Hardware Fixed-Point 2D and 3D norms

Romain Bouarah
Florent de Dinechin
INSA Lyon - Inria, CITI (UR3720), 69621 Villeurbanne, France
{romain.bouarah, florent.de-dinechin}@insa-lyon.fr

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON
Inria
citi lab

2025-05-05

Introduction

Motivation

The 3-dimensional Euclidean norm is defined as

$$\left\|(X, Y, Z)\right\|_2 = \sqrt{X^2 + Y^2 + Z^2}$$

Motivation

The 3-dimensional Euclidean norm is defined as

$$\left\|(X, Y, Z)\right\|_2 = \sqrt{X^2 + Y^2 + Z^2}$$

The Euclidean norm is used in:

- **3D graphics**: Vector normalization
- **Scientific Computing**: Coulomb's law, universal gravitation
- **Signal processing**: Cartesian to polar

Motivation

The 3-dimensional Euclidean norm is defined as

$$\left\|(X, Y, Z)\right\|_2 = \sqrt{X^2 + Y^2 + Z^2}$$

The Euclidean norm is used in:

- **3D graphics**: Vector normalization
- **Scientific Computing**: Coulomb's law, universal gravitation
- **Signal processing**: Cartesian to polar

Motivation for a hardware implementation:

- An industrial application under NDA that sought our expertise
- Relevant for FPGA: Application-specific arithmetic

Specifications

Notice that

$$\forall s \in \mathbb{R}^+ \quad \sqrt{(sX)^2 + (sY)^2 + (sZ)^2} = s\sqrt{X^2 + Y^2 + Z^2}$$

Specifications

Notice that

$$\forall s \in \mathbb{R}^+ \quad \sqrt{(sX)^2 + (sY)^2 + (sZ)^2} = s\sqrt{X^2 + Y^2 + Z^2}$$

Without loss of generality, we can assume $X, Y, Z \in [-1, 1)$.

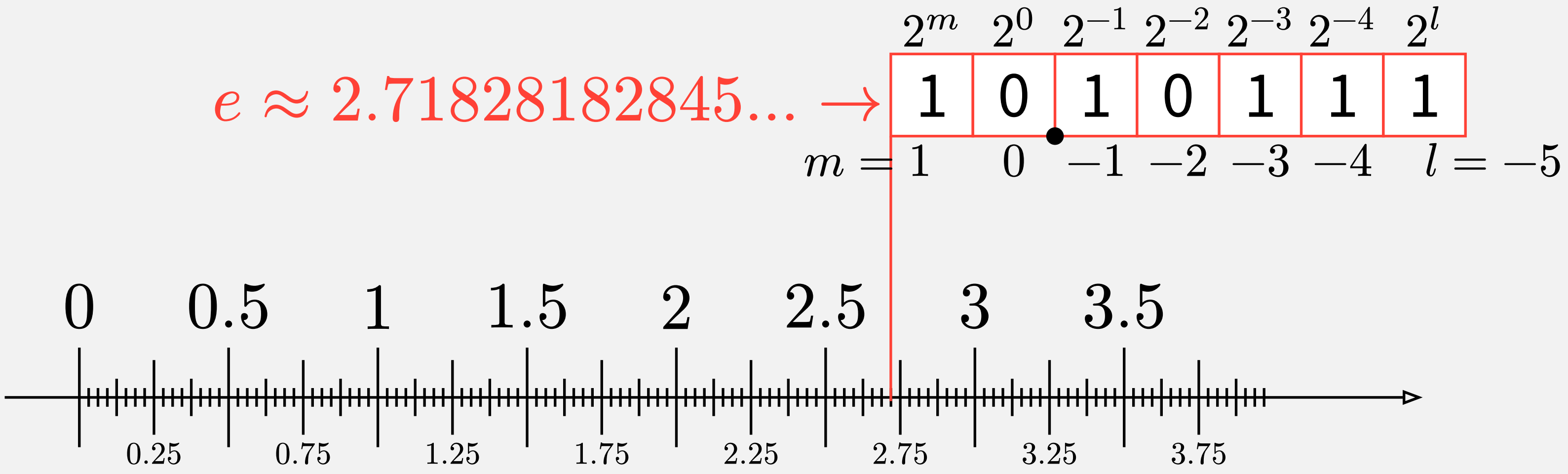
- Inputs are fixed-point values in $[-1, 1)$
- Output is a fixed-point value in $\left[0, \sqrt{2}\right]$ for 2D, and $\left[0, \sqrt{3}\right]$ for 3D.
- Output must be faithfully rounded

Unsigned fixed-point

For MSB position $m \in \mathbb{Z}$ and LSB position $\ell \in \mathbb{Z}$ s.t. $m > \ell$:

$$\mathrm{ufix}(m, \ell) = \left\{ \sum_{i=\ell}^{m} 2^i \cdot x_i \mid x_i \in \{0, 1\} \right\}$$

Real axis with tick marks in $\mathrm{ufix}(1, -5)$:



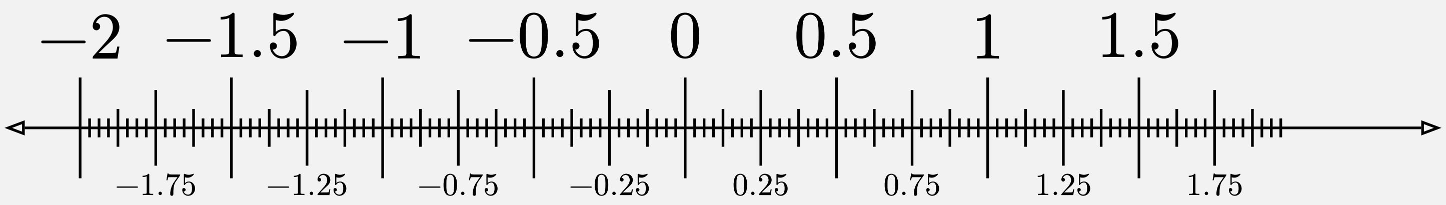
Signed (2's complement) fixed-point

Using two's complement negative values can be represented.

For MSB position $m \in \mathbb{Z}$ and LSB position $\ell \in \mathbb{Z}$ s.t. $m > \ell$:

$$\mathrm{sfix}(m, \ell) = \left\{ -2^m \cdot x_m + \sum_{i=l}^{m-1} 2^i \cdot x_i \mid x_i \in \{0, 1\} \right\}$$

$\mathrm{sfix}(1, -5)$:



Detailed specifications

Inputs X, Y and Z are fixed-point values in $[-1, 1)$

$\Rightarrow \ \mathrm{sfix}(0, \ell_{\mathrm{in}})$

Output R is a fixed-point value in $\left[0, \sqrt{2}\right]$ for 2D, and $\left[0, \sqrt{3}\right]$ for 3D.

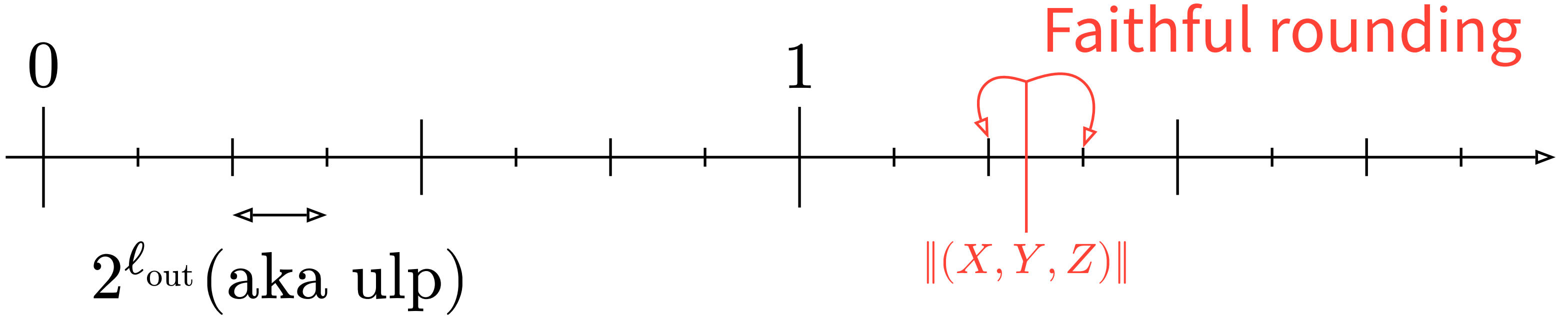
$\Rightarrow \ \mathrm{ufix}(0, \ell_{\mathrm{out}})$ (for 2D and 3D)

Detailed specifications

Inputs X, Y and Z are fixed-point values in $[-1, 1)$
$\Rightarrow \; \mathrm{sfix}(0, \ell_{\mathrm{in}})$

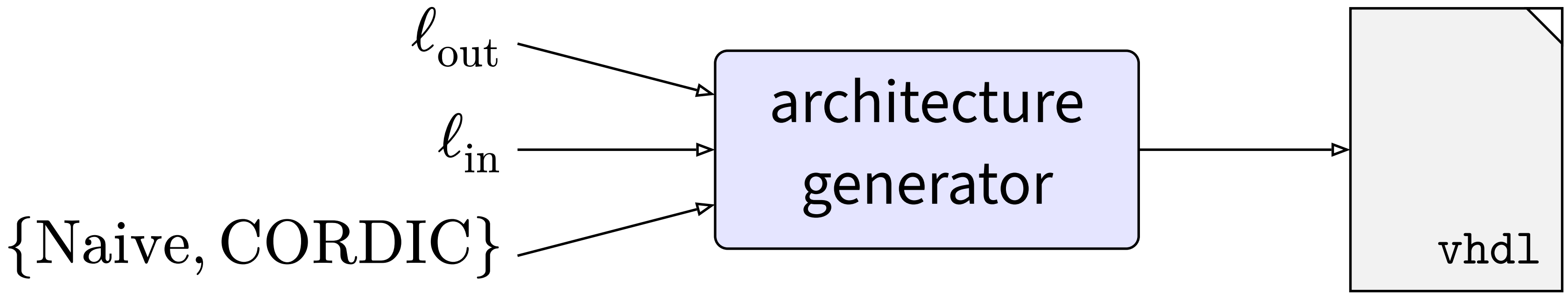
Output R is a fixed-point value in $\left[0, \sqrt{2}\right]$ for 2D, and $\left[0, \sqrt{3}\right]$ for 3D.
$\Rightarrow \; \mathrm{ufix}(0, \ell_{\mathrm{out}})$ (for 2D and 3D)

Output must be faithfully rounded: $\left| R - \|(X, Y, Z)\| \right| < 2^{\ell_{\mathrm{out}}}$



Our contributions

- A generator of circuits computing the norm:
  - Produces **faithful** circuits
  - Parameterized by ℓ_{out} and ℓ_{in}
  - Supports two methods: Naive and CORDIC
  - Integrated into the **FloPoCo**[1] project



[1] www.flopoco.org

Our contributions

- A generator of circuits computing the norm:
  - Produces **faithful** circuits
  - Parameterized by ℓ_{out} and ℓ_{in}
  - Supports two methods: Naive and CORDIC
  - Integrated into the **FloPoCo**[1] project
- Circuits do not compute useless bits
  - Optimized through **error analysis**

[1] www.flopoco.org

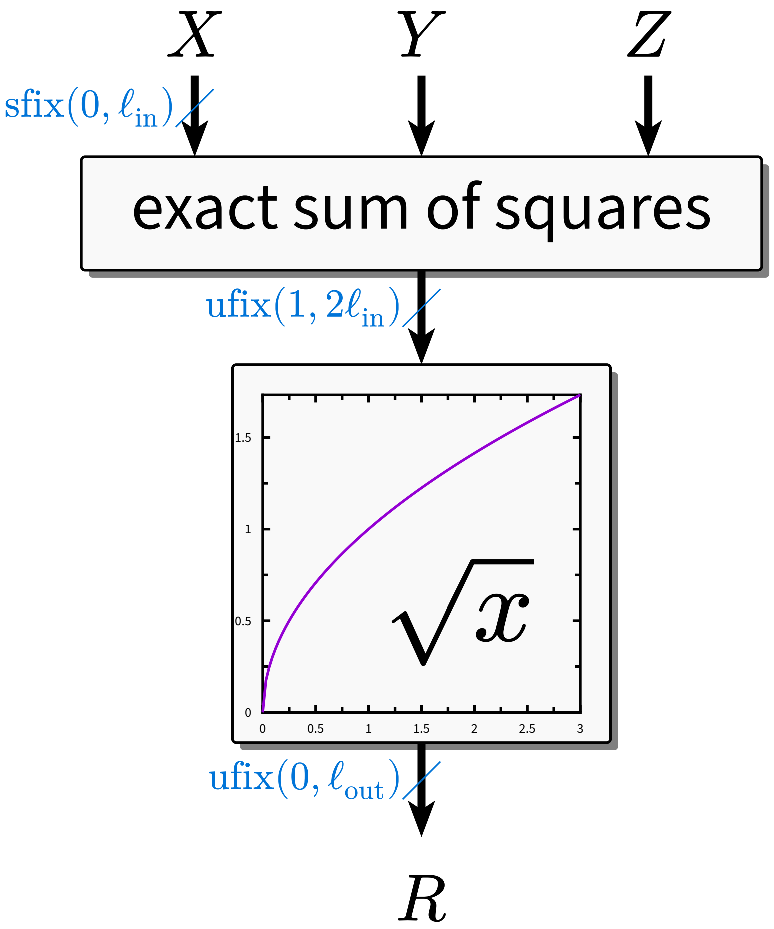
Our contributions

- A generator of circuits computing the norm:
  - Produces **faithful** circuits
  - Parameterized by ℓ_{out} and ℓ_{in}
  - Supports two methods: Naive and CORDIC
  - Integrated into the **FloPoCo**[1] project
- Circuits do not compute useless bits
  - Optimized through **error analysis**
- Comparative evaluation of these methods for FPGA:
  - Analysis of relevance domains

[1] www.flopoco.org

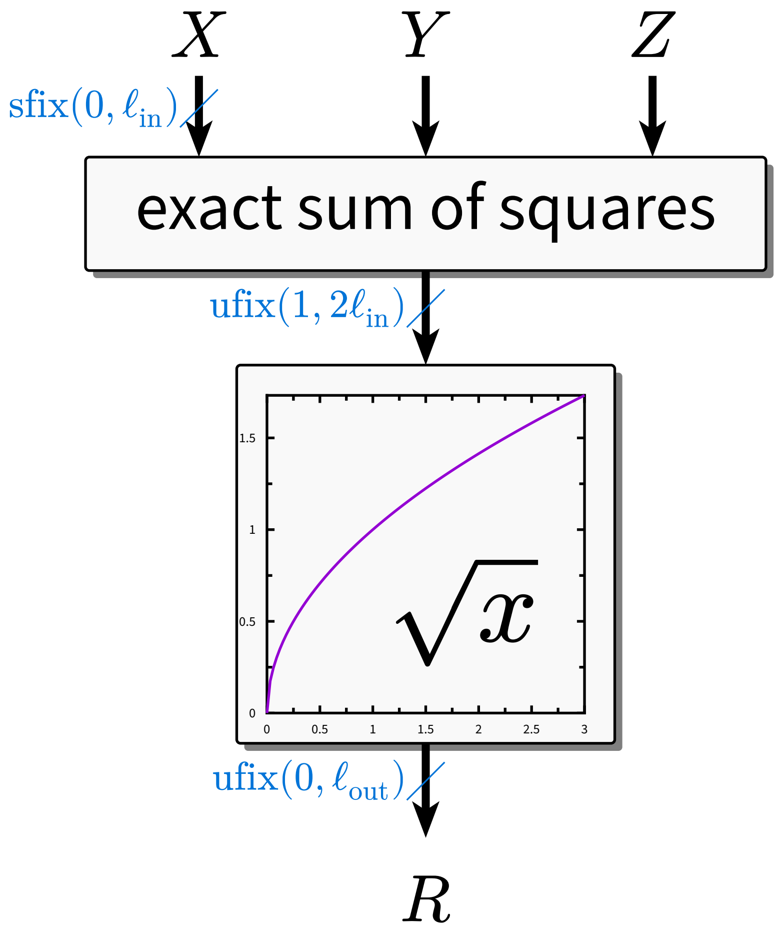
Naive architecture for 2D and 3D

A truly naive architecture



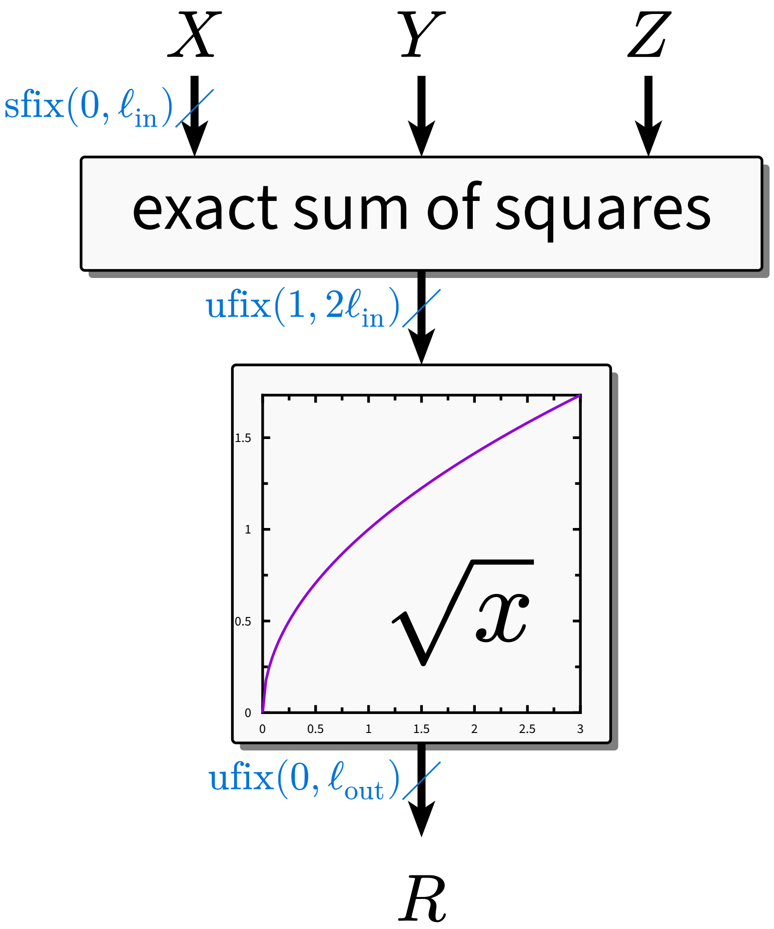
- Exact sum of squares
  - $S = X^2 + Y^2 (+Z^2)$
  - 2D: $S \in [0, 2]$
  - 3D: $S \in [0, 3]$
  - format is $\mathrm{ufix}(1, 2\ell_{\mathrm{in}})$

A truly naive architecture



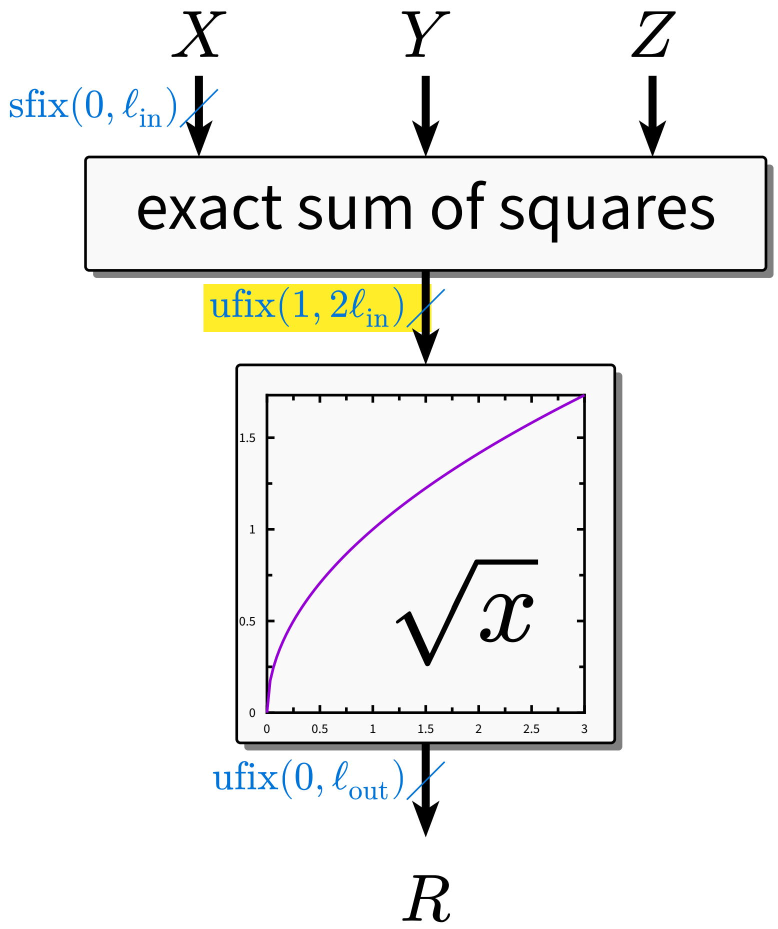
- Exact sum of squares
  - $S = X^2 + Y^2 (+Z^2)$
  - 2D: $S \in [0, 2]$
  - 3D: $S \in [0, 3]$
  - format is $\mathrm{ufix}(1, 2\ell_{\mathrm{in}})$
- $\sqrt{x}$ as a tabulated function
  - correctly/faithfully rounded

Problems with this architecture



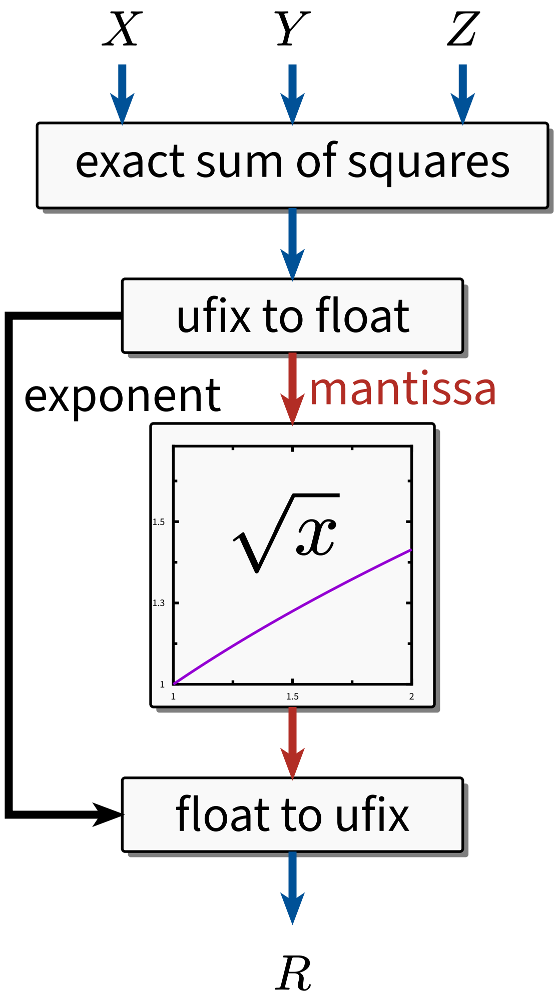
- Problems:
  1. $\sqrt{\cdot}$ is not derivable in $0 \Rightarrow$ prevents:
     - Polynomial approximation
     - Multipartite approximation

Problems with this architecture



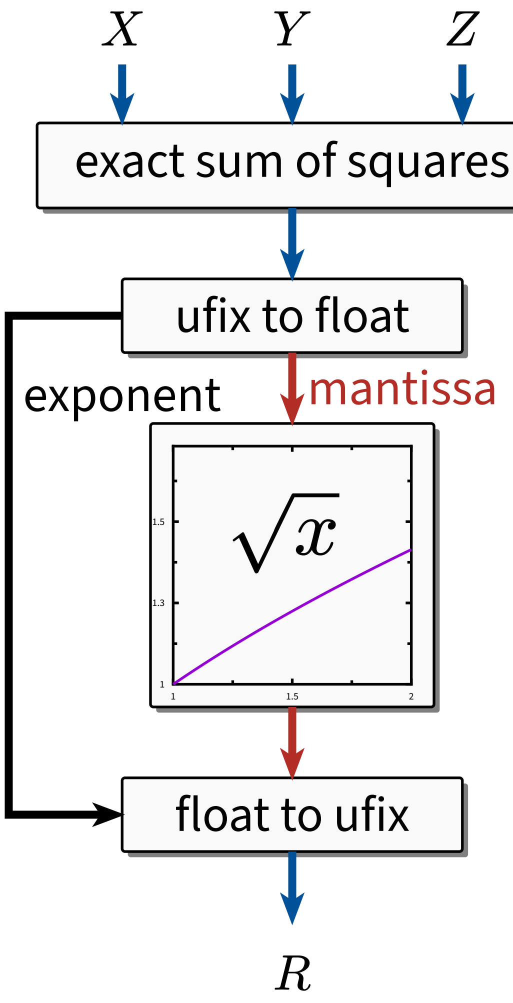
- Problems:
  1. $\sqrt{\cdot}$ is not derivable in $0 \Rightarrow$ prevents:
     - Polynomial approximation
     - Multipartite approximation
  2. Table size grows exponentially with twice the input size
     - Truncation causes errors for small input values

Overview of the new architecture



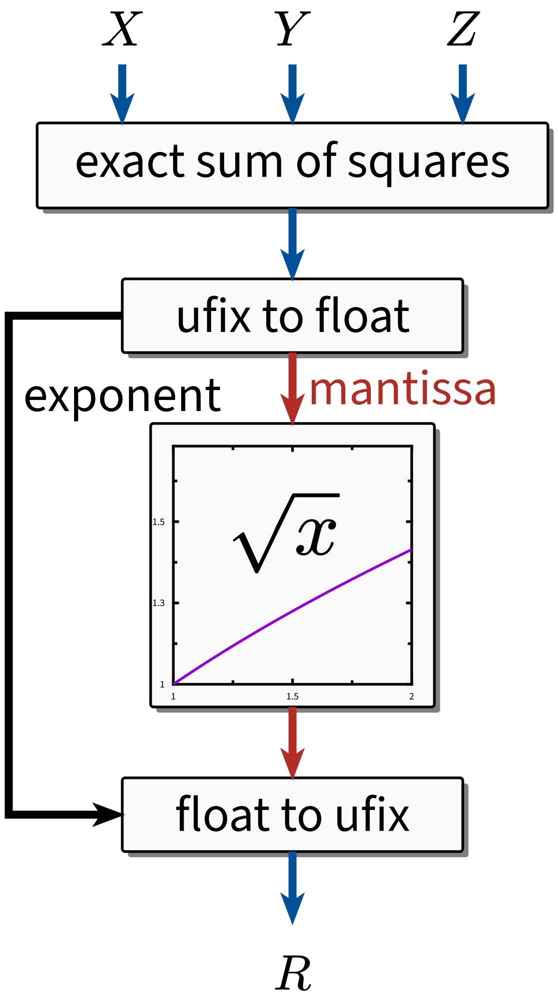
- Solution:
  - Use an internal floating point format after the sum of squares

Overview of the new architecture



- Solution:
  - Use an internal floating point format after the sum of squares
- $S \in [1, 2)$

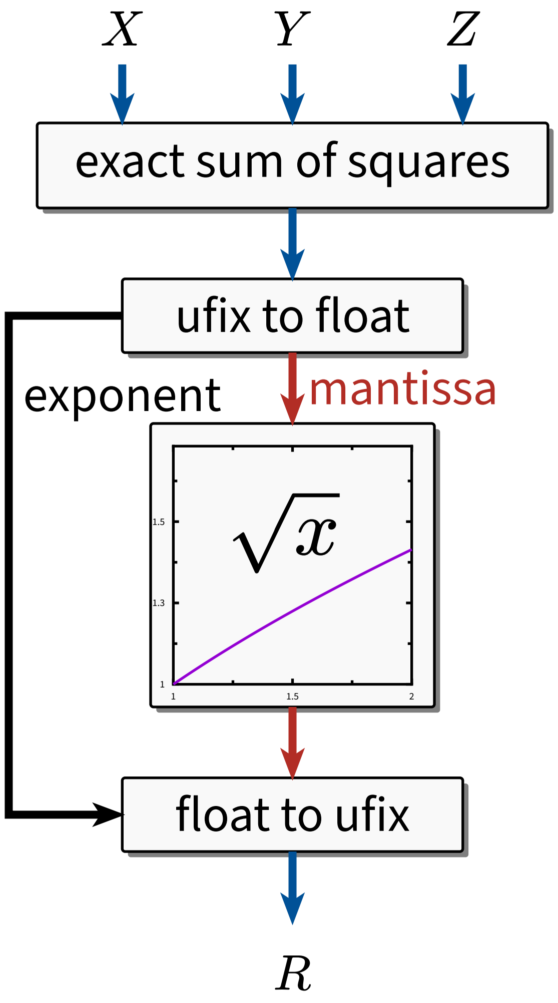
Overview of the new architecture



- Solution:
  - Use an internal floating point format after the sum of squares
- $S \in [1, 2)$

1. $\sqrt{\cdot}$ derivable on this domain. Can use:
   - Polynomial approximation
   - Multipartite approximation

Overview of the new architecture



- Solution:
  - Use an internal floating point format after the sum of squares
- $S \in [1, 2)$

1. $\sqrt{\cdot}$ derivable on this domain. Can use:
   - Polynomial approximation
   - Multipartite approximation
2. The mantissa is smaller (in bits) than the sum

Internal floating point format

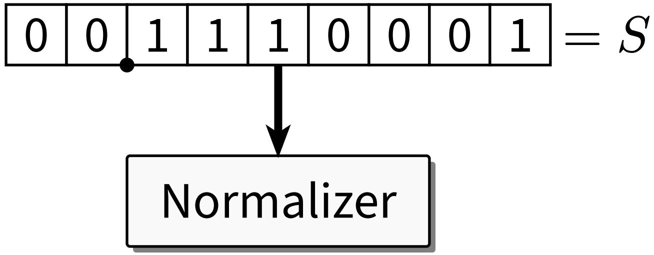
The normalizer is a combined **leading zero counter** and **shifter**. Outputs:

- L (exponent): leading zero count in the binary writing of the input
- $n.F$ (mantissa): shifted input where $n \in \{0, 1\}$ and $F \in [0, 1)$

Internal floating point format

The normalizer is a combined **leading zero counter** and **shifter**. Outputs:

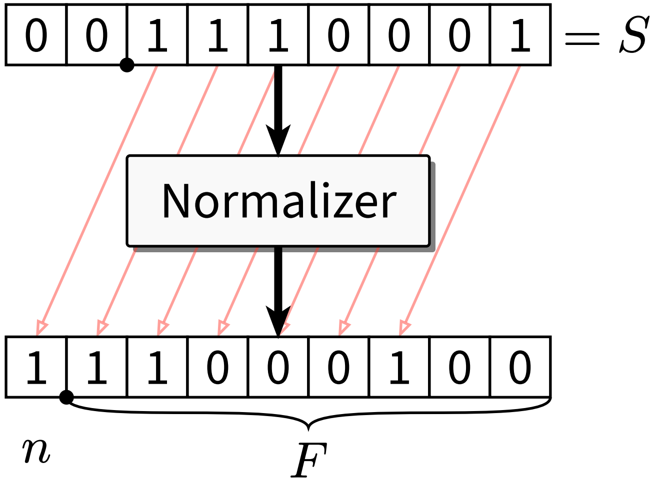
- L (exponent): leading zero count in the binary writing of the input
- $n.F$ (mantissa): shifted input where $n \in \{0, 1\}$ and $F \in [0, 1)$



Internal floating point format

The normalizer is a combined **leading zero counter** and **shifter**. Outputs:

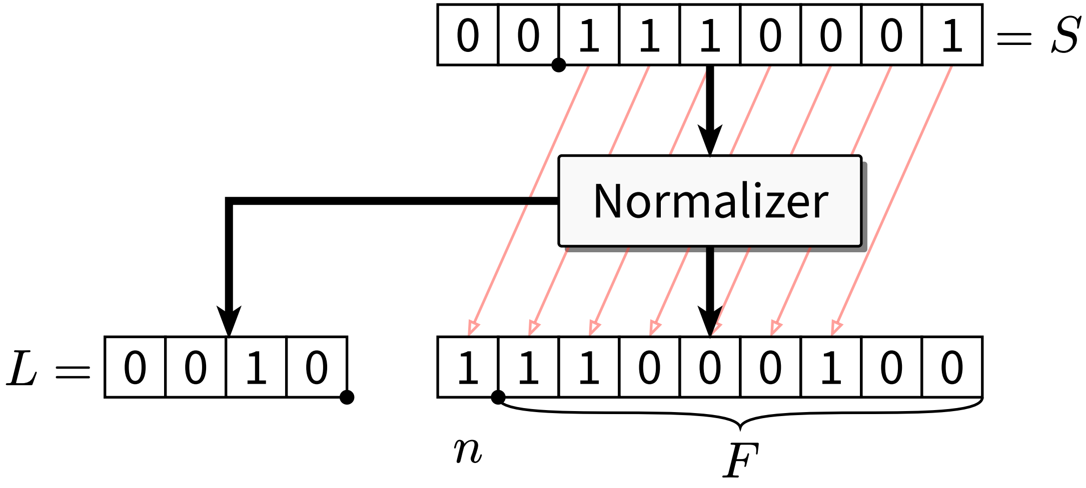
- L (exponent): leading zero count in the binary writing of the input
- $n.F$ (mantissa): shifted input where $n \in \{0,1\}$ and $F \in [0,1)$



Internal floating point format

The normalizer is a combined **leading zero counter** and **shifter**. Outputs:

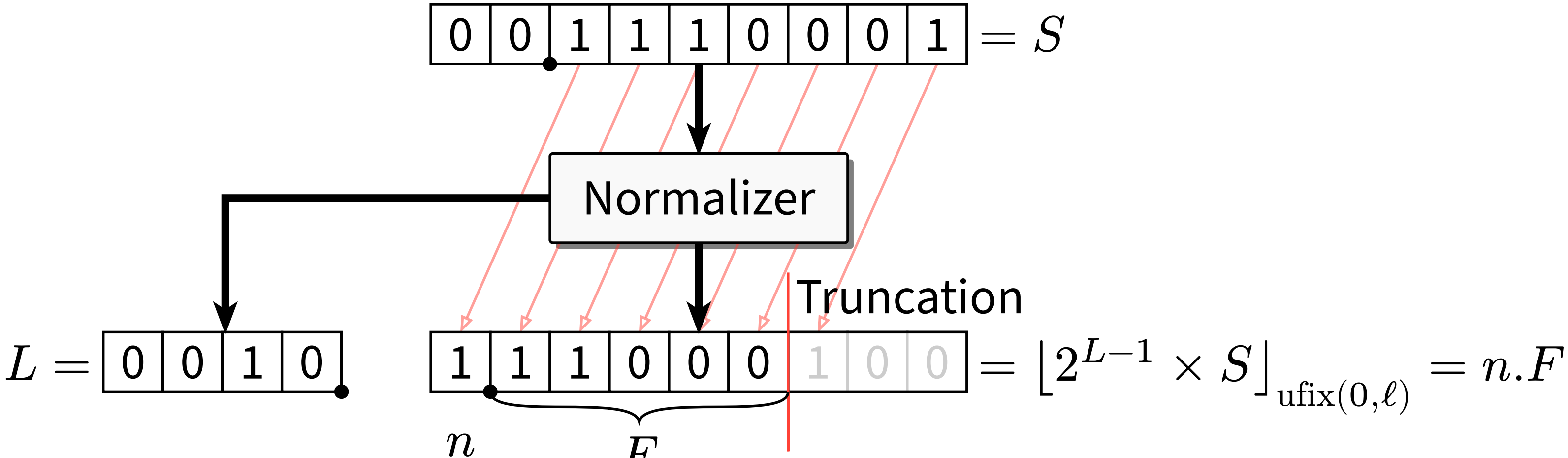
- L (exponent): leading zero count in the binary writing of the input
- $n.F$ (mantissa): shifted input where $n \in \{0, 1\}$ and $F \in [0, 1)$



Internal floating point format

The normalizer is a combined **leading zero counter** and **shifter**. Outputs:

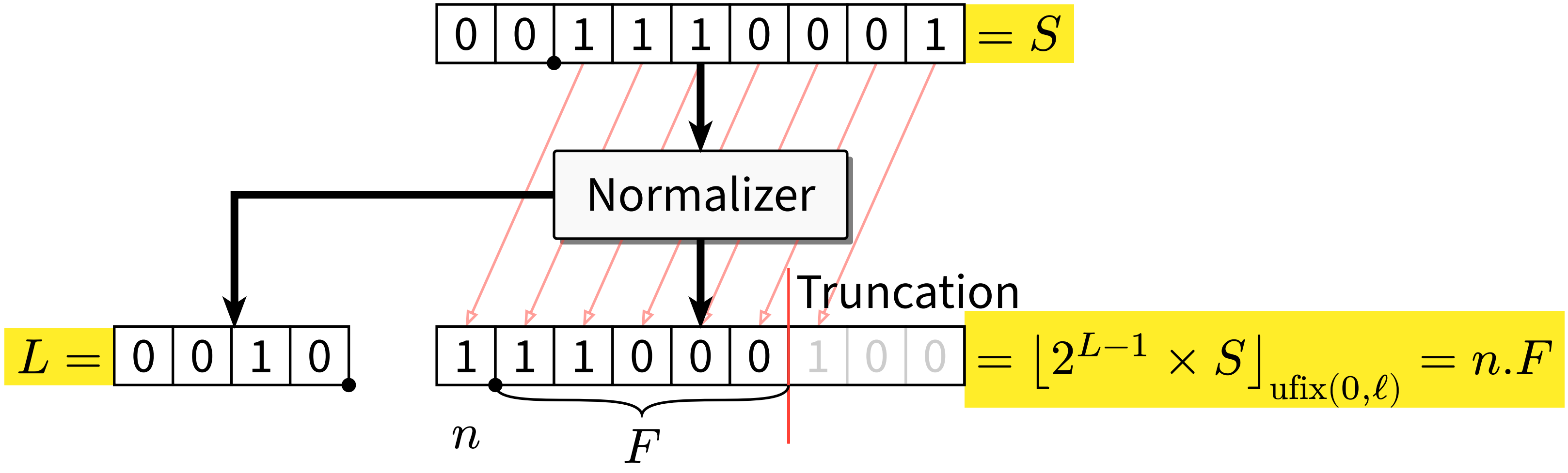
- L (exponent): leading zero count in the binary writing of the input
- $n.F$ (mantissa): shifted input where $n \in \{0, 1\}$ and $F \in [0, 1)$



Internal floating point format

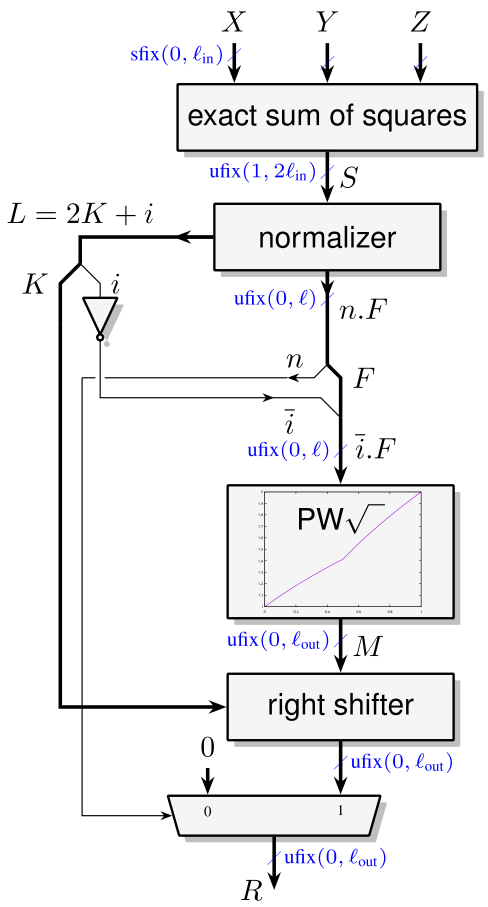
The normalizer is a combined **leading zero counter** and **shifter**. Outputs:

- L (exponent): leading zero count in the binary writing of the input
- $n.F$ (mantissa): shifted input where $n \in \{0, 1\}$ and $F \in [0, 1)$



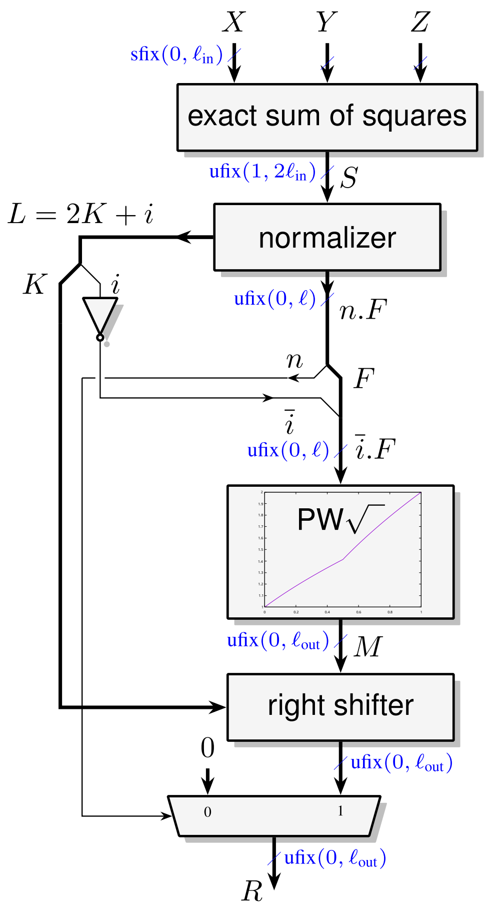
$$S = 2^{-L+1} \times n.F + \delta_{\text{truncation}}$$

A better naive solution



The normalizer outputs:

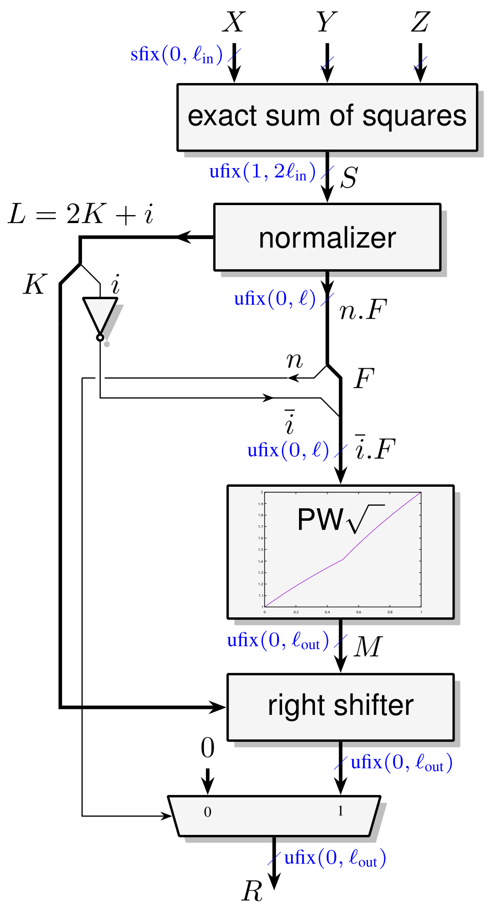
A better naive solution



The normalizer outputs:

- $L = 2K + i$ leading zero count of S

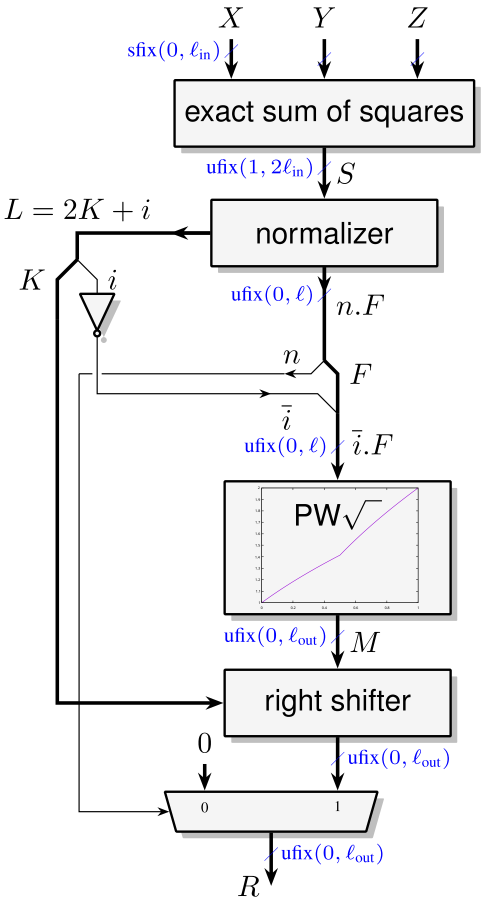
A better naive solution



The normalizer outputs:

- $L = 2K + i$ leading zero count of S
- $n.F = \lfloor 2^{L-1} S \rfloor_{\text{ufix}(0,\ell)}$ (truncation)

A better naive solution

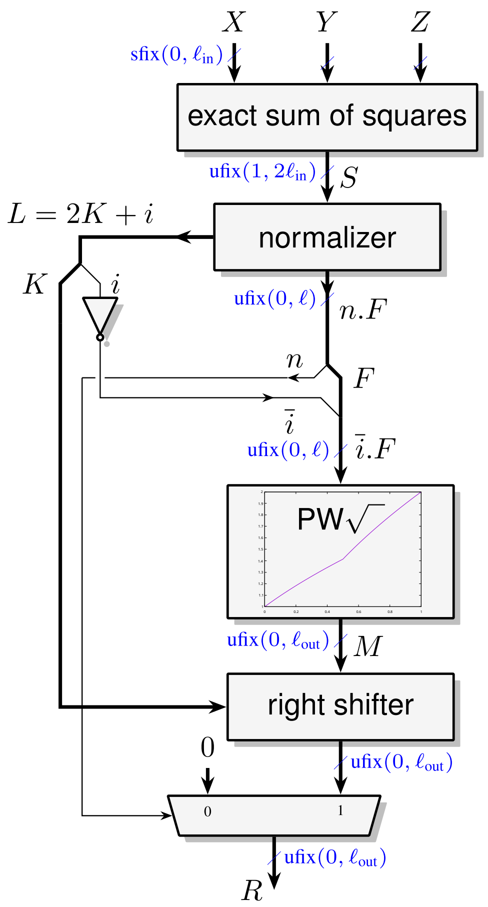


The normalizer outputs:

- $L = 2K + i$ leading zero count of S
- $n.F = \lfloor 2^{L-1} S \rfloor_{\mathrm{ufix}(0,\ell)}$ (truncation)

$$
\begin{aligned}
S &\approx 2^{-L+1} \times (1 + F) \\
&\approx 2^{-2K-i+1} \times (1 + F)
\end{aligned}
$$

A better naive solution



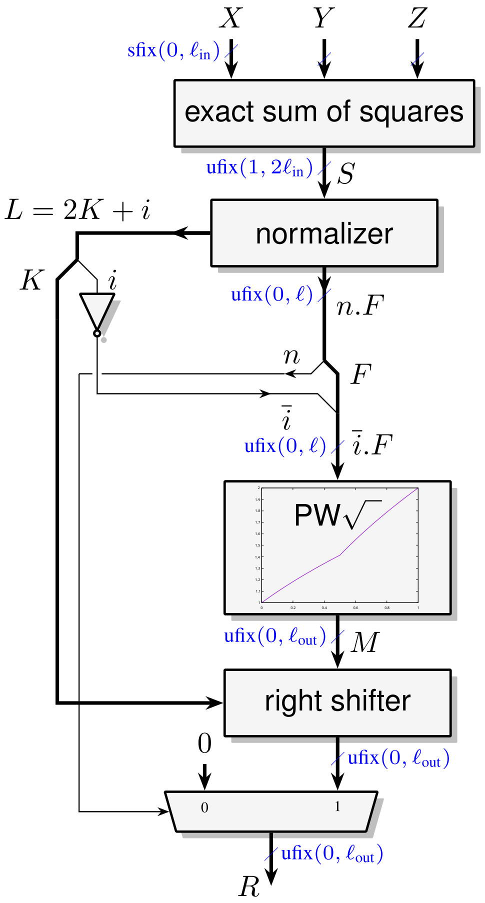
The normalizer outputs:

- $L = 2K + i$ leading zero count of S
- $n.F = \lfloor 2^{L-1} S \rfloor_{\mathrm{ufix}(0,\ell)}$ (truncation)

$$\begin{aligned} S &\approx 2^{-L+1} \times (1+F) \\ &\approx 2^{-2K-i+1} \times (1+F) \end{aligned}$$

Apply $\sqrt{\cdot}$ $\quad (n = 1)$:

A better naive solution



The normalizer outputs:

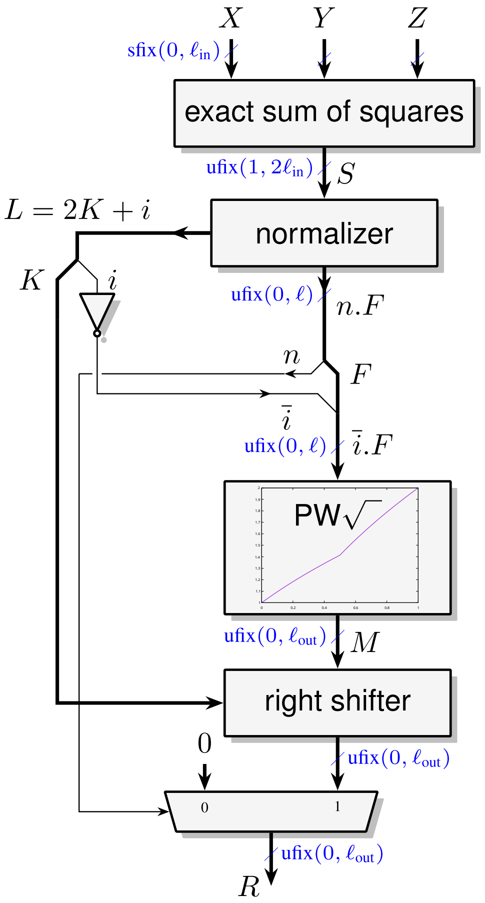
- $L = 2K + i$ leading zero count of S
- $n.F = \lfloor 2^{L-1} S \rfloor_{\mathrm{ufix}(0,\ell)}$ (truncation)

$$S \approx 2^{-L+1} \times (1 + F)$$
$$\approx 2^{-2K-i+1} \times (1 + F)$$

Apply $\sqrt{\cdot}$ $\quad (n = 1)$:

$$\sqrt{S} \approx \sqrt{2^{-2K-i+1}(1 + F)}$$

A better naive solution



The normalizer outputs:

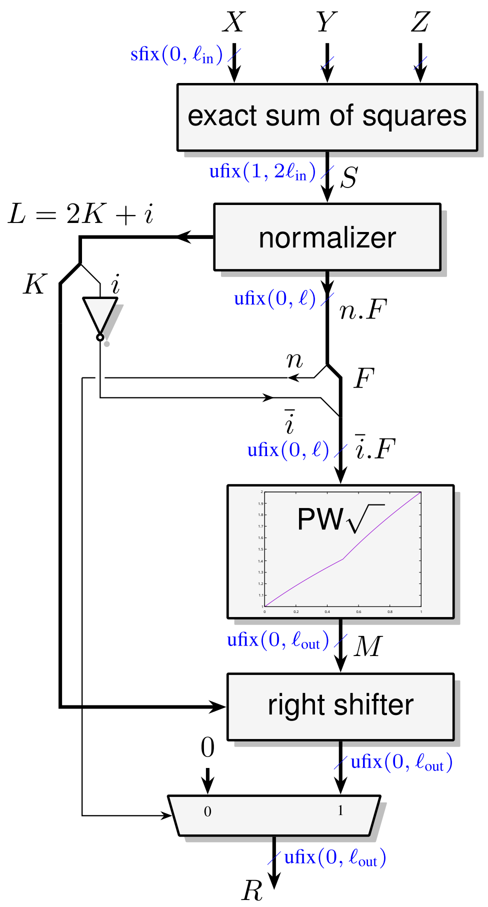
- $L = 2K + i$ leading zero count of S
- $n.F = \lfloor 2^{L-1} S \rfloor_{\text{ufix}(0,\ell)}$ (truncation)

$$S \approx 2^{-L+1} \times (1+F)$$
$$\approx 2^{-2K-i+1} \times (1+F)$$

Apply $\sqrt{\cdot}$ $\quad (n = 1)$:

$$\sqrt{S} \approx \sqrt{2^{-2K-i+1}(1+F)}$$
$$\approx 2^{-K}\sqrt{2^{1-i}(1+F)}$$

A better naive solution



The normalizer outputs:

- $L = 2K + i$ leading zero count of S
- $n.F = \lfloor 2^{L-1} S \rfloor_{\text{ufix}(0,\ell)}$ (truncation)

$$S \approx 2^{-L+1} \times (1 + F)$$
$$\approx 2^{-2K-i+1} \times (1 + F)$$

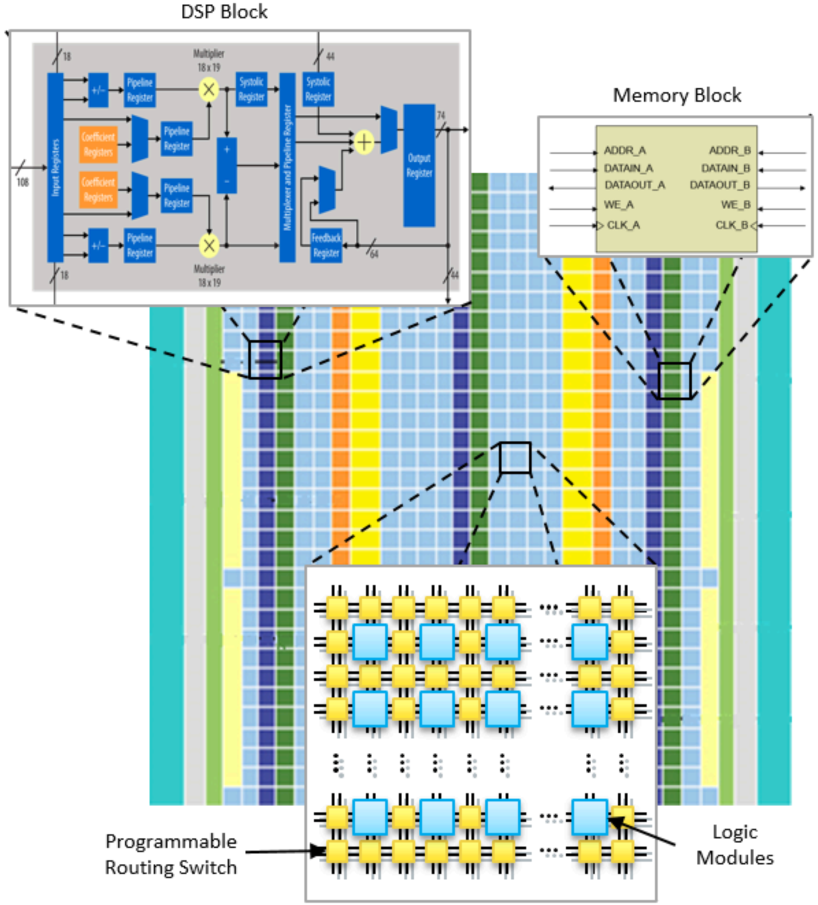
Apply $\sqrt{\cdot}$ $\quad (n = 1)$:

$$\sqrt{S} \approx \sqrt{2^{-2K-i+1}(1 + F)}$$
$$\approx 2^{-K}\sqrt{2^{1-i}(1 + F)}$$

Unify cases ($i \in \{0, 1\}$) to use approximation methods in FloPoCo:

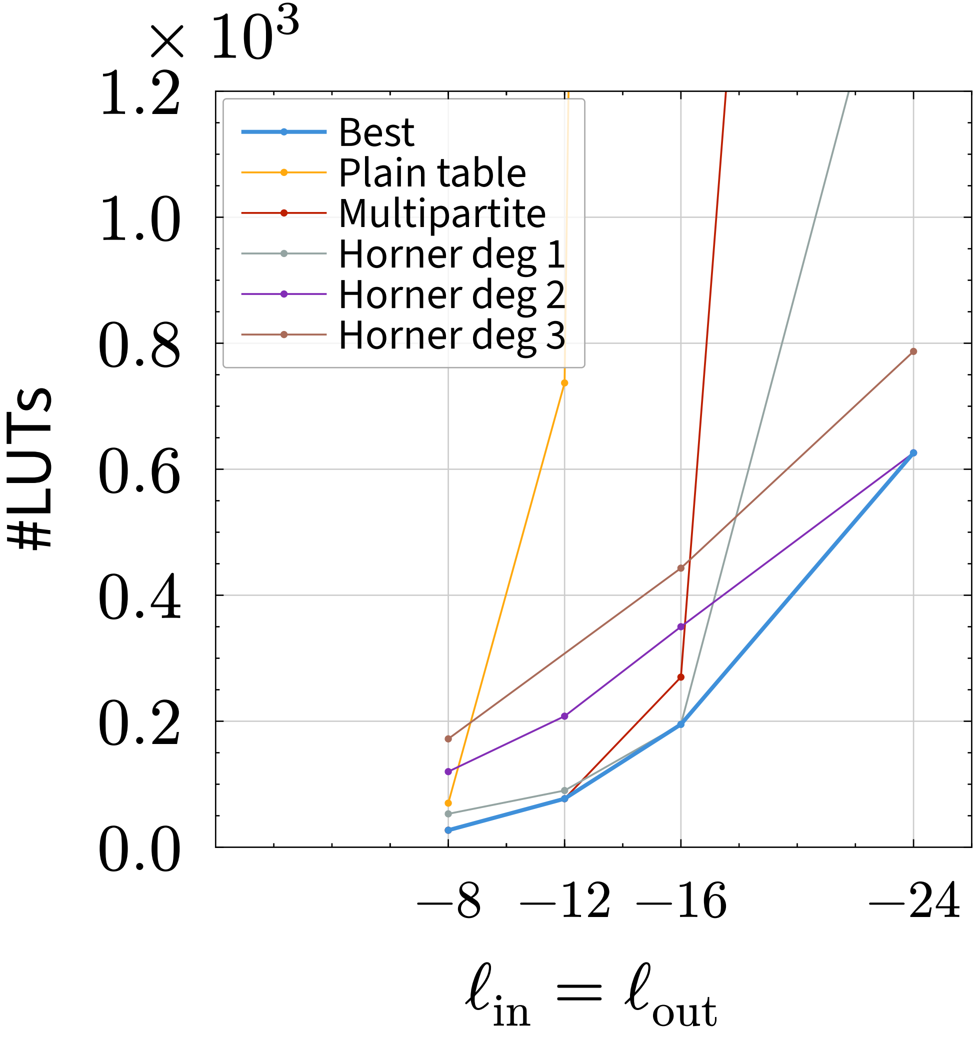
$$\sqrt{S} \approx 2^{-K} \times \mathrm{PW}\sqrt{\cdot}(\bar{i}.F)$$

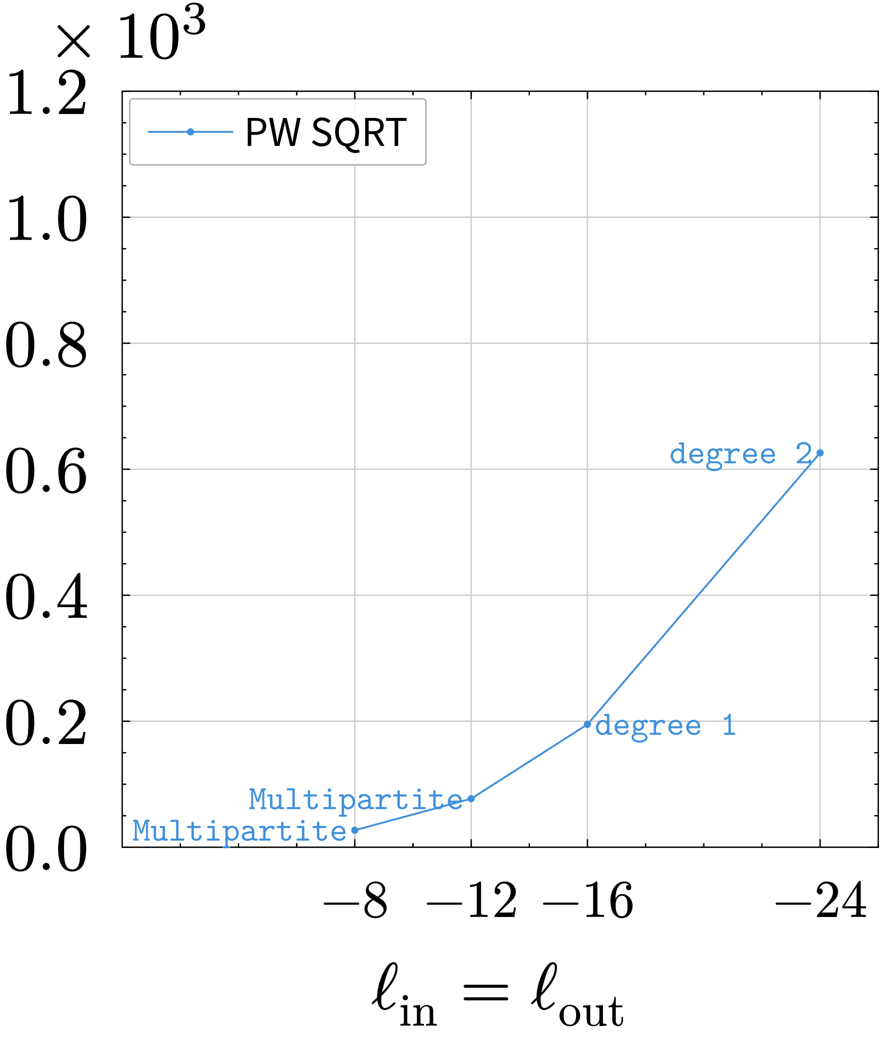
Field-programmable gate array (FPGA)



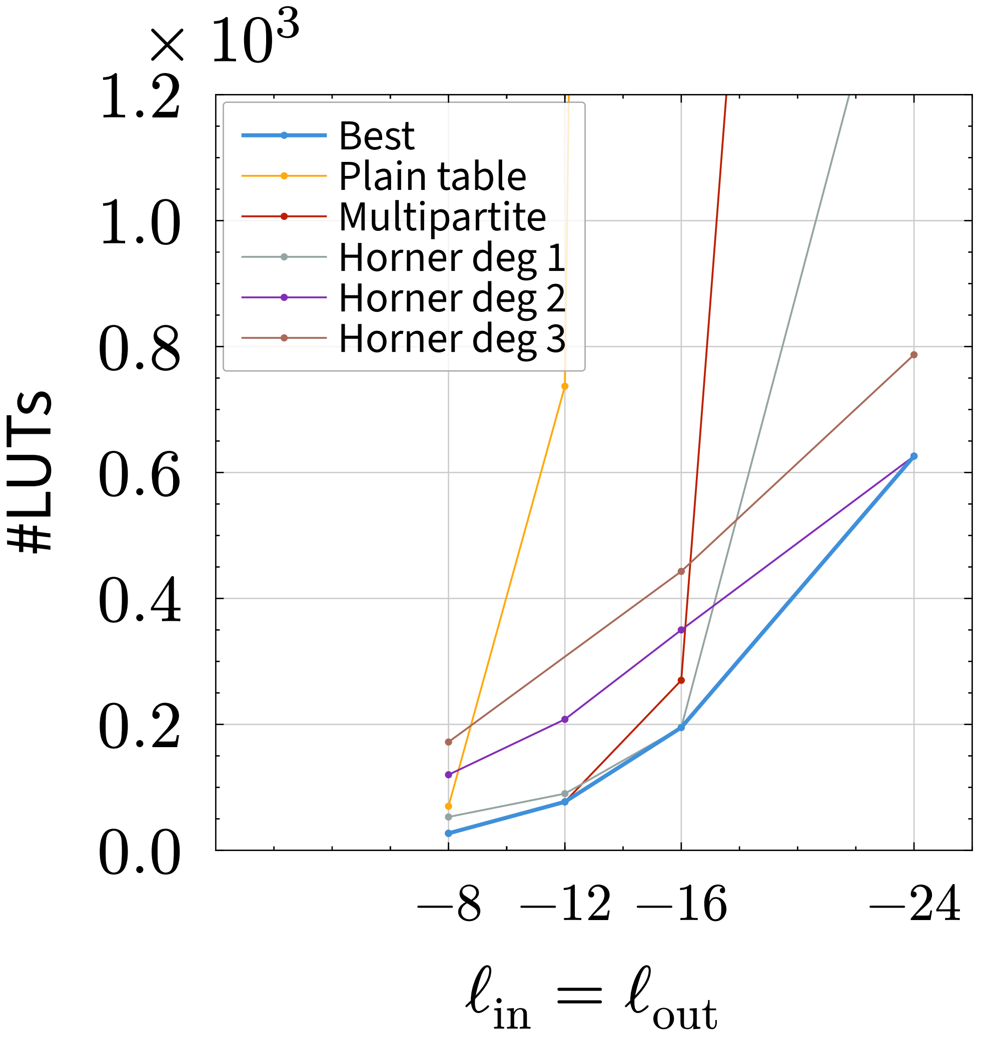
- FPGA is a type of integrated circuit that can be reprogrammed
- When running, the FPGA behaves like a digital circuit.
- Image from *Standard Edition Best Practices Guide* Intel manual

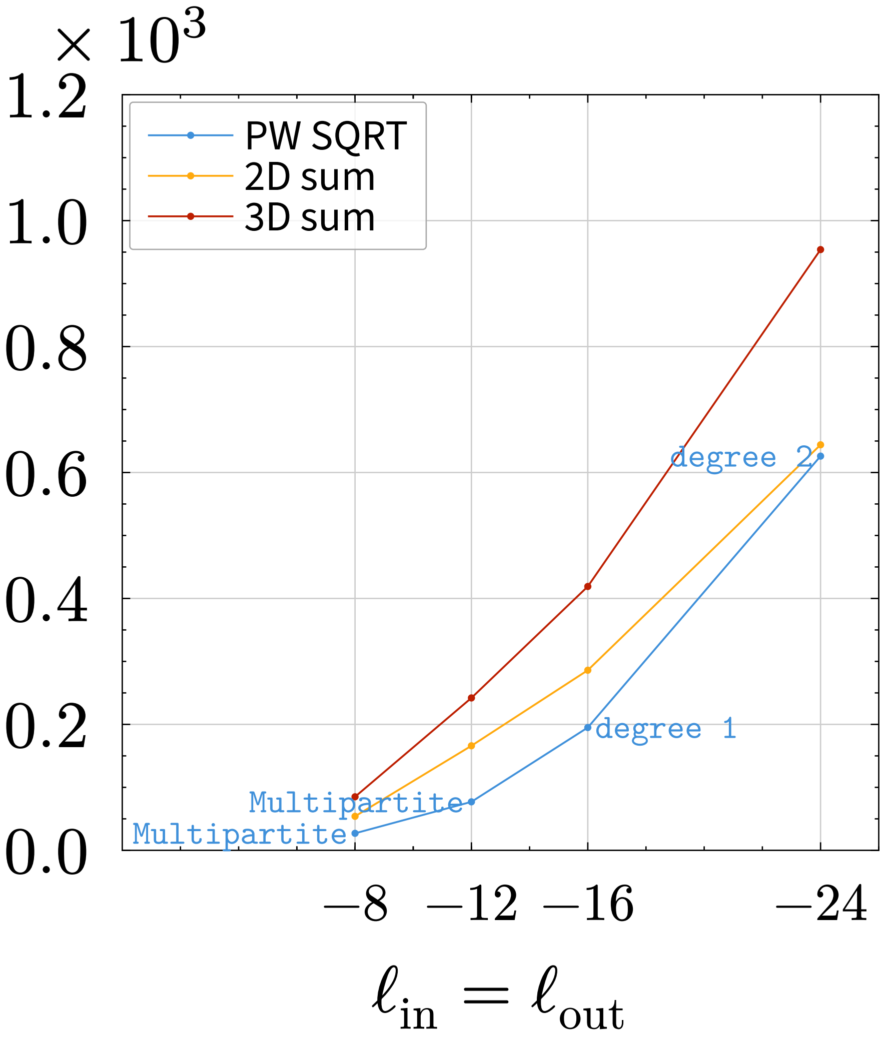
Area breakout of the Naive version



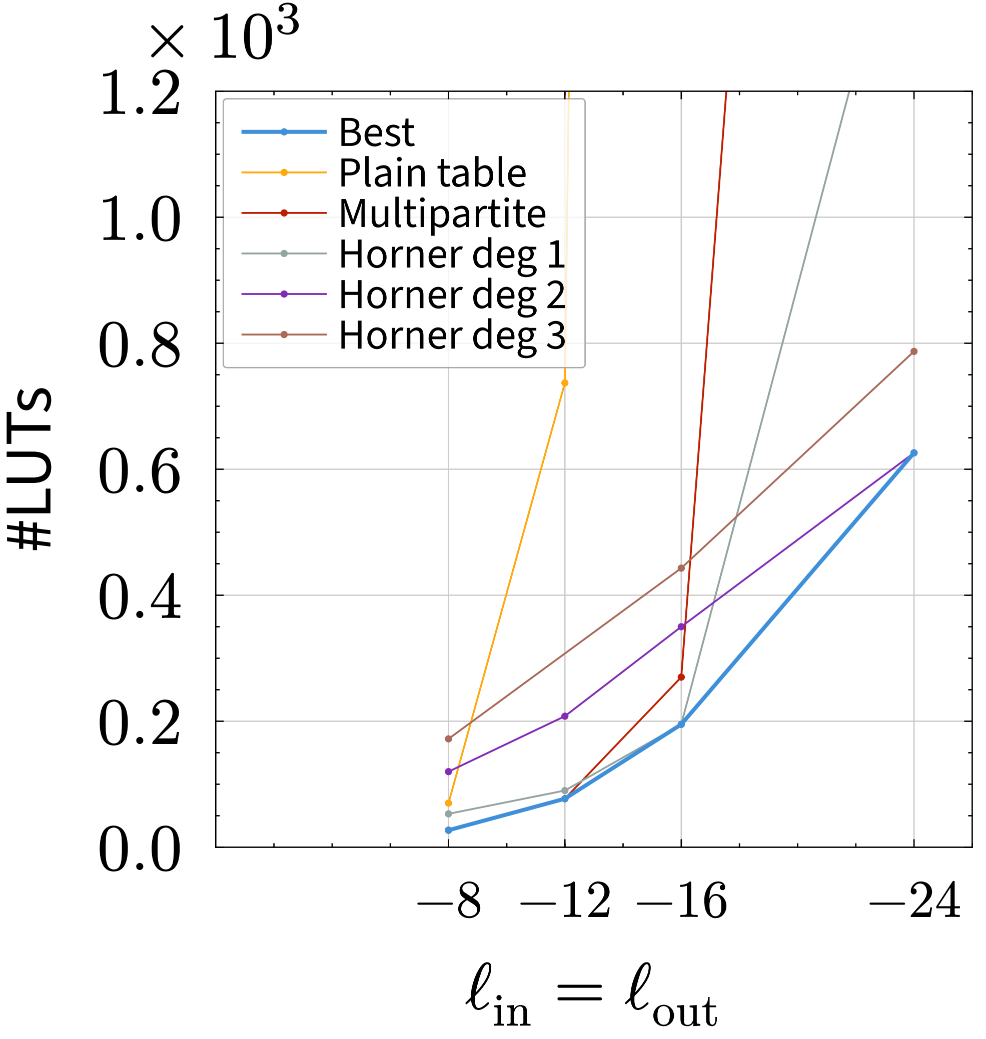


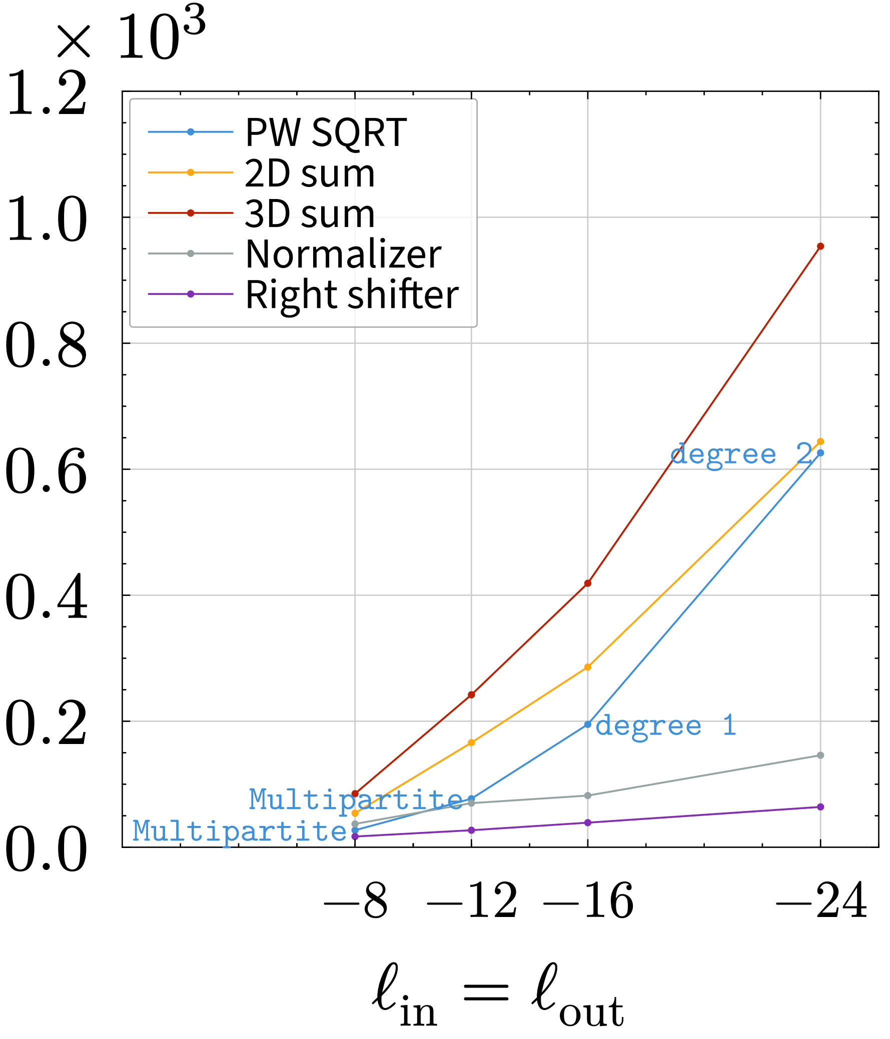
Area breakout of the Naive version





Area breakout of the Naive version





CORDIC

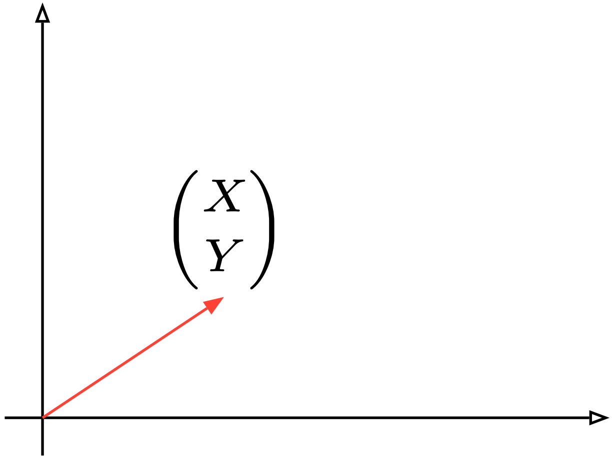
Definition

- CORDIC (for COordinate Rotation DIgital Computer) introduced by Volder in 1959. [1]
- Compute trigonometric functions and **norm** using “rotations”.
- Calculations are performed using only shift-and-add operations.

Definition

- CORDIC (for COordinate Rotation DIgital Computer) introduced by Volder in 1959. [1]
- Compute trigonometric functions and **norm** using “rotations”.
- Calculations are performed using only shift-and-add operations.

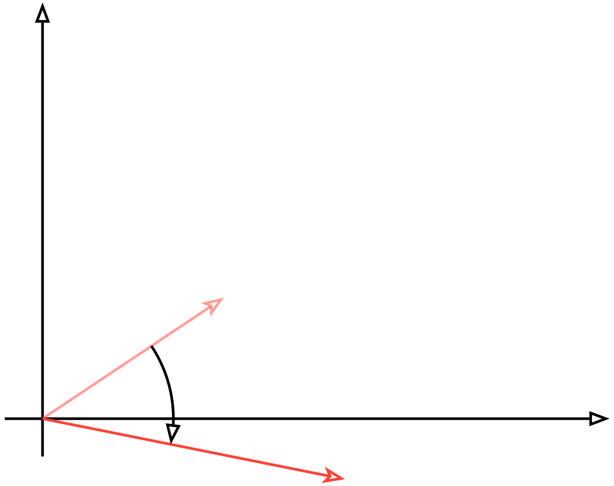
CORDIC iterations.



Definition

- CORDIC (for COordinate Rotation DIgital Computer) introduced by Volder in 1959. [1]
- Compute trigonometric functions and **norm** using “rotations”.
- Calculations are performed using only shift-and-add operations.

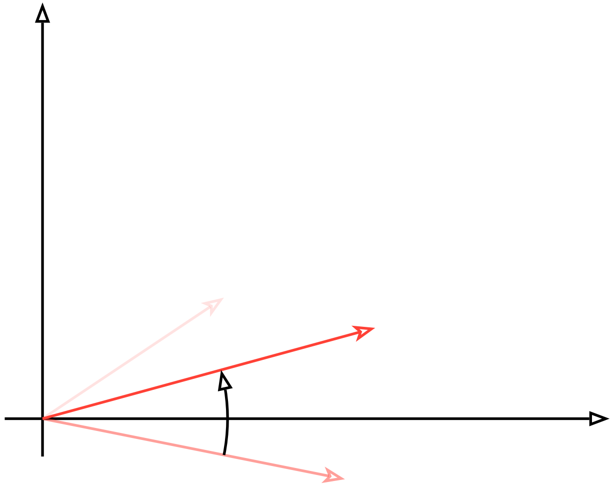
CORDIC iterations.



Definition

- CORDIC (for COordinate Rotation DIgital Computer) introduced by Volder in 1959. [1]
- Compute trigonometric functions and **norm** using “rotations”.
- Calculations are performed using only shift-and-add operations.

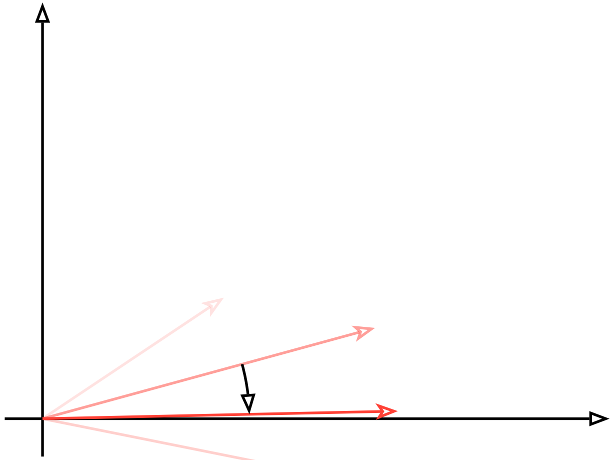
CORDIC iterations.



Definition

- CORDIC (for COordinate Rotation DIgital Computer) introduced by Volder in 1959. [1]
- Compute trigonometric functions and **norm** using “rotations”.
- Calculations are performed using only shift-and-add operations.

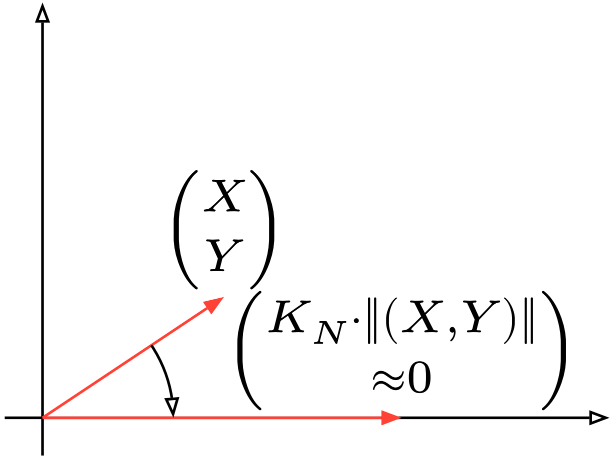
CORDIC iterations.



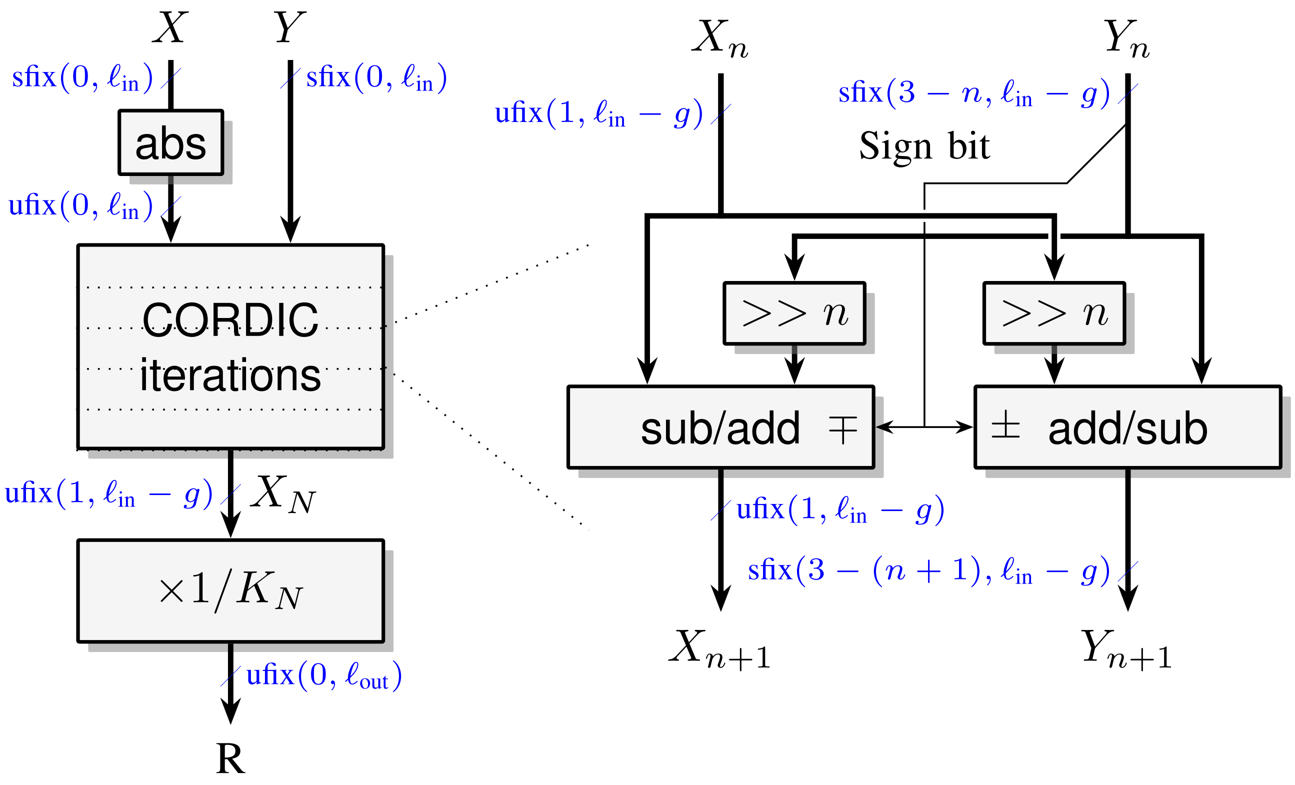
Definition

- CORDIC (for COordinate Rotation DIgital Computer) introduced by Volder in 1959. [1]
- Compute trigonometric functions and **norm** using “rotations”.
- Calculations are performed using only shift-and-add operations.

CORDIC iterations.



CORDIC 2D abstract architecture



Contributions: Convergence analysis with truncation and rounding errors $\Rightarrow$ smallest $N = \left\lceil -\frac{\ell_{out}}{2} + \frac{9}{4} \right\rceil$

CORDIC 3D

Householder CORDIC from S.-F. Hsiao and J.-M. Delosme in 3D:

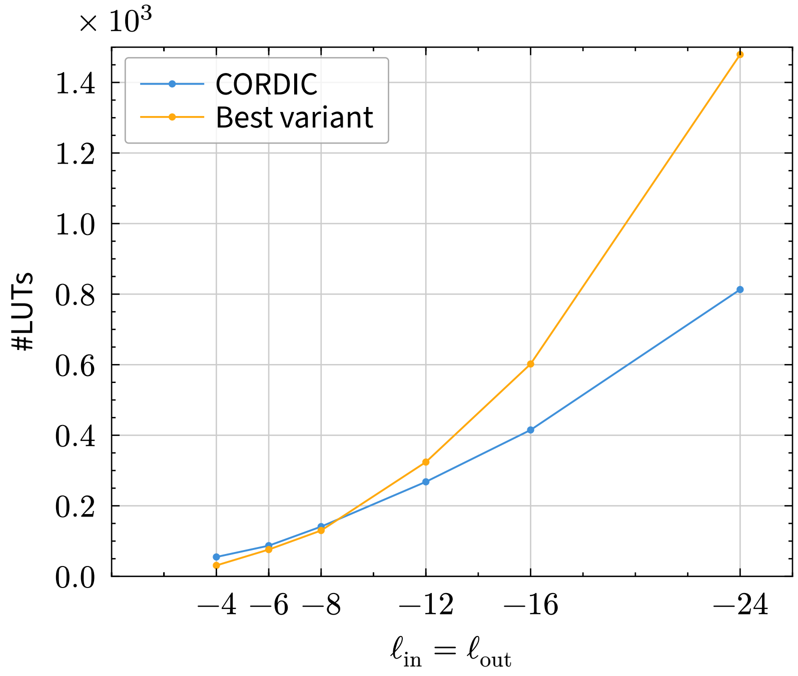
$$\begin{cases} \delta_n & = +1 \quad \text{if } x_n y_n \leq 0 \text{ else } -1 \\ \lambda_n & = +1 \quad \text{if } x_n z_n \leq 0 \text{ else } -1 \\ x_{n+1} & = x_n - 2^{-2n+1} x_n + \delta_n 2^{-n+1} y_n + \lambda_n 2^{-n+1} z_n \\ y_{n+1} & = y_n - \delta_n 2^{-n+1} x_n - \delta_n \lambda_n 2^{-2n+1} z_n \\ z_{n+1} & = z_n - \lambda_n 2^{-n+1} x_n - \delta_n \lambda_n 2^{-2n+1} y_n \end{cases}$$

We observed that $N = O(|\ell_{\text{out}}|)$

Comparison

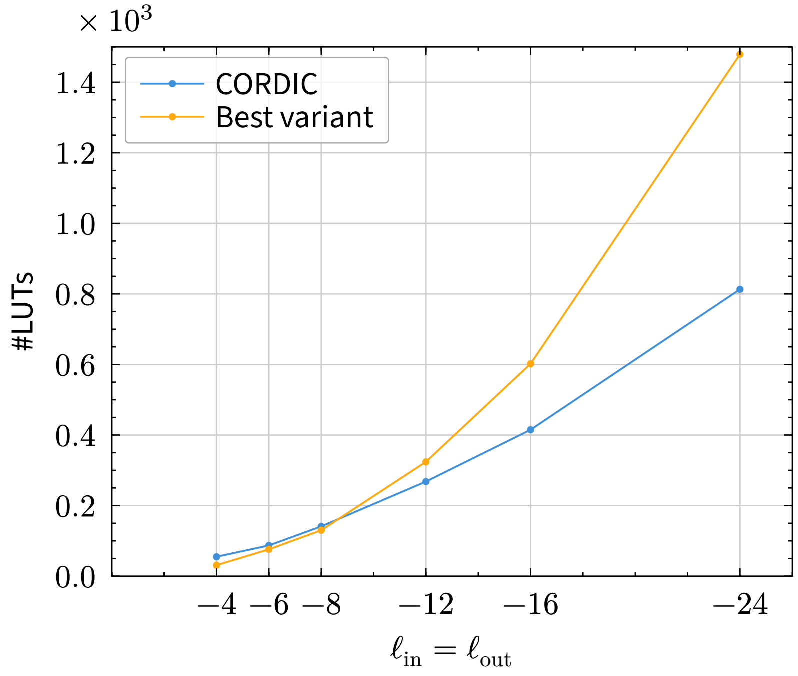
2D Results

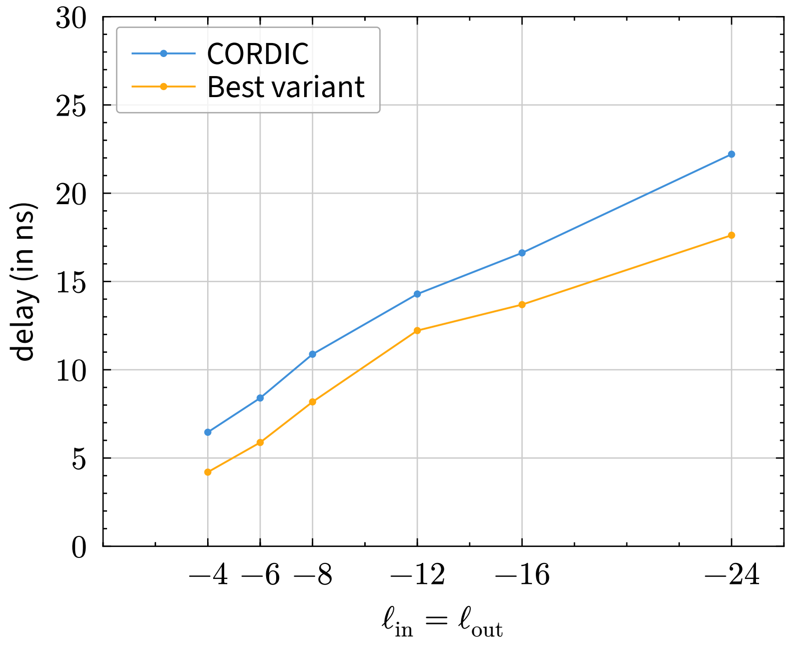
2D	4 bits	6 bits	8 bits	12 bits	16 bits	24 bits
CORDIC	55 L / 6.46 ns	87 L / 8.4 ns	141 L / 10.88 ns	**268** L / 14.29 ns	**415** L / 16.62 ns	**813** L / 22.21 ns
NaivePlainTable	34 L / **4.2** ns	**76** L / **5.88** ns	178 L / **8.18** ns	1000 L / **12.22** ns	14898 L / **13.69** ns	N/A
NaiveMultiPartite	**31** L / 4.39 ns	93 L / 6.66 ns	**130** L / 8.74 ns	324 L / 12.88 ns	677 L / 14.33 ns	5926 L / **17.62** ns
NaivePiecewiseHorner1	48 L / 5.8 ns	90 L / 8.27 ns	161 L / 9.72 ns	353 L / 13.55 ns	602 L / 14.58 ns	2441 L / 18.83 ns
NaivePiecewiseHorner2	51 L / 6.12 ns	137 L / 10.27 ns	228 L / 11.55 ns	471 L / 16.32 ns	757 L / 17.77 ns	1479 L / 20.82 ns
NaivePiecewiseHorner3	N/A	151 L / 9.94 ns	280 L / 13.26 ns	N/A	850 L / 20.76 ns	1641 L / 23.56 ns



2D Results

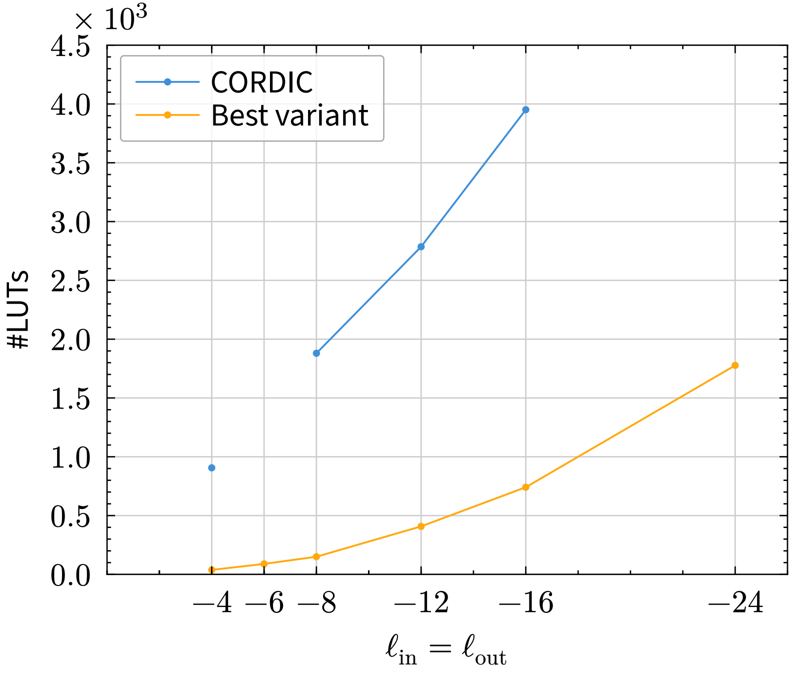
2D	4 bits	6 bits	8 bits	12 bits	16 bits	24 bits
CORDIC	55 L / 6.46 ns	87 L / 8.4 ns	141 L / 10.88 ns	**268** L / 14.29 ns	**415** L / 16.62 ns	**813** L / 22.21 ns
NaivePlainTable	34 L / **4.2** ns	**76** L / **5.88** ns	178 L / **8.18** ns	1000 L / **12.22** ns	14898 L / **13.69** ns	N/A
NaiveMultiPartite	**31** L / 4.39 ns	93 L / 6.66 ns	**130** L / 8.74 ns	324 L / 12.88 ns	677 L / 14.33 ns	5926 L / **17.62** ns
NaivePiecewiseHorner1	48 L / 5.8 ns	90 L / 8.27 ns	161 L / 9.72 ns	353 L / 13.55 ns	602 L / 14.58 ns	2441 L / 18.83 ns
NaivePiecewiseHorner2	51 L / 6.12 ns	137 L / 10.27 ns	228 L / 11.55 ns	471 L / 16.32 ns	757 L / 17.77 ns	1479 L / 20.82 ns
NaivePiecewiseHorner3	N/A	151 L / 9.94 ns	280 L / 13.26 ns	N/A	850 L / 20.76 ns	1641 L / 23.56 ns





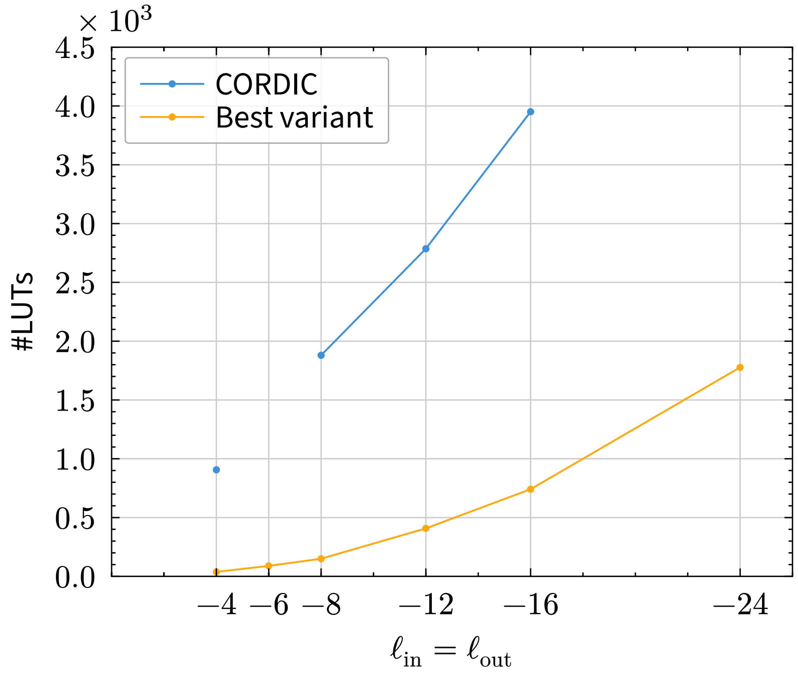
3D Results

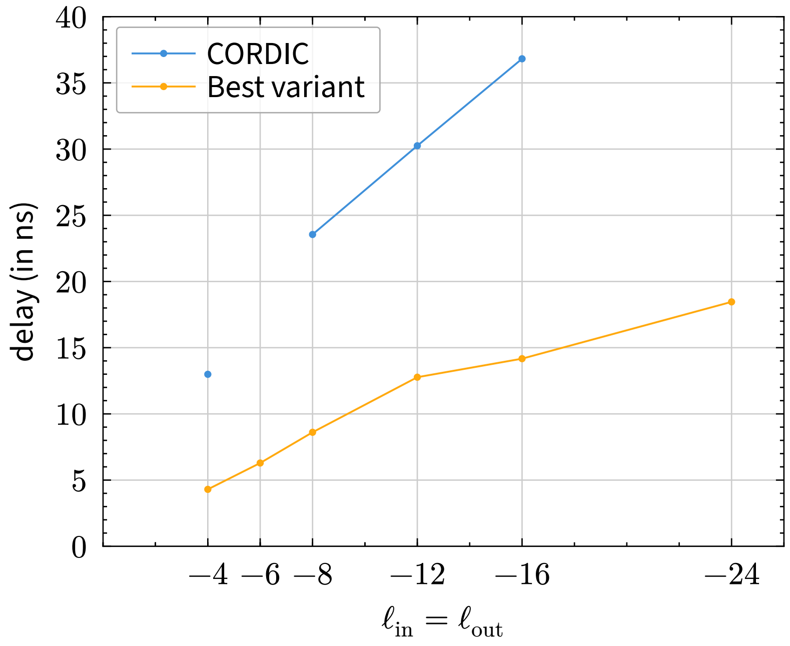
3D	4 bits	6 bits	8 bits	12 bits	16 bits	24 bits
CORDIC	906 L / 12.99 ns	N/A	1880 L / 23.55 ns	2786 L / 30.25 ns	3951 L / 36.82 ns	N/A
NaivePlainTable	**38** L / **4.3** ns	103 L / **6.29** ns	200 L / **8.61** ns	1066 L / **12.77** ns	15038 L / **14.17** ns	N/A
NaiveMultiPartite	**38** L / **4.3** ns	**89** L / 8.03 ns	**150** L / 9.09 ns	**408** L / 13.3 ns	817 L / 14.81 ns	6236 L / **18.46** ns
NaivePiecewiseHorner1	55 L / 5.85 ns	102 L / 8.58 ns	185 L / 10.02 ns	433 L / 14.08 ns	**741** L / 15.1 ns	2749 L / 19.68 ns
NaivePiecewiseHorner2	60 L / 6.33 ns	142 L / 10.49 ns	250 L / 11.98 ns	550 L / 16.69 ns	896 L / 18.19 ns	**1777** L / 21.54 ns
NaivePiecewiseHorner3	N/A	N/A	300 L / 13.69 ns	N/A	991 L / 21.16 ns	1915 L / 24.52 ns



3D Results

3D	4 bits	6 bits	8 bits	12 bits	16 bits	24 bits
CORDIC	906 L / 12.99 ns	N/A	1880 L / 23.55 ns	2786 L / 30.25 ns	3951 L / 36.82 ns	N/A
NaivePlainTable	**38** L / **4.3** ns	103 L / **6.29** ns	200 L / **8.61** ns	1066 L / **12.77** ns	15038 L / **14.17** ns	N/A
NaiveMultiPartite	**38** L / **4.3** ns	**89** L / 8.03 ns	**150** L / 9.09 ns	**408** L / 13.3 ns	817 L / 14.81 ns	6236 L / **18.46** ns
NaivePiecewiseHorner1	55 L / 5.85 ns	102 L / 8.58 ns	185 L / 10.02 ns	433 L / 14.08 ns	**741** L / 15.1 ns	2749 L / 19.68 ns
NaivePiecewiseHorner2	60 L / 6.33 ns	142 L / 10.49 ns	250 L / 11.98 ns	550 L / 16.69 ns	896 L / 18.19 ns	**1777** L / 21.54 ns
NaivePiecewiseHorner3	N/A	N/A	300 L / 13.69 ns	N/A	991 L / 21.16 ns	1915 L / 24.52 ns





Conclusion

Contributions

- A comparison in time and hardware cost between the CORDIC algorithm and the naive tabulation method for computing norms
- Two verified norm operators in the open-source project FloPoCo

Contributions

- A comparison in time and hardware cost between the CORDIC algorithm and the naive tabulation method for computing norms
- Two verified norm operators in the open-source project FloPoCo

Executive summary:

- 2D:
  - $-8 \leq (\ell_{\text{in}} = \ell_{\text{out}})$ should use **NAIVE**
  - $-8 > (\ell_{\text{in}} = \ell_{\text{out}})$ should use **CORDIC**
- 3D:
  - Should use **NAIVE**

Future work

- Fix truncated FloPoCo squarers and merged arithmetic ($X^2 + Y^2 + Z^2$).

Future work

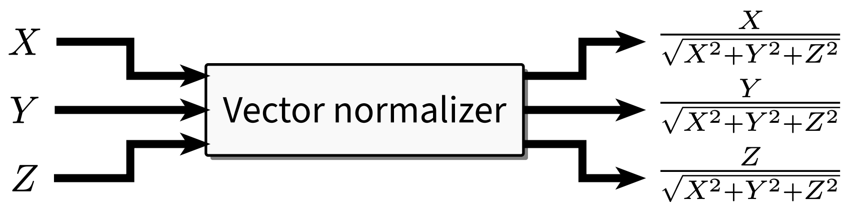
- Fix truncated FloPoCo squarers and merged arithmetic ($X^2 + Y^2 + Z^2$).
- Extend to floating-point norm computation

Future work

- Fix truncated FloPoCo squarers and merged arithmetic ($X^2 + Y^2 + Z^2$).
- Extend to floating-point norm computation
- Explore redundant encodings from the literature [3], [4]:
  Can they let CORDIC match the naive method in delay? At what hardware cost?

Future work

- Fix truncated FloPoCo squarers and merged arithmetic ($X^2 + Y^2 + Z^2$).
- Extend to floating-point norm computation
- Explore redundant encodings from the literature [3], [4]:
  Can they let CORDIC match the naive method in delay? At what hardware cost?
- Generator of circuits that directly normalize vectors:



Bibliography

[1] J. E. Volder, “The CORDIC Trigonometric Computing Technique,” *IRE Transactions on Electronic Computers*, no. 3, pp. 330–334, 1959, doi: 10.1109/TEC.1959.5222693.

[2] S.-F. Hsiao and J.-M. Delosme, “Householder CORDIC Algorithms,” *IEEE Transactions on Computers*, vol. 44, no. 8, pp. 990–1001, Aug. 1995, doi: 10.1109/12.403715.

[3] J. Villalba, J. Arrabal, E. Zapata, E. Antelo, and J. Bruguera, “Radix-4 Vectoring CORDIC Algorithm and Architectures,” in *Proceedings of International Conference on Application Specific Systems, Architectures and Processors: ASAP*, Chicago, IL, USA: IEEE Computer Soc. Press, 1996, pp. 55–64. doi: 10.1109/ASAP.1996.542801.

[4] T. Lang and P. Montuschi, “Very High Radix Square Root with Prescaling and Rounding and a Combined Division/Square Root Unit,” *IEEE Transactions on Computers*, vol. 48, no. 8, pp. 827–841, Aug. 1999, doi: 10.1109/12.795124.

[5] F. de Dinechin and B. Pasca, “Designing Custom Arithmetic Data Paths with FloPoCo,” *IEEE Design & Test of Computers*, vol. 28, no. 4, pp. 18–27, Jul. 2011.

[6] J.-M. Muller, *Elementary Functions*, 3rd ed. Birkhäuser Boston, 2016.

Bibliography

[7] P. K. Meher, J. Valls, T.-B. Juang, K. Sridharan, and K. Maharatna, “50 Years of CORDIC: Algorithms, Architectures, and Applications,” *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 56, no. 9, pp. 1893–1907, 2009, doi: 10.1109/TCSI.2009.2025803.

[8] F. de Dinechin and M. Kumm, *Application-Specific Arithmetic*, 1st ed. Springer Cham, 2024.

[9] D. Das Sarma and D. Matula, “Faithful Bipartite ROM Reciprocal Tables,” in *Proceedings of the 12th Symposium on Computer Arithmetic*, Jul. 1995, pp. 17–28. doi: 10.1109/ARITH.1995.465381.

Appendix

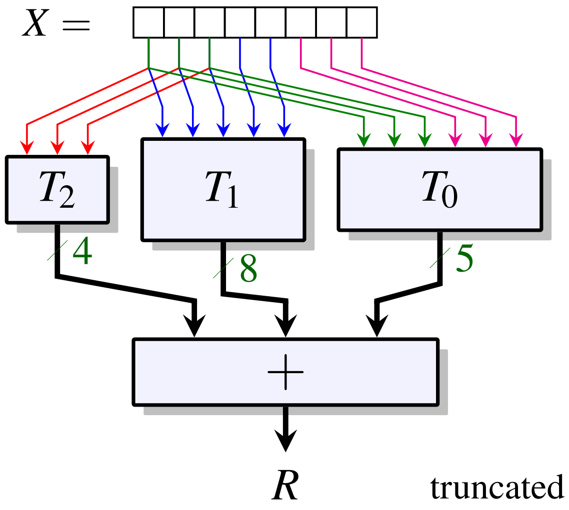
CORDIC 2D:

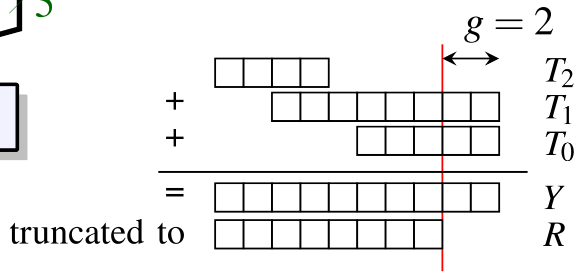
$$\begin{cases} d_n & = +1 \quad \text{if } y_n < 0 \text{ else} -1 \\ x_{n+1} & = x_n - 2^{-n} d_n y_n \\ y_{n+1} & = y_n + 2^{-n} d_n x_n \\ \theta_{n+1} & = x_n - d_n \arctan(2^{-n}) \end{cases}$$

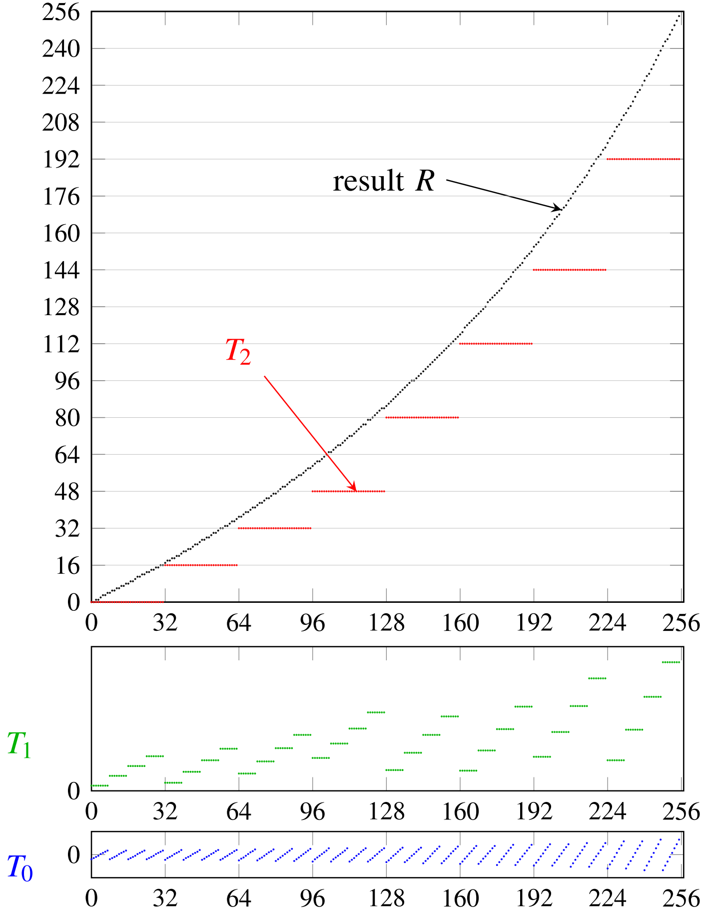
Householder CORDIC in 3D:

$$\begin{cases} \delta_n & = +1 \quad \text{if } x_n y_n \leq 0 \text{ else} -1 \\ \lambda_n & = +1 \quad \text{if } x_n z_n \leq 0 \text{ else} -1 \\ x_{n+1} & = x_n - 2^{-2n+1} x_n + \delta_n 2^{-n+1} y_n + \lambda_n 2^{-n+1} z_n \\ y_{n+1} & = y_n - \delta_n 2^{-n+1} x_n - \delta_n \lambda_n 2^{-2n+1} z_n \\ z_{n+1} & = z_n - \lambda_n 2^{-n+1} x_n - \delta_n \lambda_n 2^{-2n+1} y_n \end{cases}$$

Approximation of $\frac{2}{2-x} - 1$:

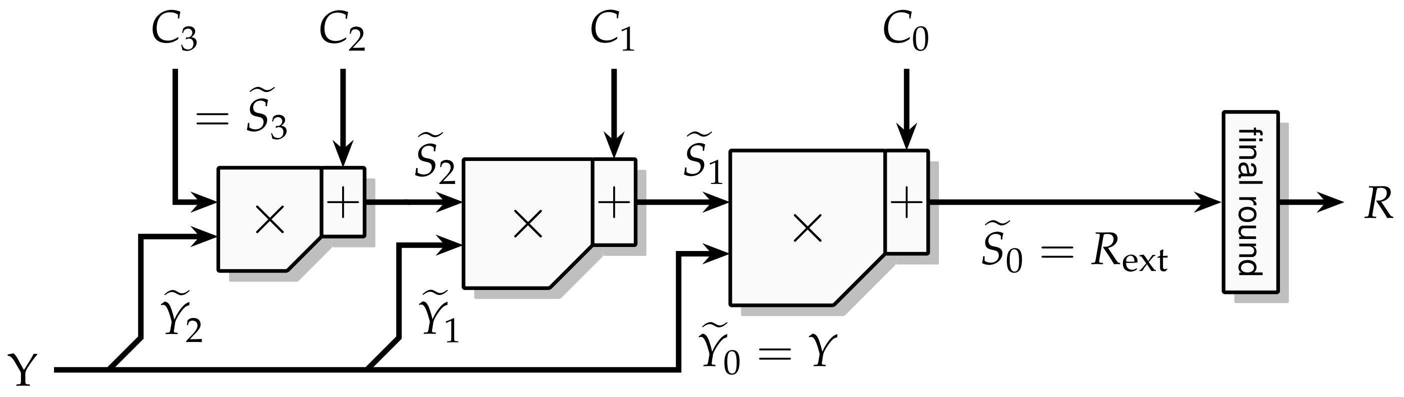






C_i determined with Taylor, Remez minimax, Sollya's fpminimax

$$p(Y) = C_0 + Y \times (C_1 + Y \times (C_2 + Y \times C_3))$$



- Faithful: $|R - \mathrm{norm}| < 2^{\ell_{\mathrm{out}}}$
- Correct: $|R - \mathrm{norm}| \leq 2^{\ell_{\mathrm{out}}-1}$

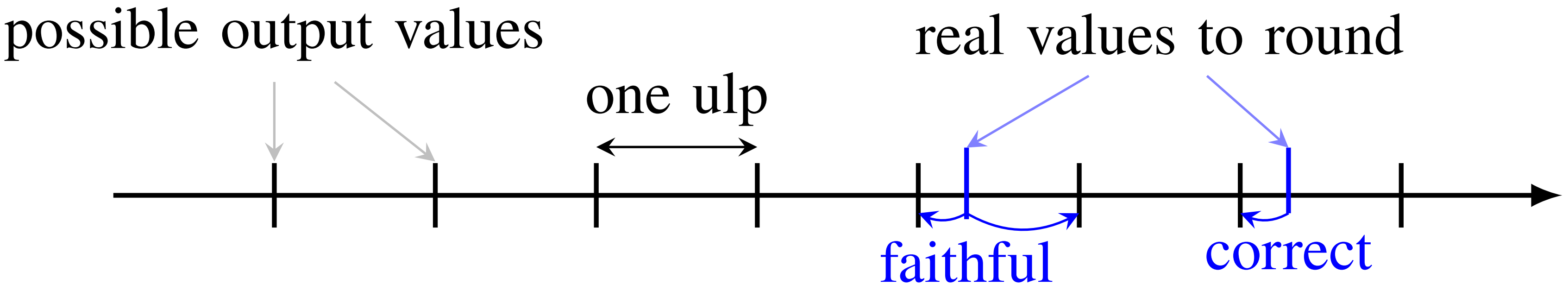
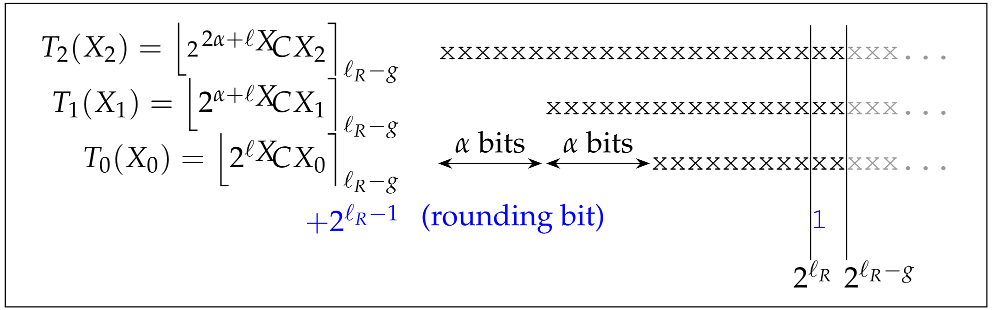


Fig. 4: Faithful versus correct rounding

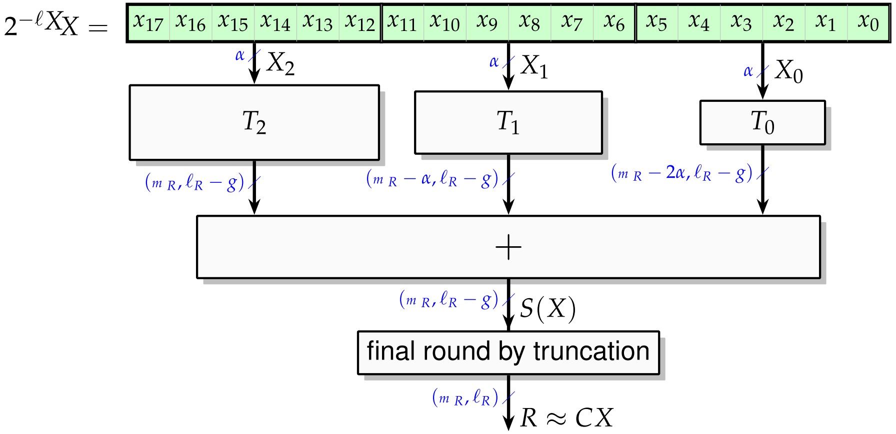
Why inputs in $[-1, 1)$ $\mathrm{sfix}(0, \ell_{\mathrm{in}})$ and not in $[0, 1)$ in $\mathrm{ufix}(0, \ell_{\mathrm{in}})$?

- Useful for most users
- User is not an expert

Implemented with a Ken Chapman's multiplier



(a) *Aligment of the partial products*



(b) *Architecture*

Fig. 12.15 Fix by Real KCM. The 18-bit input X is split in $D = 3$ chunks of $\alpha = 6$ bits.

TABLE IV: Norms with 8-bit inputs and wider output

output width	8 bits	12 bits	16 bits
2D CORDIC	141 L / 10.88 ns	216 L / 13.28 ns	311 L / 15.65 ns
2D Naive	130 L / 8.74 ns	188 L / 9.54 ns	363 L / 9.94 ns
3D Naive	150 L / 9.09 ns	211 L / 9.98 ns	385 L / 10.38 ns