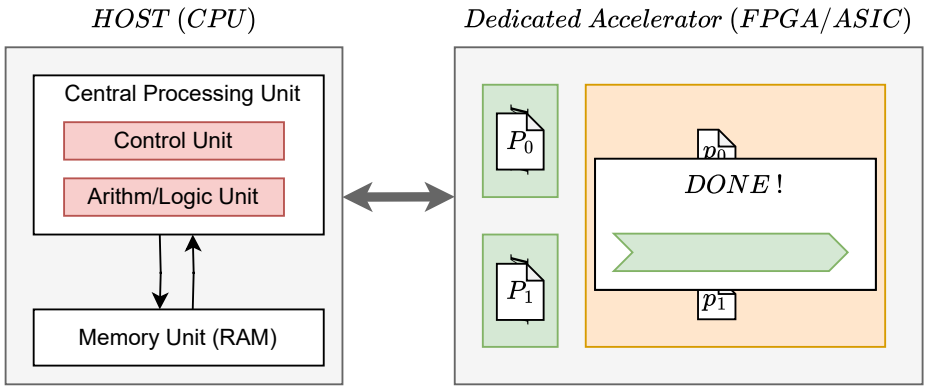
Computational Data Flow of PCS



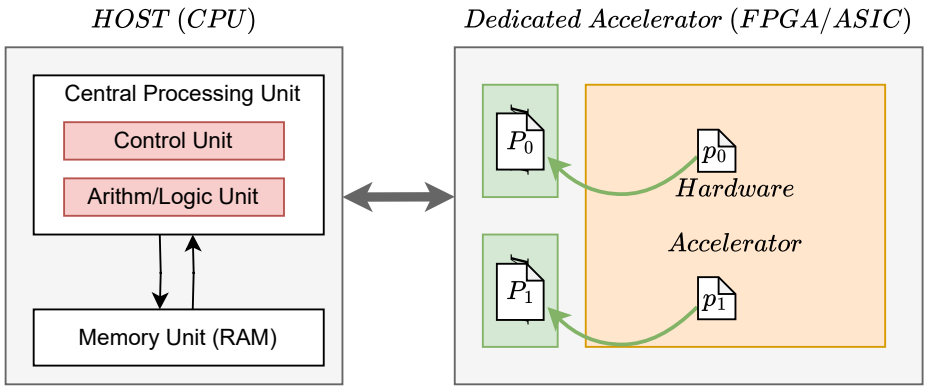
Computational Data Flow of PCS



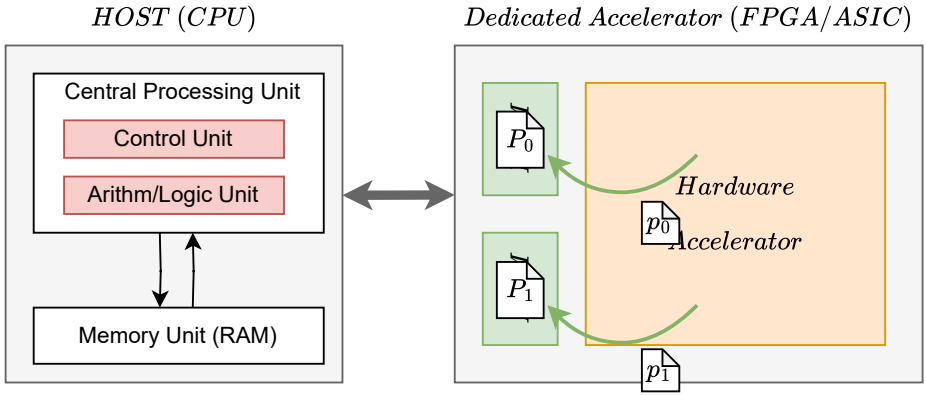
Computational Data Flow of PCS



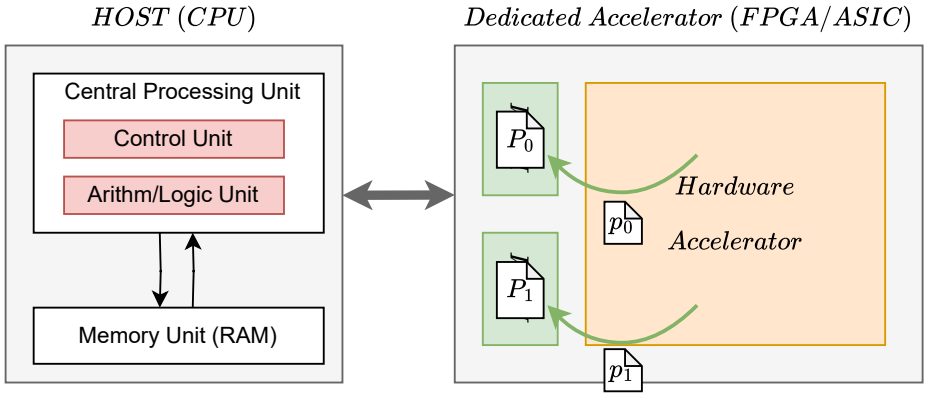
Computational Data Flow of PCS



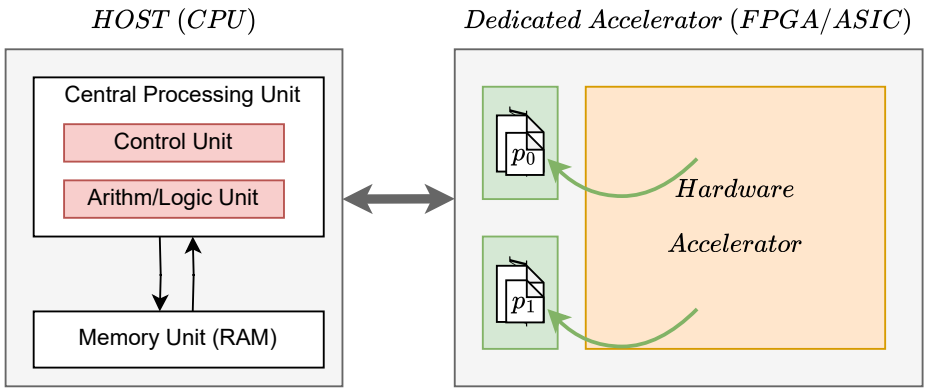
Computational Data Flow of PCS



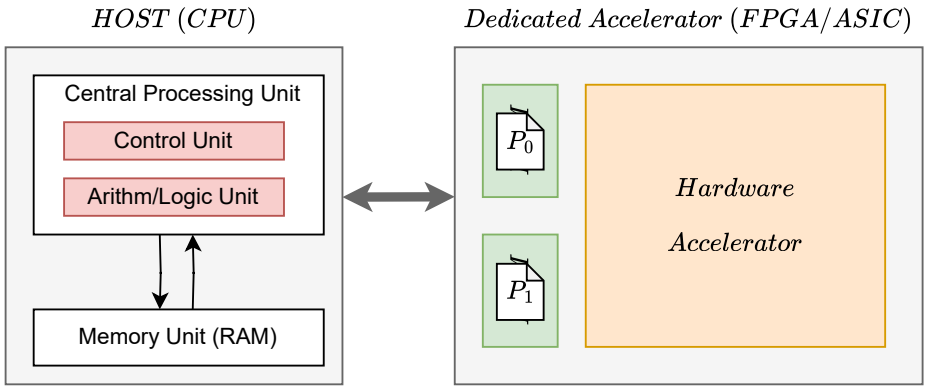
Computational Data Flow of PCS



Computational Data Flow of PCS



Computational Data Flow of PCS



High-Bandwidth Memory (HBM) on Alveo U280

High-Bandwidth Memory (HBM) on Alveo U280

- HBM2 attached to FPGA fabric
- Peak bandwidth: **460 GB/s** (vendor figure)
- **32** independent pseudo-channels (PCs)

High-Bandwidth Memory (HBM) on Alveo U280

- HBM2 attached to FPGA fabric
- Peak bandwidth: **460 GB/s** (vendor figure)
- **32** independent pseudo-channels (PCs)

Achievable bandwidth (rule-of-thumb)

Access pattern	**% of peak BW**
Linear / burst reads	∼**90%**
Frequent Rd / Wr turnarounds	∼**50%**
Random (poor locality)	∼**30%**

High-Bandwidth Memory (HBM) on Alveo U280

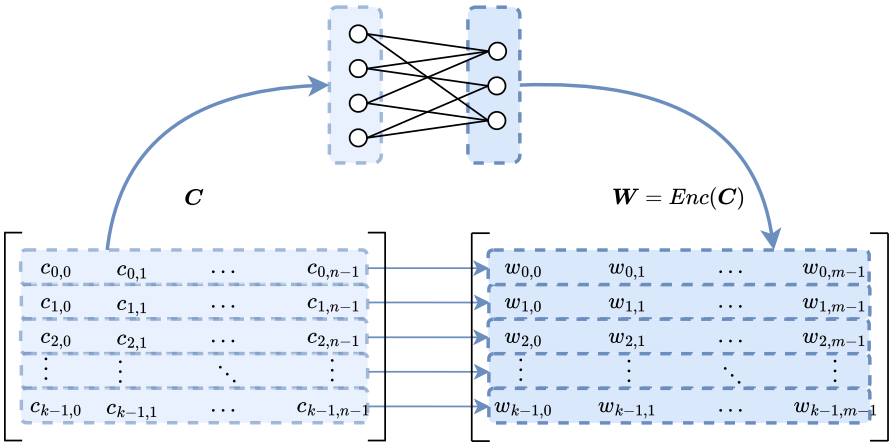
- HBM2 attached to FPGA fabric
- Peak bandwidth: **460 GB/s** (vendor figure)
- **32** independent pseudo-channels (PCs)

Achievable bandwidth (rule-of-thumb)

Access pattern	**% of peak BW**
Linear / burst reads	∼**90%**
Frequent Rd / Wr turnarounds	∼**50%**
Random (poor locality)	∼**30%**

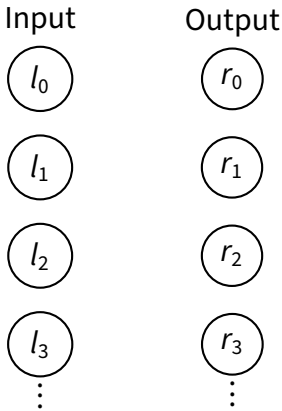
Linear Encoding heavily uses random access

Linear Encoding using Expander Graphs



Expander Graphs: The Core Operation in Spielman Codes

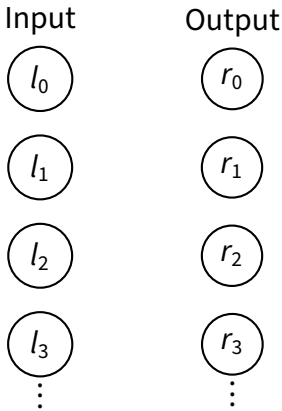
Baseline Graph: L2R



Expander Graphs: The Core Operation in Spielman Codes

Baseline Graph: L2R

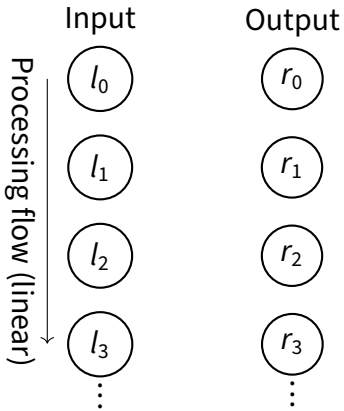
$$r_i = \sum \omega_{i,j} \cdot l_j$$



Expander Graphs: The Core Operation in Spielman Codes

Baseline Graph: L2R

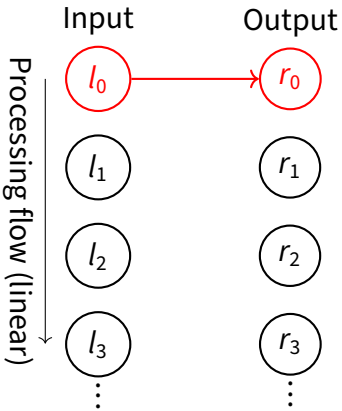
$$r_i = \sum \omega_{i,j} \cdot l_j$$



Expander Graphs: The Core Operation in Spielman Codes

Baseline Graph: L2R

$$r_i = \sum \omega_{i,j} \cdot l_j$$

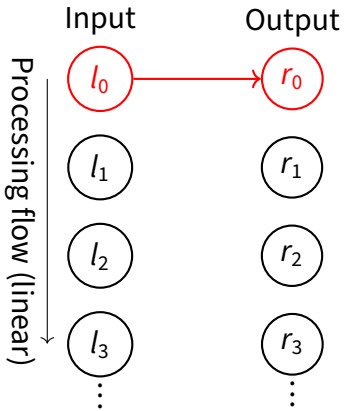


Step 1: Rd(l_0) |

Expander Graphs: The Core Operation in Spielman Codes

Baseline Graph: L2R

$$r_i = \sum \omega_{i,j} \cdot l_j$$

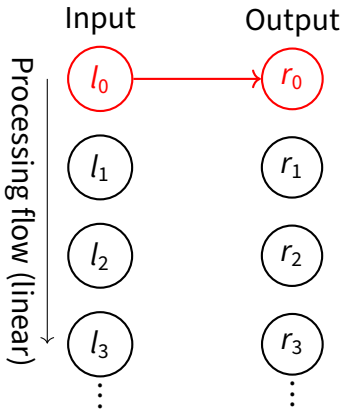


Step 1: Rd(l_0) | Rd(r_0) |

Expander Graphs: The Core Operation in Spielman Codes

Baseline Graph: L2R

$$r_i = \sum \omega_{i,j} \cdot l_j$$

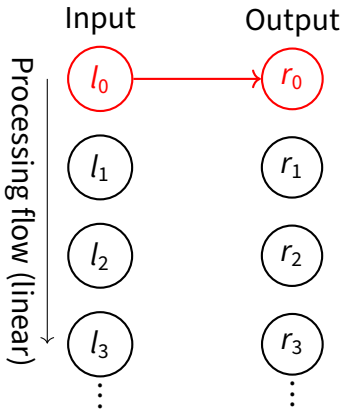


Step 1: Rd(l_0) | Rd(r_0) | r_0+= $\omega_{0,0} l_0$ |

Expander Graphs: The Core Operation in Spielman Codes

Baseline Graph: L2R

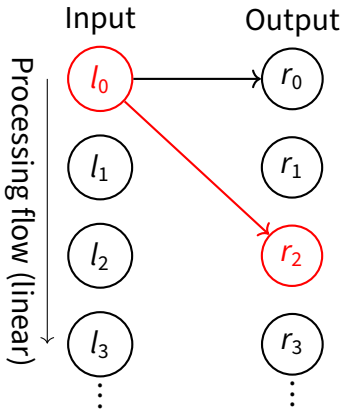
$$r_i = \sum \omega_{i,j} \cdot l_j$$



Step 1: $\mathrm{Rd}(l_0) \mid \mathrm{Rd}(r_0) \mid r_0 \mathrel{+}= \omega_{0,0} l_0 \mid \mathrm{Wr}(r_0)$

Expander Graphs: The Core Operation in Spielman Codes

Baseline Graph: L2R



$$r_i = \sum \omega_{i,j} \cdot l_j$$

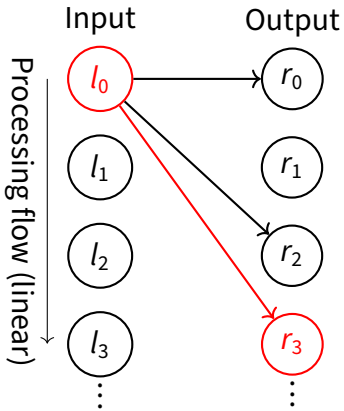
Step 1: $\mathrm{Rd}(l_0) \mid \mathrm{Rd}(r_0) \mid r_0 \mathrel{+}= \omega_{0,0} l_0 \mid \mathrm{Wr}(r_0)$

Step 2: $\mathrm{Rd}(r_2) \mid r_2 \mathrel{+}= \omega_{2,0} l_0 \mid \mathrm{Wr}(r_2)$

Expander Graphs: The Core Operation in Spielman Codes

Baseline Graph: L2R

$$r_i = \sum \omega_{i,j} \cdot l_j$$



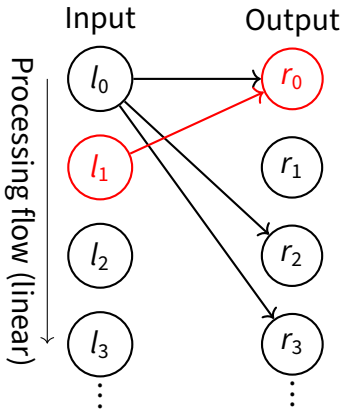
Step 1: $\text{Rd}(l_0) \mid \text{Rd}(r_0) \mid r_0 \mathrel{+}= \omega_{0,0} l_0 \mid \text{Wr}(r_0)$

Step 2: $\text{Rd}(r_2) \mid r_2 \mathrel{+}= \omega_{2,0} l_0 \mid \text{Wr}(r_2)$

Step 3: $\text{Rd}(r_3) \mid r_3 \mathrel{+}= \omega_{3,0} l_0 \mid \text{Wr}(r_3)$

Expander Graphs: The Core Operation in Spielman Codes

Baseline Graph: L2R



$$r_i = \sum \omega_{i,j} \cdot l_j$$

Step 1: $\text{Rd}(l_0) \mid \text{Rd}(r_0) \mid r_0 \mathrel{+}= \omega_{0,0} l_0 \mid \text{Wr}(r_0)$

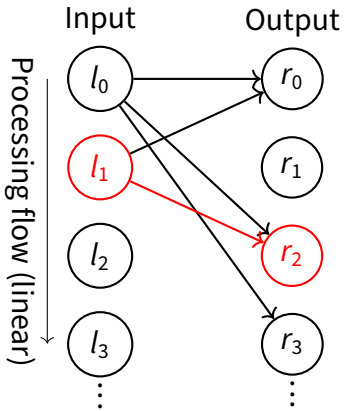
Step 2: $\text{Rd}(r_2) \mid r_2 \mathrel{+}= \omega_{2,0} l_0 \mid \text{Wr}(r_2)$

Step 3: $\text{Rd}(r_3) \mid r_3 \mathrel{+}= \omega_{3,0} l_0 \mid \text{Wr}(r_3)$

Step 4: $\text{Rd}(l_1) \mid \text{Rd}(r_0) \mid r_0 \mathrel{+}= \omega_{0,1} l_1 \mid \text{Wr}(r_0)$

Expander Graphs: The Core Operation in Spielman Codes

Baseline Graph: L2R



$$r_i = \sum \omega_{i,j} \cdot l_j$$

Step 1: $\mathrm{Rd}(l_0) \mid \mathrm{Rd}(r_0) \mid r_0 \mathrel{+}= \omega_{0,0} l_0 \mid \mathrm{Wr}(r_0)$

Step 2: $\mathrm{Rd}(r_2) \mid r_2 \mathrel{+}= \omega_{2,0} l_0 \mid \mathrm{Wr}(r_2)$

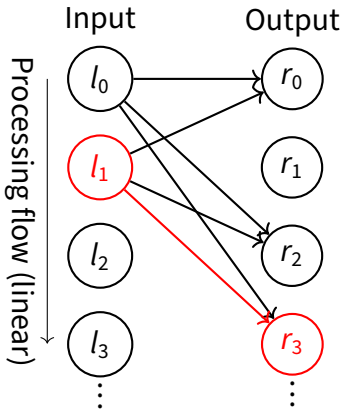
Step 3: $\mathrm{Rd}(r_3) \mid r_3 \mathrel{+}= \omega_{3,0} l_0 \mid \mathrm{Wr}(r_3)$

Step 4: $\mathrm{Rd}(l_1) \mid \mathrm{Rd}(r_0) \mid r_0 \mathrel{+}= \omega_{0,1} l_1 \mid \mathrm{Wr}(r_0)$

Step 5: $\mathrm{Rd}(r_2) \mid r_2 \mathrel{+}= \omega_{2,1} l_1 \mid \mathrm{Wr}(r_2)$

Expander Graphs: The Core Operation in Spielman Codes

Baseline Graph: L2R

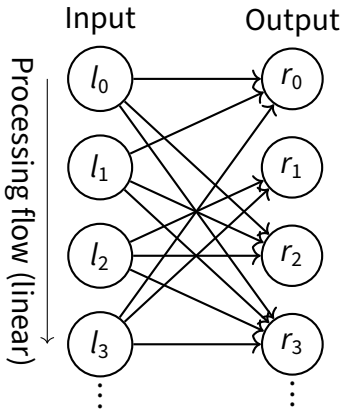


$$r_i = \sum \omega_{i,j} \cdot l_j$$

Step 1:	Rd(l_0)	Rd(r_0)	$r_0 \mathrel{+}= \omega_{0,0} l_0$	Wr(r_0)
Step 2:		Rd(r_2)	$r_2 \mathrel{+}= \omega_{2,0} l_0$	Wr(r_2)
Step 3:		Rd(r_3)	$r_3 \mathrel{+}= \omega_{3,0} l_0$	Wr(r_3)
Step 4:	Rd(l_1)	Rd(r_0)	$r_0 \mathrel{+}= \omega_{0,1} l_1$	Wr(r_0)
Step 5:		Rd(r_2)	$r_2 \mathrel{+}= \omega_{2,1} l_1$	Wr(r_2)
Step 6:		Rd(r_3)	$r_3 \mathrel{+}= \omega_{3,1} l_1$	Wr(r_3)

Expander Graphs: The Core Operation in Spielman Codes

Baseline Graph: L2R

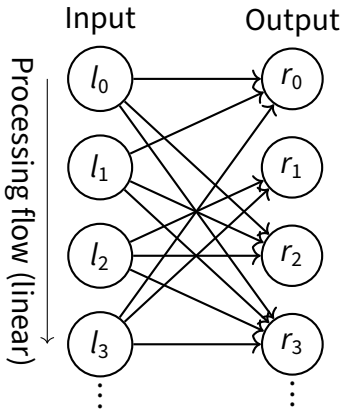


$$r_i = \sum \omega_{i,j} \cdot l_j$$

Step 1: $\mathrm{Rd}(l_0) \mid \mathrm{Rd}(r_0) \mid r_0 \mathrel{+}= \omega_{0,0} l_0 \mid \mathrm{Wr}(r_0)$
Step 2: $\mathrm{Rd}(r_2) \mid r_2 \mathrel{+}= \omega_{2,0} l_0 \mid \mathrm{Wr}(r_2)$
Step 3: $\mathrm{Rd}(r_3) \mid r_3 \mathrel{+}= \omega_{3,0} l_0 \mid \mathrm{Wr}(r_3)$
Step 4: $\mathrm{Rd}(l_1) \mid \mathrm{Rd}(r_0) \mid r_0 \mathrel{+}= \omega_{0,1} l_1 \mid \mathrm{Wr}(r_0)$
Step 5: $\mathrm{Rd}(r_2) \mid r_2 \mathrel{+}= \omega_{2,1} l_1 \mid \mathrm{Wr}(r_2)$
Step 6: $\mathrm{Rd}(r_3) \mid r_3 \mathrel{+}= \omega_{3,1} l_1 \mid \mathrm{Wr}(r_3)$
$\ldots$

Expander Graphs: The Core Operation in Spielman Codes

Baseline Graph: L2R



$$r_i = \sum \omega_{i,j} \cdot l_j$$

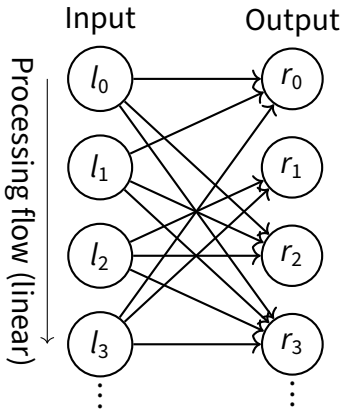
Step 1:	$\mathrm{Rd}(l_0) \mid \mathrm{Rd}(r_0) \mid r_0 \mathrel{+}= \omega_{0,0} l_0 \mid \mathrm{Wr}(r_0)$
Step 2:	$\mathrm{Rd}(r_2) \mid r_2 \mathrel{+}= \omega_{2,0} l_0 \mid \mathrm{Wr}(r_2)$
Step 3:	$\mathrm{Rd}(r_3) \mid r_3 \mathrel{+}= \omega_{3,0} l_0 \mid \mathrm{Wr}(r_3)$
Step 4:	$\mathrm{Rd}(l_1) \mid \mathrm{Rd}(r_0) \mid r_0 \mathrel{+}= \omega_{0,1} l_1 \mid \mathrm{Wr}(r_0)$
Step 5:	$\mathrm{Rd}(r_2) \mid r_2 \mathrel{+}= \omega_{2,1} l_1 \mid \mathrm{Wr}(r_2)$
Step 6:	$\mathrm{Rd}(r_3) \mid r_3 \mathrel{+}= \omega_{3,1} l_1 \mid \mathrm{Wr}(r_3)$
	...

Problems of L2R:

- ⛔ Many random Rd / Wr

Expander Graphs: The Core Operation in Spielman Codes

Baseline Graph: L2R



$$r_i = \sum \omega_{i,j} \cdot l_j$$

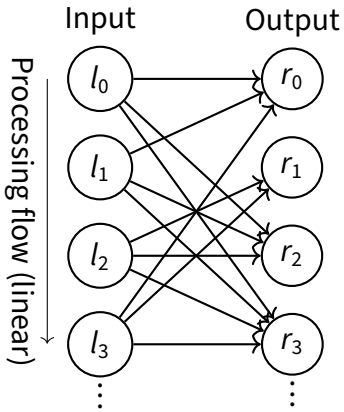
Step 1:	$\mathrm{Rd}(l_0)$	$\mathrm{Rd}(r_0)$	$r_0 \mathrel{+}= \omega_{0,0} l_0$ | $\mathrm{Wr}(r_0)$
Step 2:		$\mathrm{Rd}(r_2)$	$r_2 \mathrel{+}= \omega_{2,0} l_0$ | $\mathrm{Wr}(r_2)$
Step 3:		$\mathrm{Rd}(r_3)$	$r_3 \mathrel{+}= \omega_{3,0} l_0$ | $\mathrm{Wr}(r_3)$
Step 4:	$\mathrm{Rd}(l_1)$	$\mathrm{Rd}(r_0)$	$r_0 \mathrel{+}= \omega_{0,1} l_1$ | $\mathrm{Wr}(r_0)$
Step 5:		$\mathrm{Rd}(r_2)$	$r_2 \mathrel{+}= \omega_{2,1} l_1$ | $\mathrm{Wr}(r_2)$
Step 6:		$\mathrm{Rd}(r_3)$	$r_3 \mathrel{+}= \omega_{3,1} l_1$ | $\mathrm{Wr}(r_3)$
	...		

Problems of L2R:

- ⛔ Many random Rd / Wr
- ⛔ Rd / Wr turnarounds

Expander Graphs: The Core Operation in Spielman Codes

Baseline Graph: L2R



$$r_i = \sum \omega_{i,j} \cdot l_j$$

Step 1:	Rd(l_0)	Rd(r_0)	r_0+=$\omega_{0,0} l_0$	Wr(r_0)
Step 2:		Rd(r_2)	r_2+=$\omega_{2,0} l_0$	Wr(r_2)
Step 3:		Rd(r_3)	r_3+=$\omega_{3,0} l_0$	Wr(r_3)
Step 4:	Rd(l_1)	Rd(r_0)	r_0+=$\omega_{0,1} l_1$	Wr(r_0)
Step 5:		Rd(r_2)	r_2+=$\omega_{2,1} l_1$	Wr(r_2)
Step 6:		Rd(r_3)	r_3+=$\omega_{3,1} l_1$	Wr(r_3)
	...			

Problems of L2R:

- ⛔ Many random Rd / Wr
- ⛔ Rd / Wr turnarounds
- ⛔ Read-after-Write hazards

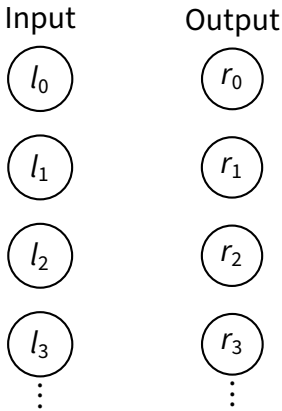
Our Inverted Expander Graphs

R2L Evaluation

Inverted Expander Graphs

Inverted Graph: R2L

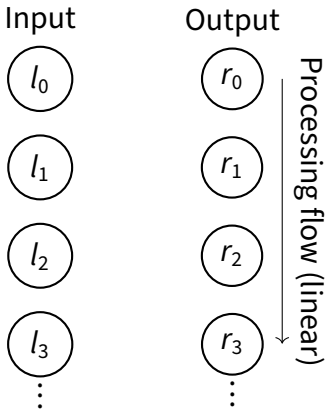
$$r_i = \sum \omega_{i,j} \cdot l_j$$



Inverted Expander Graphs

Inverted Graph: R2L

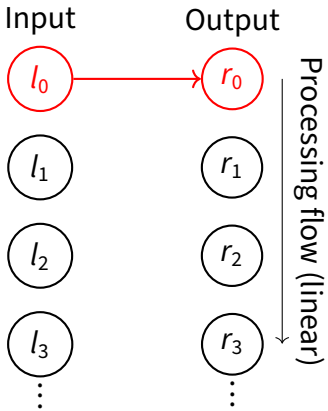
$$r_i = \sum \omega_{i,j} \cdot l_j$$



Inverted Expander Graphs

Inverted Graph: R2L

$$r_i = \sum \omega_{i,j} \cdot l_j$$

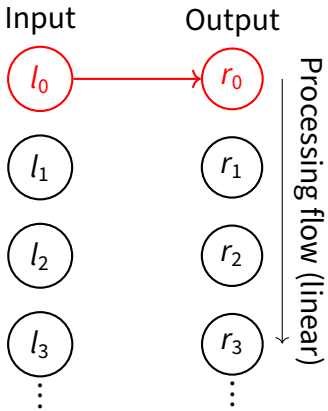


Step 1: $\mathrm{Rd}(l_0)$ |

Inverted Expander Graphs

Inverted Graph: R2L

$$r_i = \sum \omega_{i,j} \cdot l_j$$

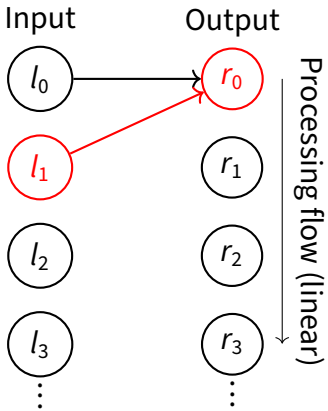


Step 1: $\text{Rd}(l_0) \mid r_0 \mathrel{+}= \omega_{0,0} l_0$

Inverted Expander Graphs

Inverted Graph: R2L

$$r_i = \sum \omega_{i,j} \cdot l_j$$

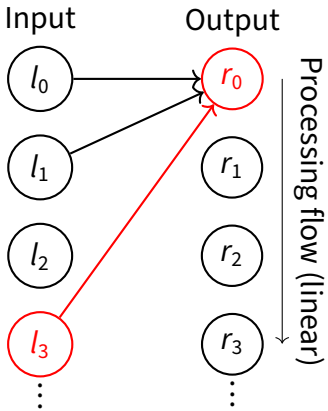


Step 1: $\text{Rd}(l_0) \mid r_0 \mathrel{+}= \omega_{0,0} l_0$

Step 2: $\text{Rd}(l_1) \mid r_0 \mathrel{+}= \omega_{0,1} l_1$

Inverted Expander Graphs

Inverted Graph: R2L



$$r_i = \sum \omega_{i,j} \cdot l_j$$

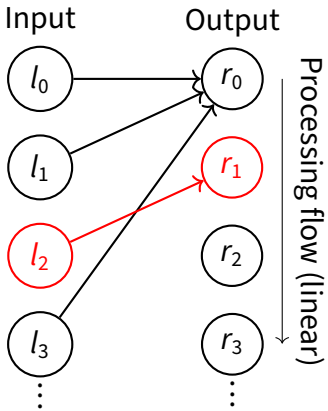
Step 1: $\text{Rd}(l_0) \mid r_0 \mathrel{+}= \omega_{0,0} l_0$

Step 2: $\text{Rd}(l_1) \mid r_0 \mathrel{+}= \omega_{0,1} l_1$

Step 3: $\text{Rd}(l_3) \mid r_0 \mathrel{+}= \omega_{0,3} l_3 \mid \text{Wr}(r_0)$

Inverted Expander Graphs

Inverted Graph: R2L



$$r_i = \sum \omega_{i,j} \cdot l_j$$

Step 1: $\mathrm{Rd}(l_0) \mid r_0 \mathrel{+}= \omega_{0,0} l_0$

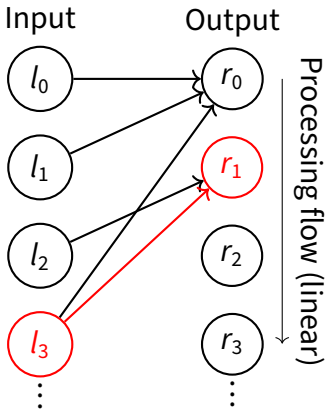
Step 2: $\mathrm{Rd}(l_1) \mid r_0 \mathrel{+}= \omega_{0,1} l_1$

Step 3: $\mathrm{Rd}(l_3) \mid r_0 \mathrel{+}= \omega_{0,3} l_3 \mid \mathrm{Wr}(r_0)$

Step 4: $\mathrm{Rd}(l_2) \mid r_1 \mathrel{+}= \omega_{1,2} l_2$

Inverted Expander Graphs

Inverted Graph: R2L

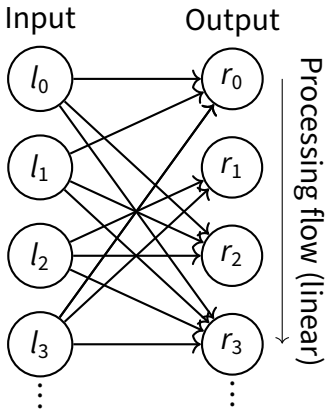


$$r_i = \sum \omega_{i,j} \cdot l_j$$

Step 1: $\text{Rd}(l_0) \mid r_0 \mathrel{+}= \omega_{0,0} l_0$
Step 2: $\text{Rd}(l_1) \mid r_0 \mathrel{+}= \omega_{0,1} l_1$
Step 3: $\text{Rd}(l_3) \mid r_0 \mathrel{+}= \omega_{0,3} l_3 \mid \text{Wr}(r_0)$
Step 4: $\text{Rd}(l_2) \mid r_1 \mathrel{+}= \omega_{1,2} l_2$
Step 5: $\text{Rd}(l_3) \mid r_1 \mathrel{+}= \omega_{1,3} l_3 \mid \text{Wr}(r_1)$

Inverted Expander Graphs

Inverted Graph: R2L

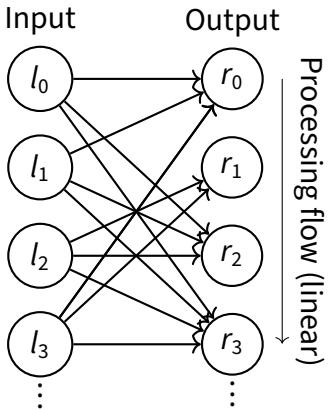


$$r_i = \sum \omega_{i,j} \cdot l_j$$

Step 1: $\mathrm{Rd}(l_0) \mid r_0 \mathrel{+}= \omega_{0,0} l_0$
Step 2: $\mathrm{Rd}(l_1) \mid r_0 \mathrel{+}= \omega_{0,1} l_1$
Step 3: $\mathrm{Rd}(l_3) \mid r_0 \mathrel{+}= \omega_{0,3} l_3 \mid \mathrm{Wr}(r_0)$
Step 4: $\mathrm{Rd}(l_2) \mid r_1 \mathrel{+}= \omega_{1,2} l_2$
Step 5: $\mathrm{Rd}(l_3) \mid r_1 \mathrel{+}= \omega_{1,3} l_3 \mid \mathrm{Wr}(r_1)$
$\ldots$

Inverted Expander Graphs

Inverted Graph: R2L



$$r_i = \sum \omega_{i,j} \cdot l_j$$

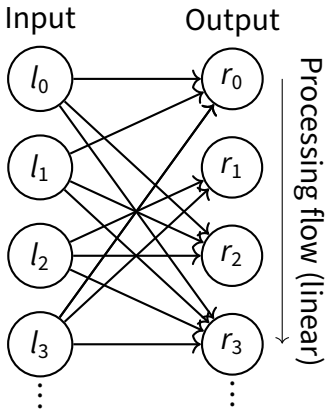
Step 1: $\mathrm{Rd}(l_0) \mid r_0 \mathrel{+}= \omega_{0,0} l_0$
Step 2: $\mathrm{Rd}(l_1) \mid r_0 \mathrel{+}= \omega_{0,1} l_1$
Step 3: $\mathrm{Rd}(l_3) \mid r_0 \mathrel{+}= \omega_{0,3} l_3 \mid \mathrm{Wr}(r_0)$
Step 4: $\mathrm{Rd}(l_2) \mid r_1 \mathrel{+}= \omega_{1,2} l_2$
Step 5: $\mathrm{Rd}(l_3) \mid r_1 \mathrel{+}= \omega_{1,3} l_3 \mid \mathrm{Wr}(r_1)$
$\ldots$

Benefits of R2L:

- ⊕ Only linear Wr

Inverted Expander Graphs

Inverted Graph: R2L



$$r_i = \sum \omega_{i,j} \cdot l_j$$

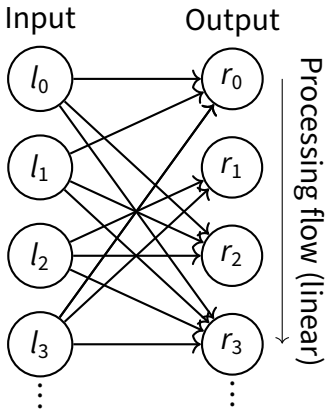
Step 1: $\mathrm{Rd}(l_0) \mid r_0 \mathrel{+}= \omega_{0,0} l_0$
Step 2: $\mathrm{Rd}(l_1) \mid r_0 \mathrel{+}= \omega_{0,1} l_1$
Step 3: $\mathrm{Rd}(l_3) \mid r_0 \mathrel{+}= \omega_{0,3} l_3 \mid \mathrm{Wr}(r_0)$
Step 4: $\mathrm{Rd}(l_2) \mid r_1 \mathrel{+}= \omega_{1,2} l_2$
Step 5: $\mathrm{Rd}(l_3) \mid r_1 \mathrel{+}= \omega_{1,3} l_3 \mid \mathrm{Wr}(r_1)$
$\ldots$

Benefits of R2L:

- ⊕ Only linear Wr
- ⊕ Fewer turnarounds

Inverted Expander Graphs

Inverted Graph: R2L



$$r_i = \sum \omega_{i,j} \cdot l_j$$

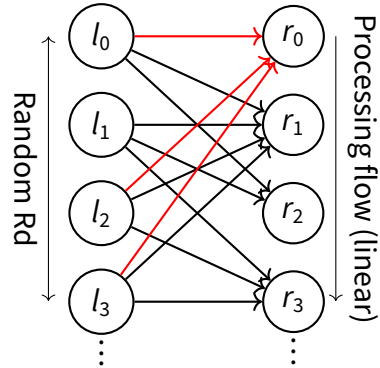
Step 1: $\text{Rd}(l_0) \mid r_0 \mathrel{+}= \omega_{0,0} l_0$
Step 2: $\text{Rd}(l_1) \mid r_0 \mathrel{+}= \omega_{0,1} l_1$
Step 3: $\text{Rd}(l_3) \mid r_0 \mathrel{+}= \omega_{0,3} l_3 \mid \text{Wr}(r_0)$
Step 4: $\text{Rd}(l_2) \mid r_1 \mathrel{+}= \omega_{1,2} l_2$
Step 5: $\text{Rd}(l_3) \mid r_1 \mathrel{+}= \omega_{1,3} l_3 \mid \text{Wr}(r_1)$
$\ldots$

Benefits of R2L:

- ⊕ Only linear Wr
- ⊕ Fewer turnarounds
- ⊕ No hazards

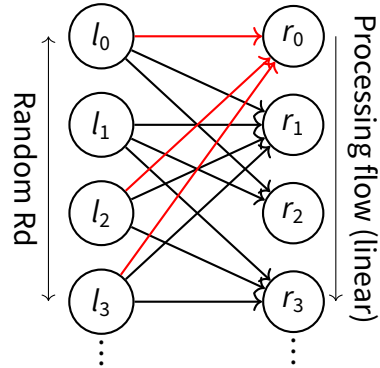
Inverted Expander Graphs: R2L

- R2L reduces off-chip memory accesses



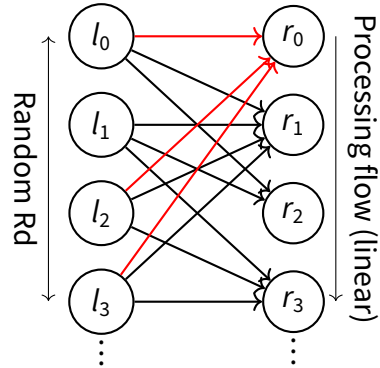
Inverted Expander Graphs: R2L

- R2L reduces off-chip memory accesses
- How to handle graph structure?
  - Randomly generated for L2R



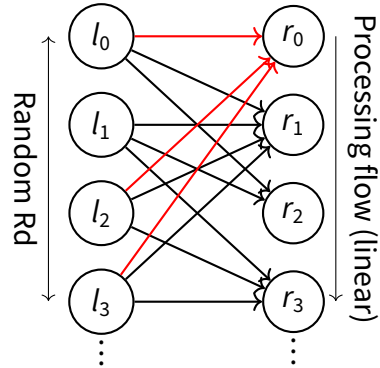
Inverted Expander Graphs: R2L

- R2L reduces off-chip memory accesses
- How to handle graph structure?
  - Randomly generated for L2R



Inverted Expander Graphs: R2L

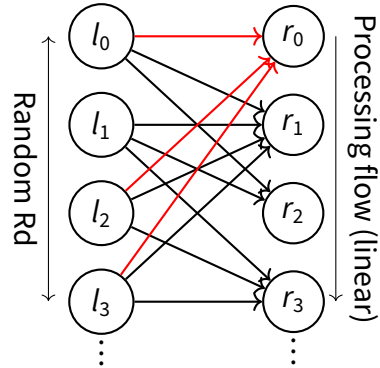
- R2L reduces off-chip memory accesses
- How to handle graph structure?
  - Randomly generated for L2R
  1. R2L with stored inverted graph



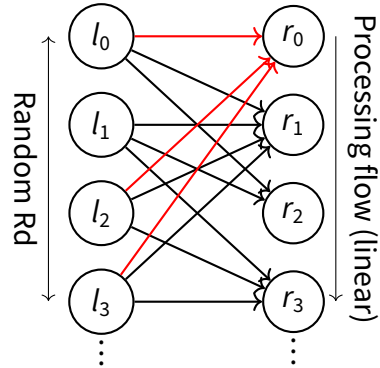
Inverted Expander Graphs: R2L

- R2L reduces off-chip memory accesses
- How to handle graph structure?
  - Randomly generated for L2R

1. R2L with stored inverted graph
2. R2L with generated graph + Postprocessing

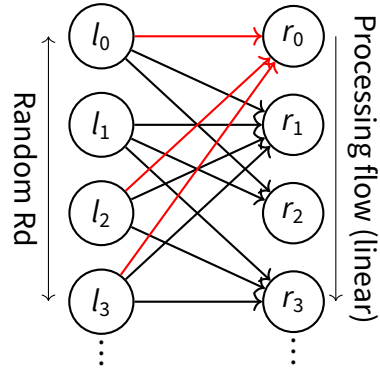


R2L: Stored Inverted Graph



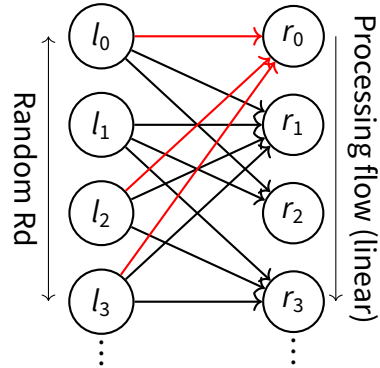
R2L: Stored Inverted Graph

1. Randomly generated in **left-node** major order $(l_0, r_0), (l_0, r_1), (l_0, r_2), (l_1, r_1), \ldots$



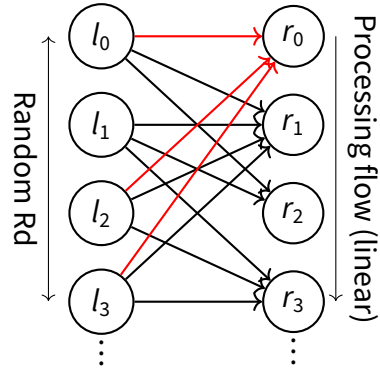
R2L: Stored Inverted Graph

1. Randomly generated in **left-node** major order $(l_0, r_0), (l_0, r_1), (l_0, r_2), (l_1, r_1), \ldots$
2. Transform the graph to **right-node** major order $(r_0, l_0), (r_0, l_2), (r_0, l_3), \ldots$



R2L: Stored Inverted Graph

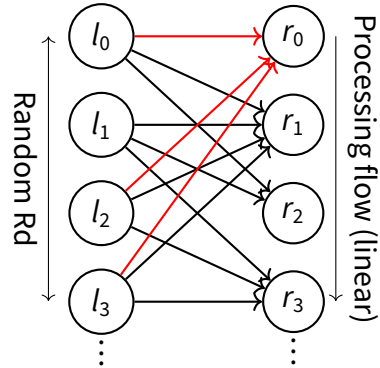
1. Randomly generated in **left-node** major order $(l_0, r_0), (l_0, r_1), (l_0, r_2), (l_1, r_1), \ldots$
2. Transform the graph to **right-node** major order $(r_0, l_0), (r_0, l_2), (r_0, l_3), \ldots$
3. Store transformed graph



R2L: Stored Inverted Graph

1. Randomly generated in **left-node** major order $(l_0, r_0), (l_0, r_1), (l_0, r_2), (l_1, r_1), \ldots$
2. Transform the graph to **right-node** major order $(r_0, l_0), (r_0, l_2), (r_0, l_3), \ldots$
3. Store transformed graph

⊕ Identical result

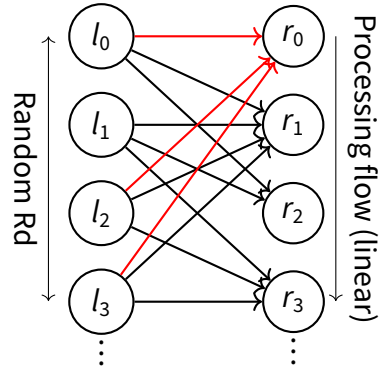


R2L: Stored Inverted Graph

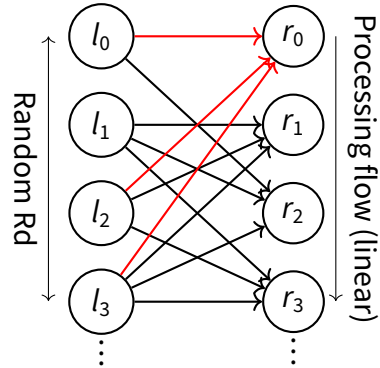
1. Randomly generated in **left-node** major order $(l_0, r_0), (l_0, r_1), (l_0, r_2), (l_1, r_1), \ldots$
2. Transform the graph to **right-node** major order $(r_0, l_0), (r_0, l_2), (r_0, l_3), \ldots$
3. Store transformed graph

⊕ Identical result

⊖ Must be stored in off-chip memory (Gigabytes!)

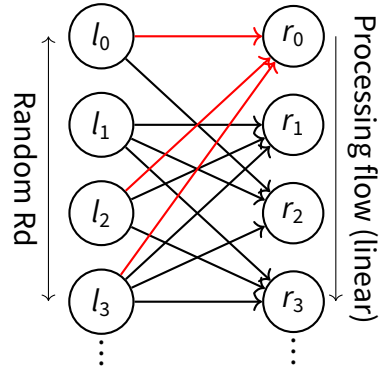


R2L: Generated Graph + Postprocessing



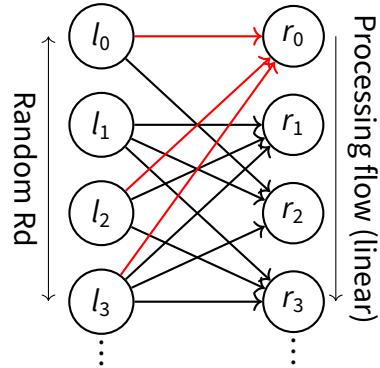
R2L: Generated Graph + Postprocessing

1. Fix right node degree



R2L: Generated Graph + Postprocessing

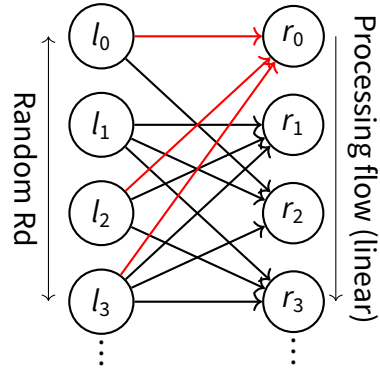
1. Fix right node degree
2. Randomly generated in **right-node** major order $(r_0, l_0), (r_0, l_1), (r_0, l_2), \ldots$



R2L: Generated Graph + Postprocessing

1. Fix right node degree
2. Randomly generated in **right-node** major order $(r_0, l_0), (r_0, l_1), (r_0, l_2), \ldots$

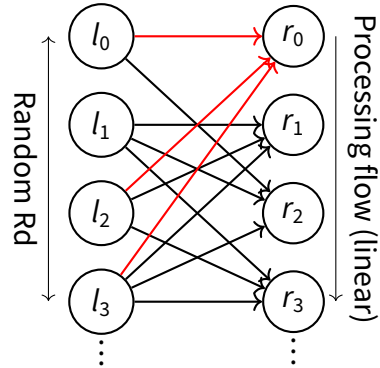
⊕ Generated using small PRNG



R2L: Generated Graph + Postprocessing

1. Fix right node degree
2. Randomly generated in **right-node** major order $(r_0, l_0), (r_0, l_1), (r_0, l_2), \ldots$

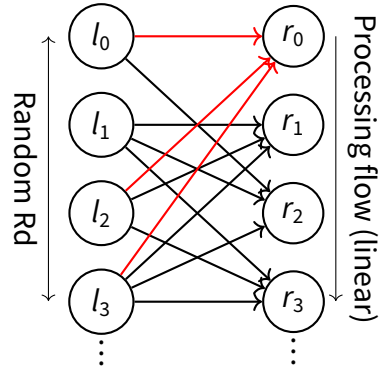
- ⊕ Generated using small PRNG
- ⊕ No off-chip graph memory!



R2L: Generated Graph + Postprocessing

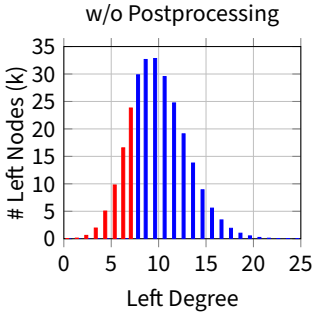
1. Fix right node degree
2. Randomly generated in **right-node** major order
   $(r_0, l_0), (r_0, l_1), (r_0, l_2), \ldots$

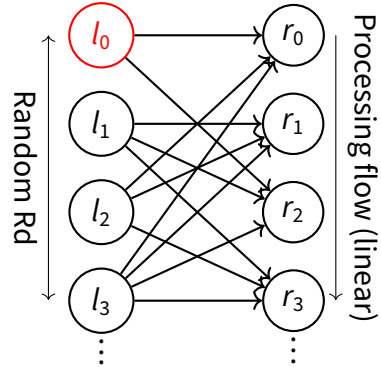
- ⊕ Generated using small PRNG
- ⊕ No off-chip graph memory!
- ? Security



R2L: Generated Graph + Postprocessing

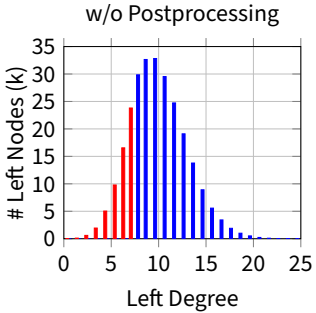
- Some l_i have low degree

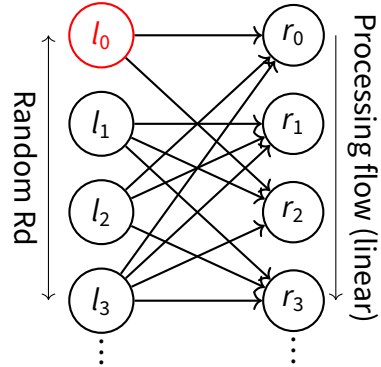




R2L: Generated Graph + Postprocessing

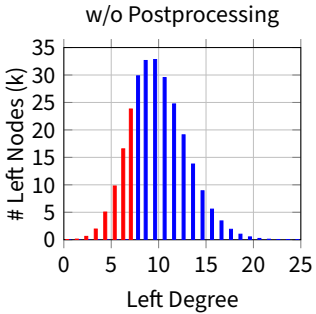
- Some l_i have low degree

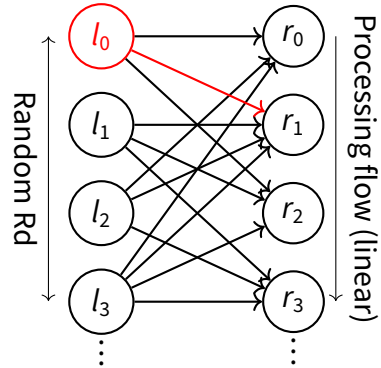




R2L: Generated Graph + Postprocessing

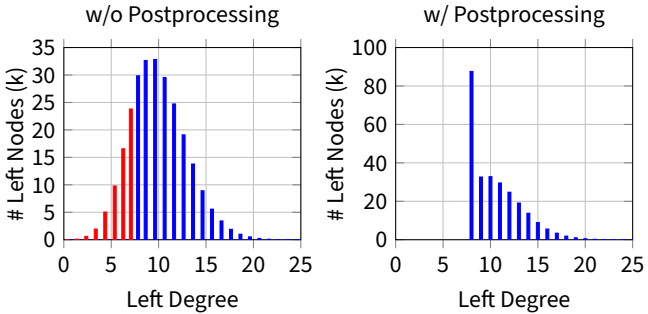
- Some l_i have low degree
- Add connections via "Postprocessing"
  - Store l_i in on-chip memory

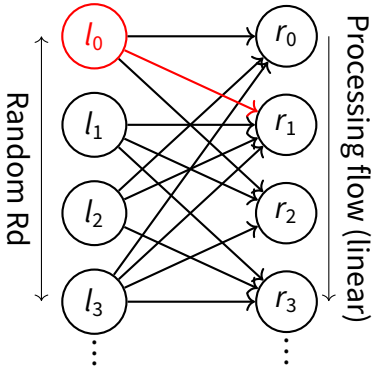




R2L: Generated Graph + Postprocessing

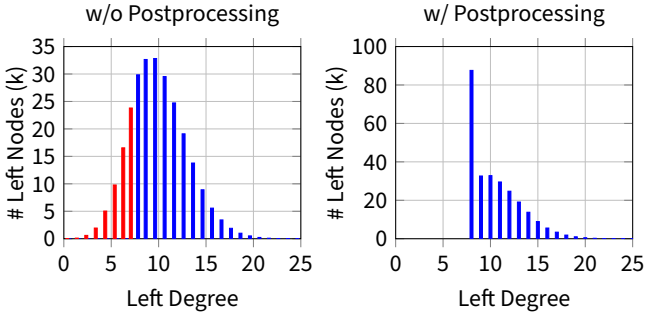
- Some l_i have low degree
- Add connections via "Postprocessing"
  - Store l_i in on-chip memory
- Each l_i has **at least** a certain degree

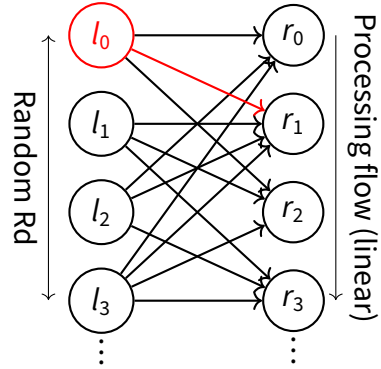




R2L: Generated Graph + Postprocessing

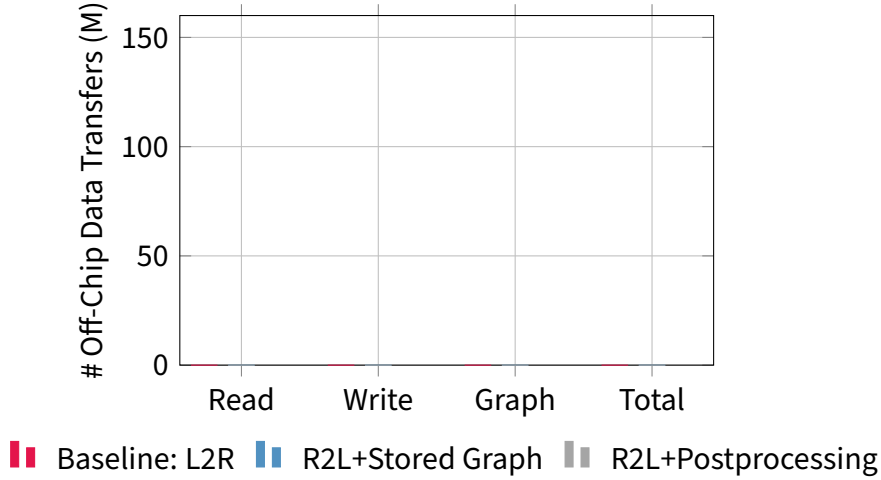
- Some l_i have low degree
- Add connections via "Postprocessing"
  - Store l_i in on-chip memory
- Each l_i has **at least** a certain degree



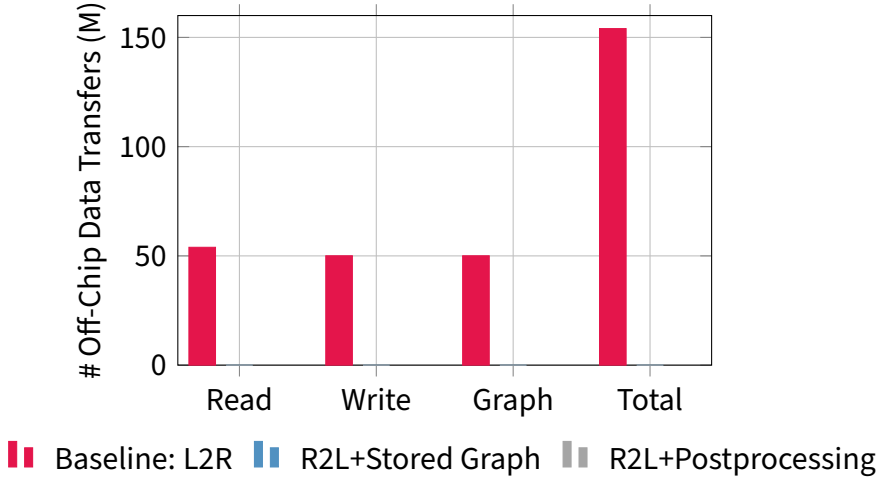


☑ Security Proof in Paper

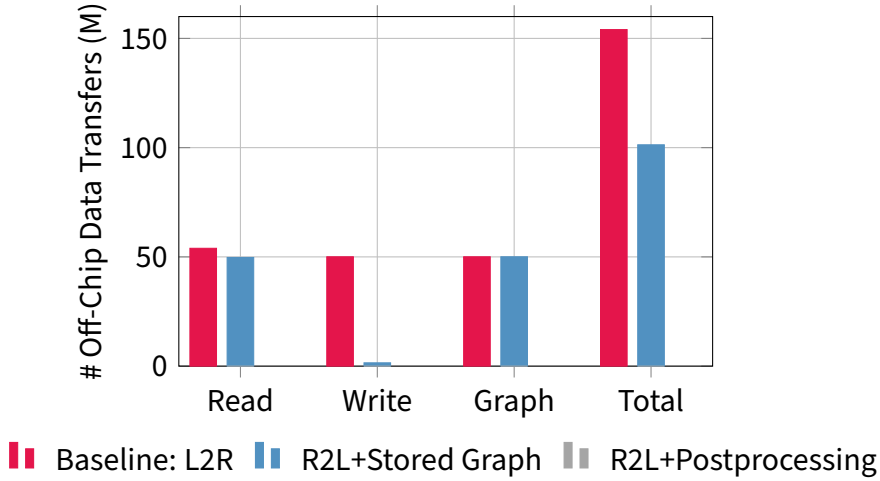
Off-Chip Data Transfers



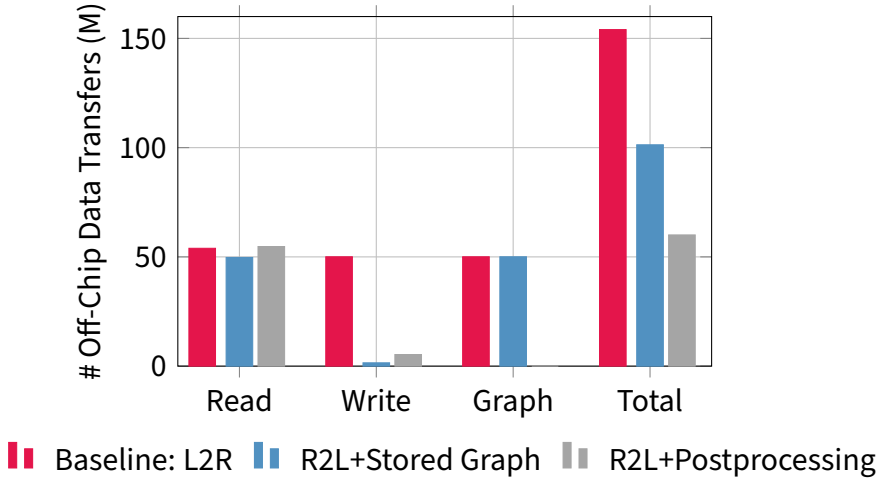
Off-Chip Data Transfers



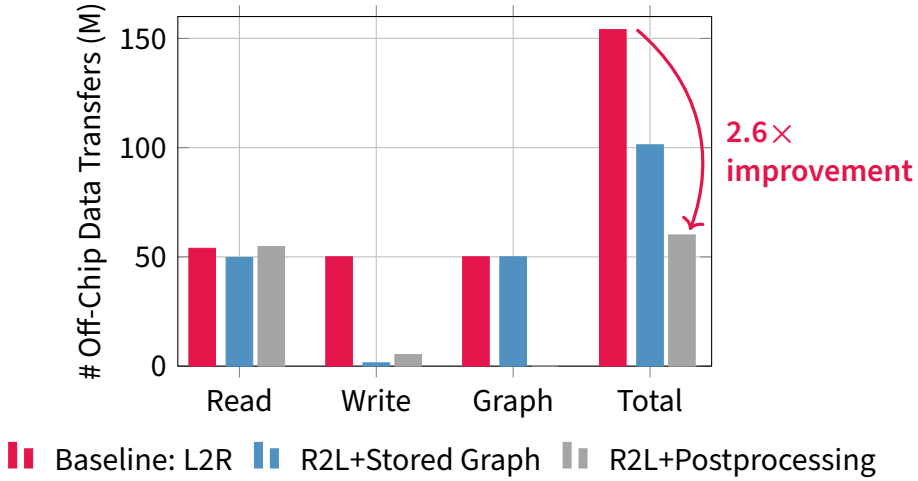
Off-Chip Data Transfers



Off-Chip Data Transfers

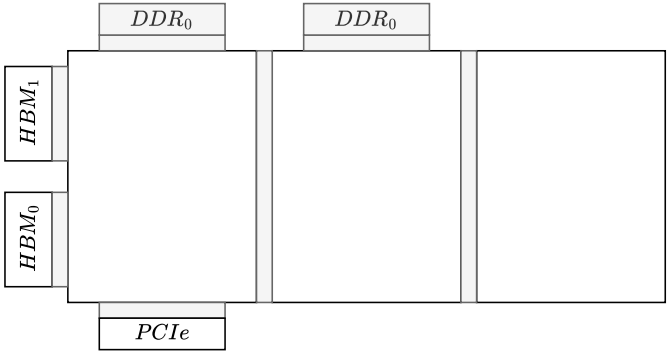


Off-Chip Data Transfers



Hardware Architecture

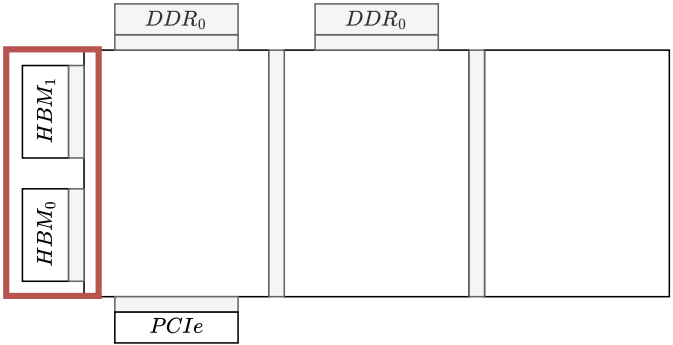
Hardware Architecture and Unit Placement



Alveo U280 FPGA

- HBM2 (460 GB/s)
- DDR4 (2x19 GB/s)
- 3 x SLR-Regions
- 20K SLL Lines à SLR

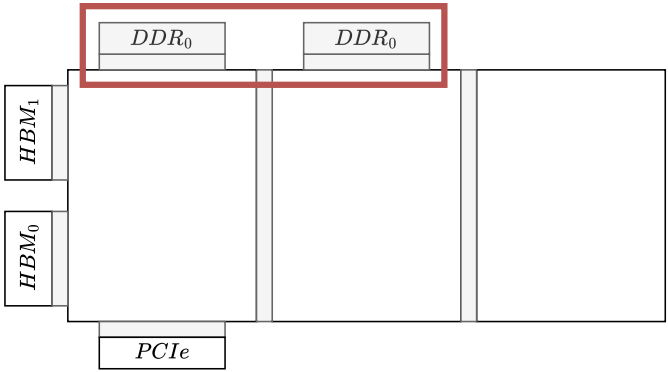
Hardware Architecture and Unit Placement



Alveo U280 FPGA

- HBM2 (460 GB/s)
- DDR4 (2x19 GB/s)
- 3 x SLR-Regions
- 20K SLL Lines à SLR

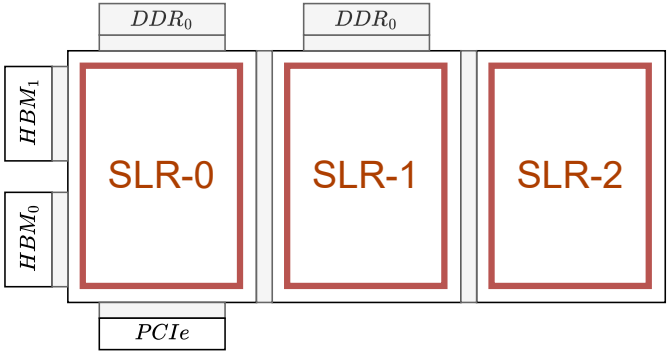
Hardware Architecture and Unit Placement



Alveo U280 FPGA

- HBM2 (460 GB/s)
- DDR4 (2x19 GB/s)
- 3 x SLR-Regions
- 20K SLL Lines à SLR

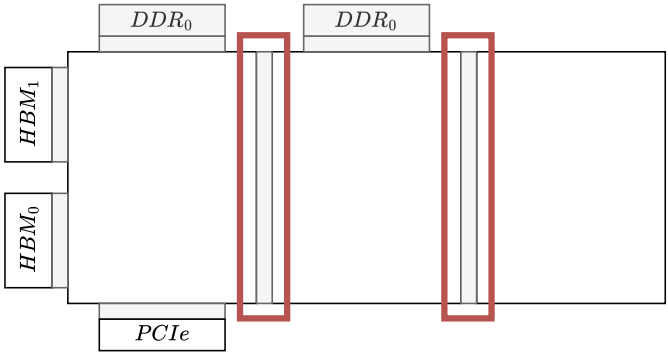
Hardware Architecture and Unit Placement



Alveo U280 FPGA

- HBM2 (460 GB/s)
- DDR4 (2x19 GB/s)
- 3 x SLR-Regions
- 20K SLL Lines à SLR

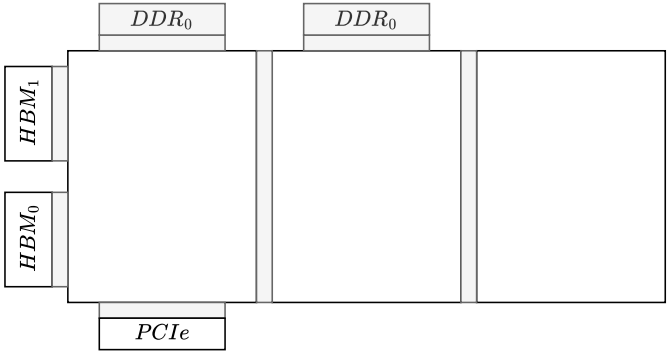
Hardware Architecture and Unit Placement



Alveo U280 FPGA

- HBM2 (460 GB/s)
- DDR4 (2x19 GB/s)
- 3 x SLR-Regions
- 20K SLL Lines à SLR

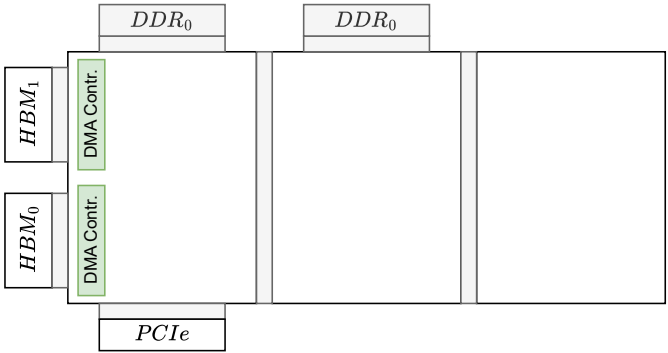
Hardware Architecture and Unit Placement



Alveo U280 FPGA

- HBM2 (460 GB/s)
- DDR4 (2x19 GB/s)
- 3 x SLR-Regions
- 20K SLL Lines à SLR

Hardware Architecture and Unit Placement

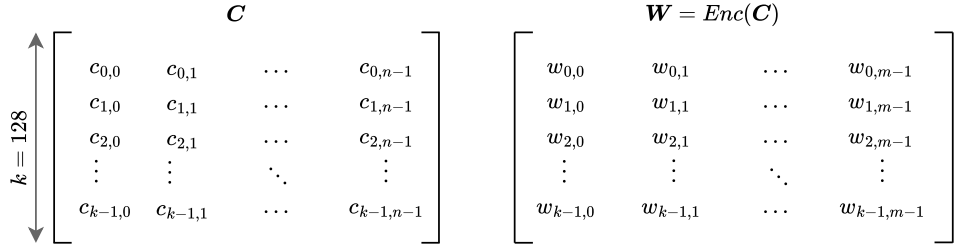


Alveo U280 FPGA

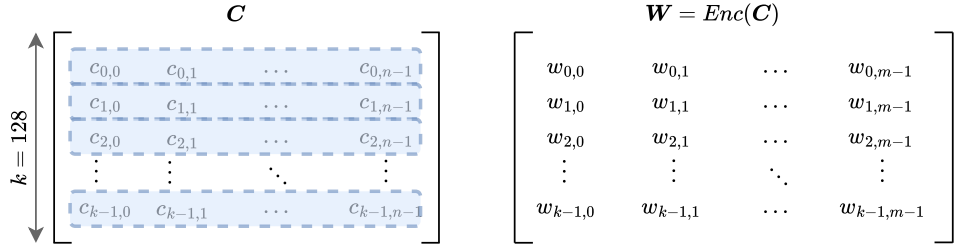
- HBM2 (460 GB/s)
- DDR4 (2x19 GB/s)
- 3 x SLR-Regions
- 20K SLL Lines à SLR

→ Requires synchronization of 32 AXI-MM and 32 AXI-ST interfaces
→ Data movement of 16K bits per cycle @ 200 MHz

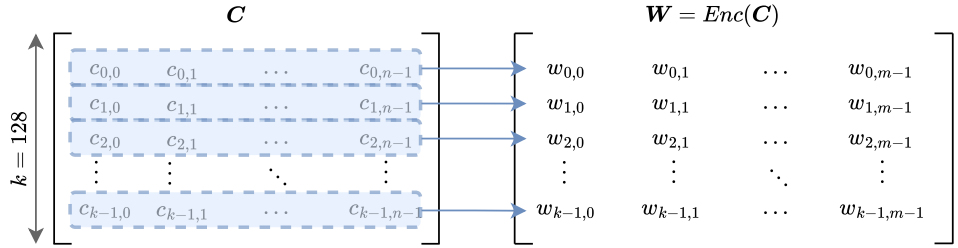
Hardware Architecture and Unit Placement



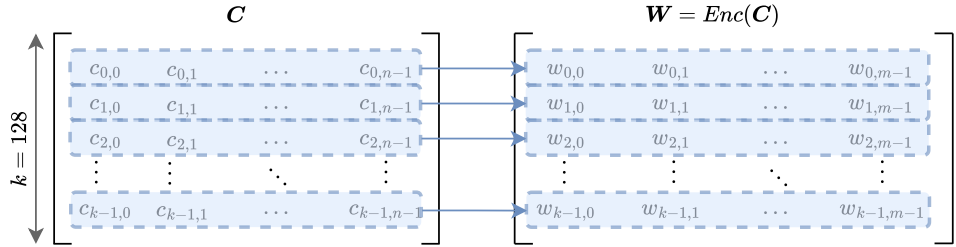
Hardware Architecture and Unit Placement



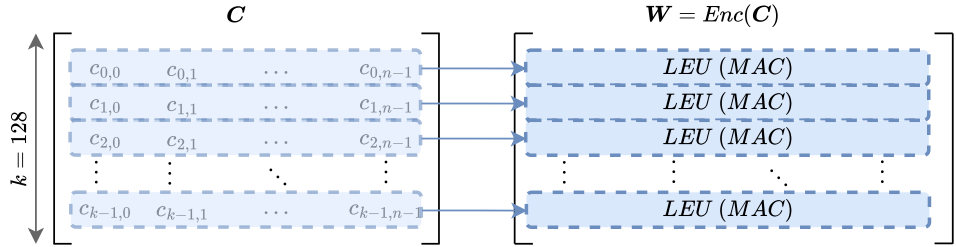
Hardware Architecture and Unit Placement



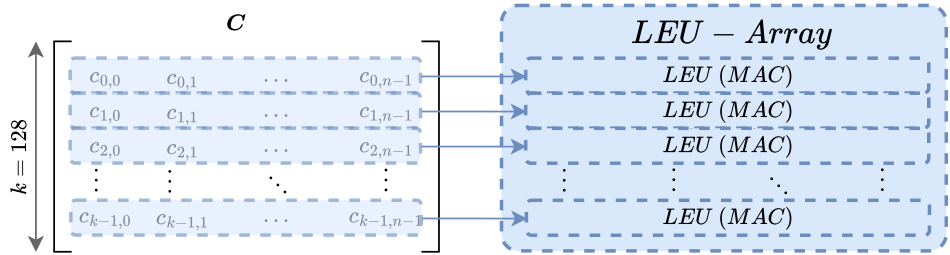
Hardware Architecture and Unit Placement



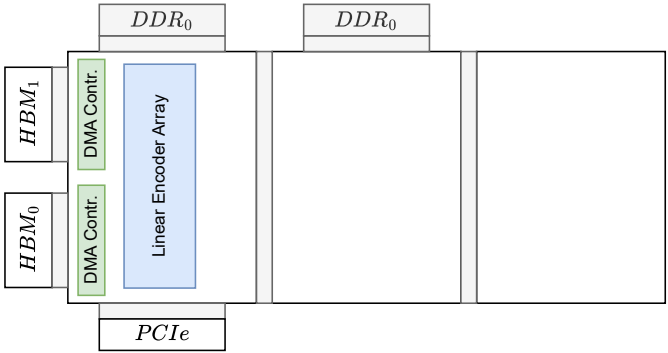
Hardware Architecture and Unit Placement



Hardware Architecture and Unit Placement



Hardware Architecture and Unit Placement

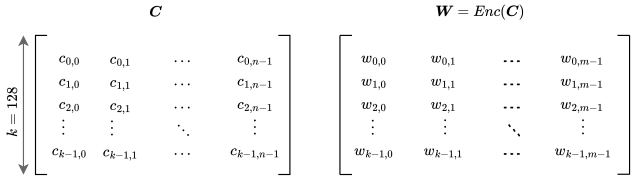


Alveo U280 FPGA

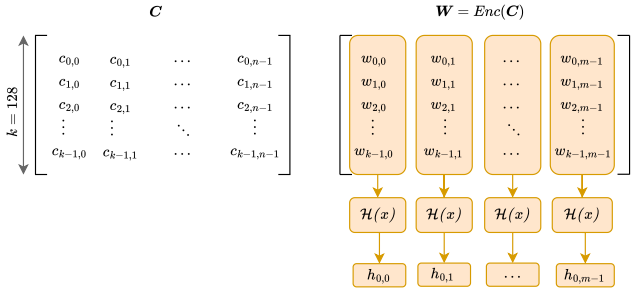
- HBM2 (460 GB/s)
- DDR4 (2x19 GB/s)
- 3 x SLR-Regions
- 20K SLL Lines à SLR

→ Requires 32K IO lines (16K input and 16K output)

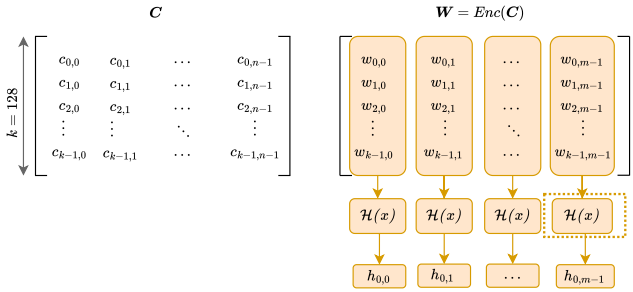
Hardware Architecture and Unit Placement



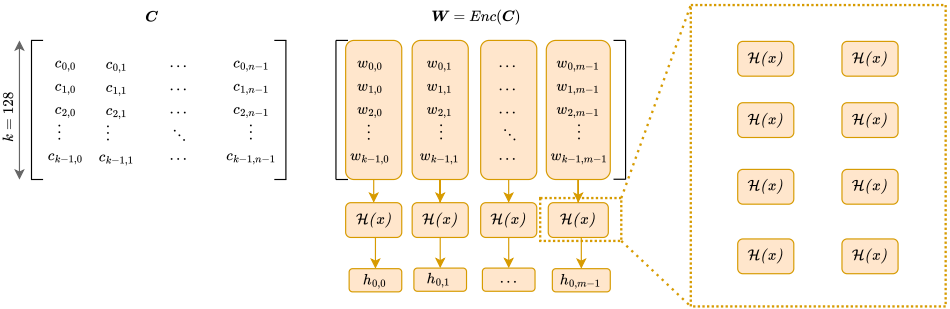
Hardware Architecture and Unit Placement



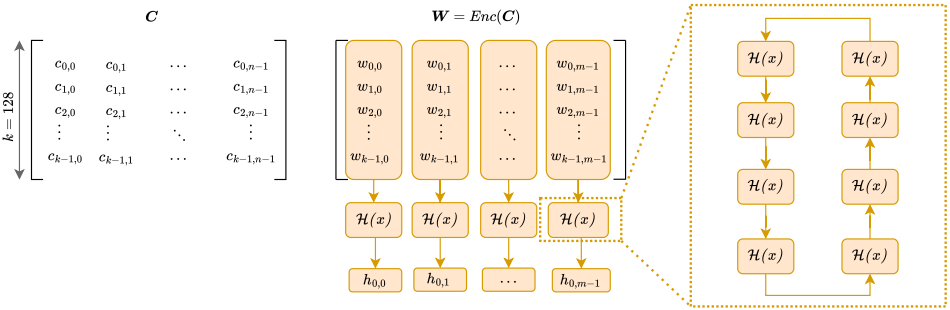
Hardware Architecture and Unit Placement



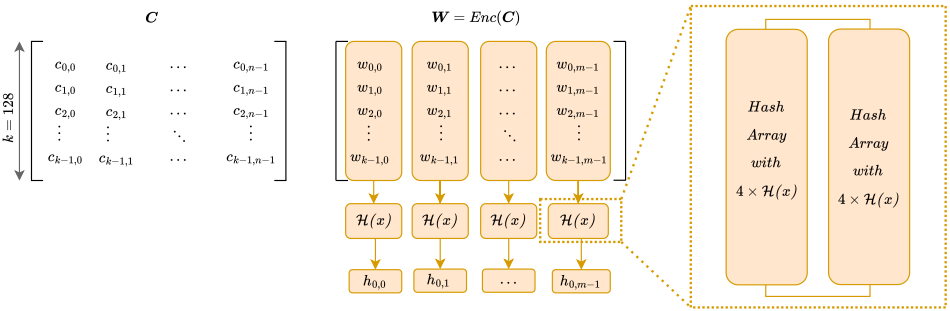
Hardware Architecture and Unit Placement



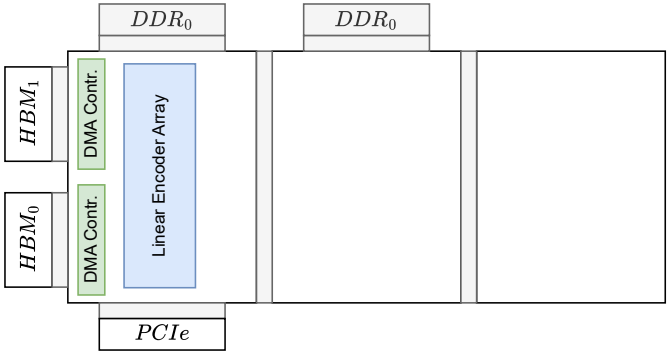
Hardware Architecture and Unit Placement



Hardware Architecture and Unit Placement



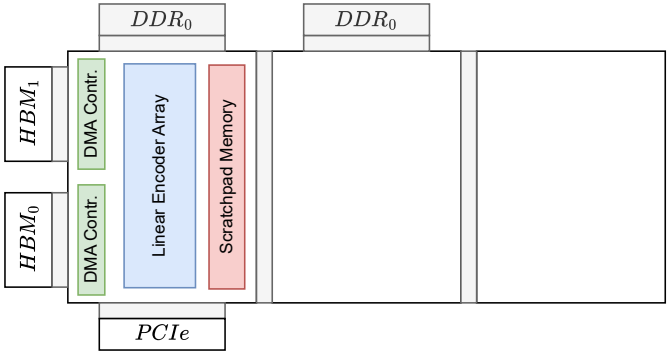
Hardware Architecture and Unit Placement



Alveo U280 FPGA

- HBM2 (460 GB/s)
- DDR4 (2x19 GB/s)
- 3 x SLR-Regions
- 20K SLL Lines à SLR

Hardware Architecture and Unit Placement

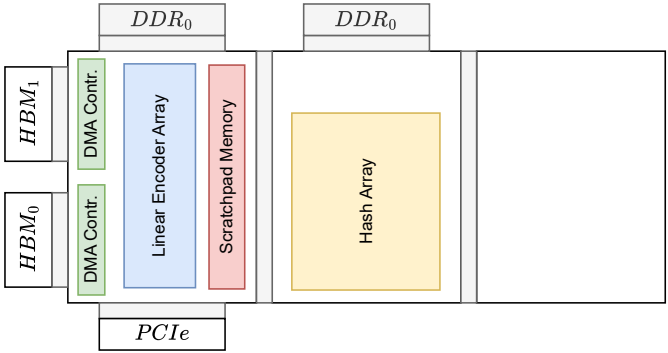


Alveo U280 FPGA

- HBM2 (460 GB/s)
- DDR4 (2x19 GB/s)
- 3 x SLR-Regions
- 20K SLL Lines à SLR

→ Scratchpad reshapes data from 512x32 to 1088x16 (muxed to 1088x8)

Hardware Architecture and Unit Placement

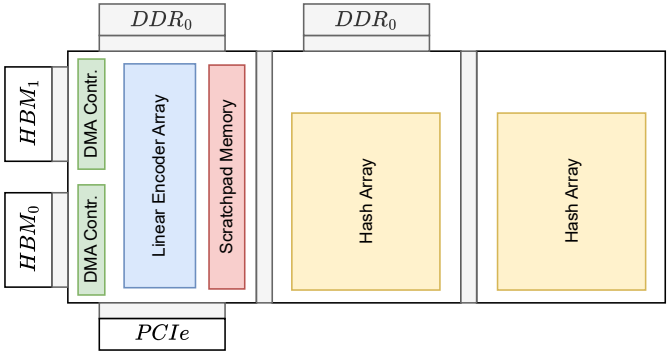


Alveo U280 FPGA

- HBM2 (460 GB/s)
- DDR4 (2x19 GB/s)
- 3 x SLR-Regions
- 20K SLL Lines à SLR

→ Hash Array requires a total of 12K (1088x8 + 1600x2)

Hardware Architecture and Unit Placement

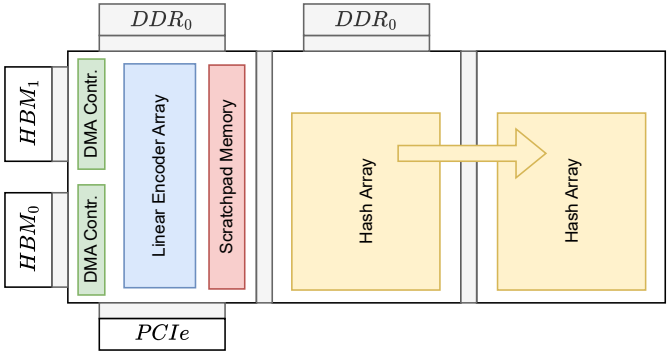


Alveo U280 FPGA

- HBM2 (460 GB/s)
- DDR4 (2x19 GB/s)
- 3 x SLR-Regions
- 20K SLL Lines à SLR

→ Hash Array requires a total of 12K (1088x8 + 1600x2)

Hardware Architecture and Unit Placement

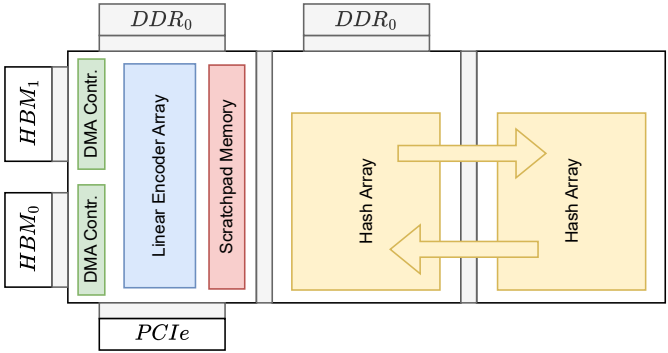


Alveo U280 FPGA

- HBM2 (460 GB/s)
- DDR4 (2x19 GB/s)
- 3 x SLR-Regions
- 20K SLL Lines à SLR

→ Hash Array requires a total of 12K (1088x8 + 1600x2)

Hardware Architecture and Unit Placement

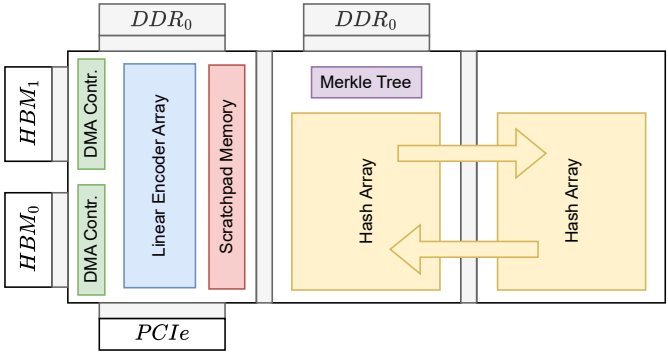


Alveo U280 FPGA

- HBM2 (460 GB/s)
- DDR4 (2x19 GB/s)
- 3 x SLR-Regions
- 20K SLL Lines à SLR

→ Hash Array requires a total of 12K (1088x8 + 1600x2)

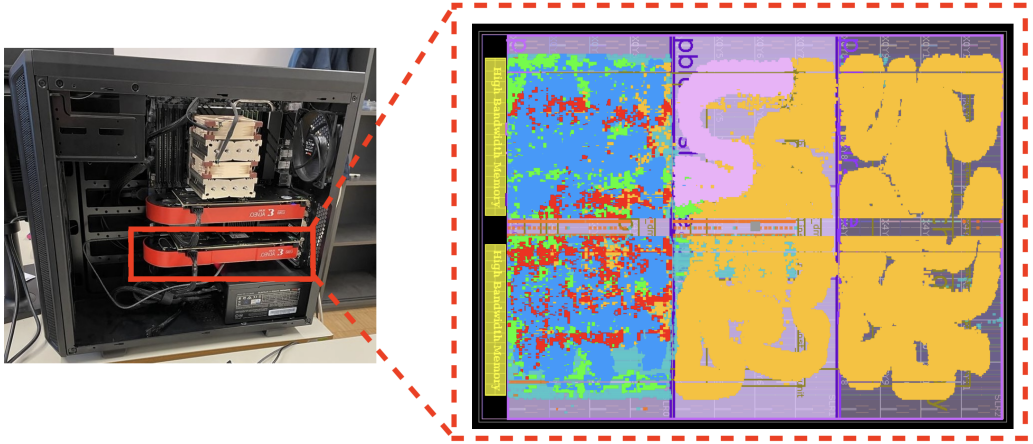
Hardware Architecture and Unit Placement



Alveo U280 FPGA

- HBM2 (460 GB/s)
- DDR4 (2x19 GB/s)
- 3 x SLR-Regions
- 20K SLL Lines à SLR

Hardware Architecture and Unit Placement



Speedup Results over Software

- **Commitment: up to 264×**

  Linear Encode: up to 231×
  Leaf Hash: up to 1,015×
  Merkle Tree: up to 21×

Speedup Results over Software

- **Commitment: up to 264×**

  Linear Encode: up to 231×
  Leaf Hash: up to 1,015×
  Merkle Tree: up to 21×

- **Proving: up to 65×**

Conclusion

- Multi-SLR architecture with HBM memory
- First FPGA accelerator for PCS with linear prover time
- Inverted expander graphs + Proof
- Reducing off-chip memory accesses

TU Graz
SCIENCE
PASSION
TECHNOLOGY

Accelerating Hash-Based Polynomial Commitment Schemes with Linear Prover Time

Florian Hirner **Florian Krieger** Constantin Piber Sujoy Sinha Roy

Graz University of Technology

CHES 2025 Conference

› isec.tugraz.at

Demo

Bonus Slides

Spielman Encoding