Accuracy-preserving Layer Normalization Approximations

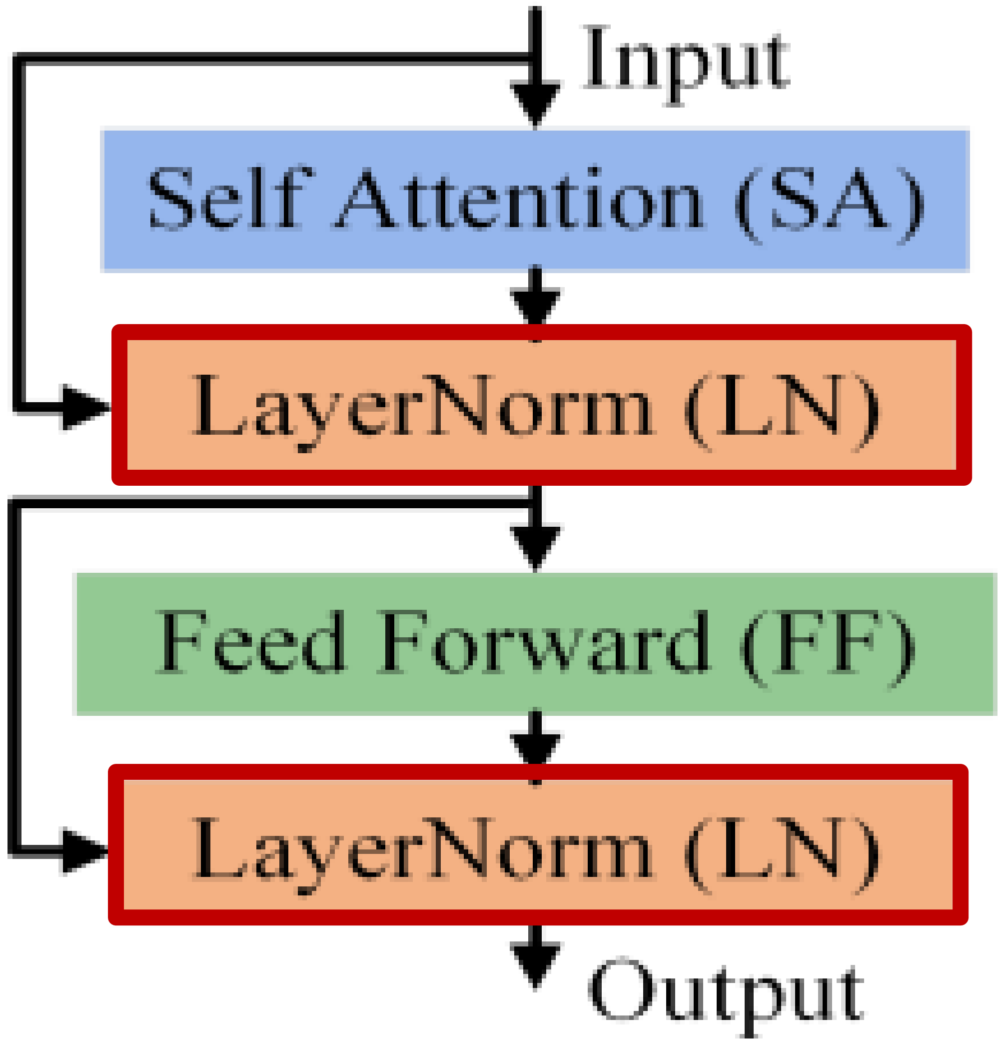
2025.7.01

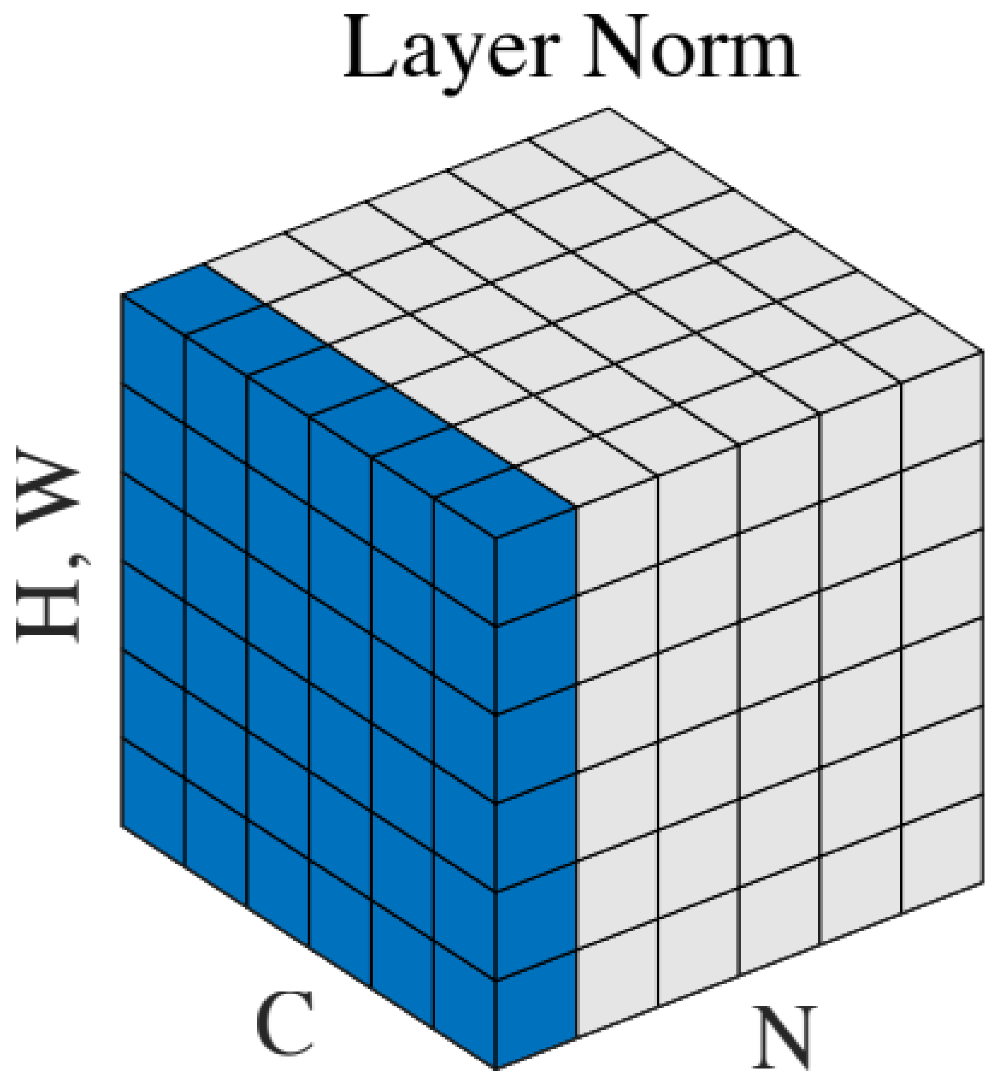
Sangyun Kim
Eunju Kim

CONTENTS

What is Layer Normalization?

LN location in Transformer



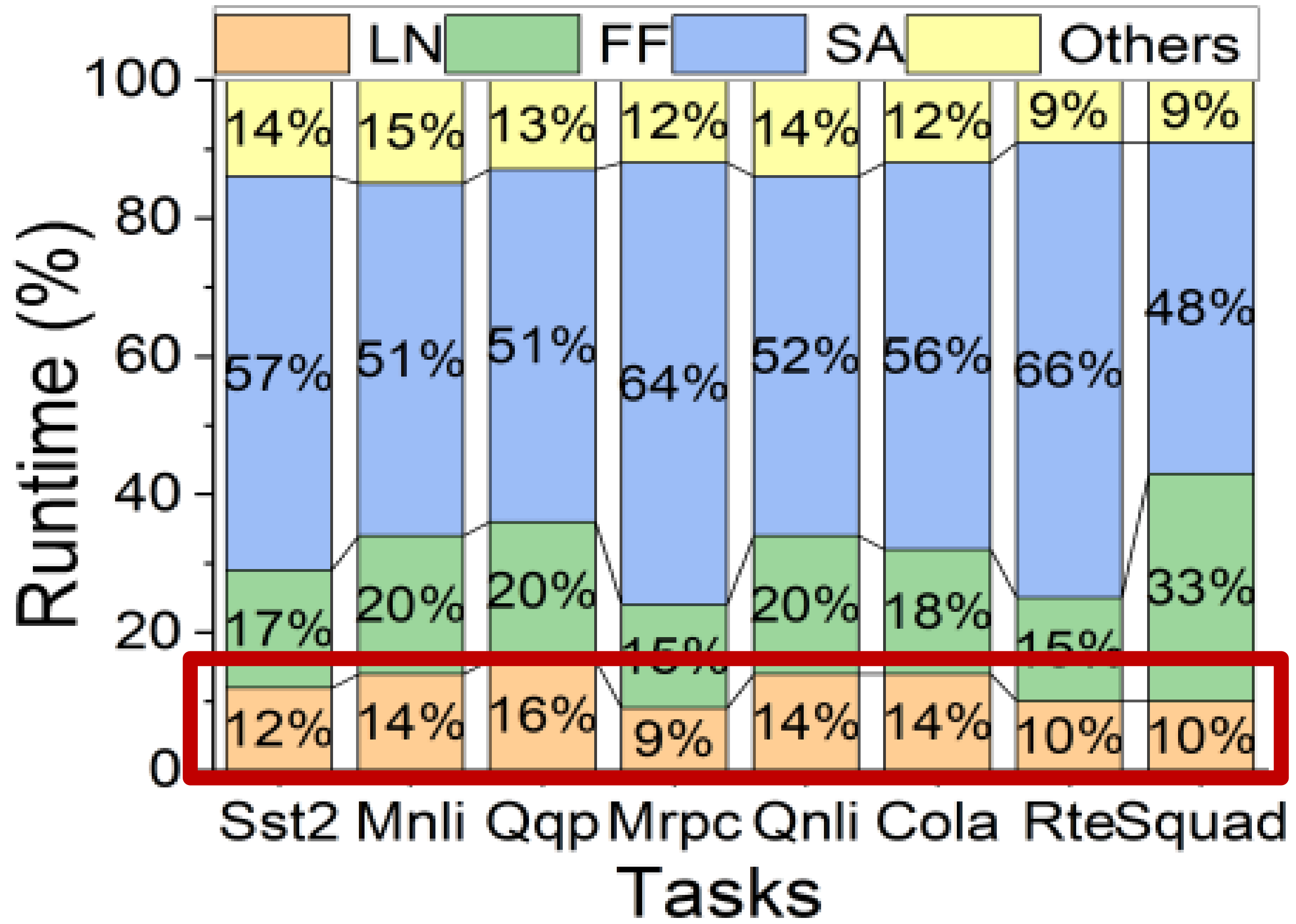


- Transformer have emerged as the choice learning model for NLP.
- LN contributes to the overall latency of the system

Why Optimize Layer Normalization?

$$LN(x_{i,j}) = \widehat{x_{i,j}} \cdot \gamma_j + \beta_j, \qquad for\ j = 1,2, \dots, d_{model}$$

- **First iteration** $\mu_i = \frac{1}{d_{model}} \cdot \sum_{j=1}^{d_{model}} x_{i,j}$
- **Second iteration** $\sigma_i^2 = \frac{1}{d_{model}} \cdot \sum_{j=1}^{d_{model}} (x_{i,j} - \mu_i)^2$
- **Third iteration** $\widehat{x_{i,j}} = \frac{x_{i,j} - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}}, \qquad for\ i = 1,2, \dots, d_{token}$



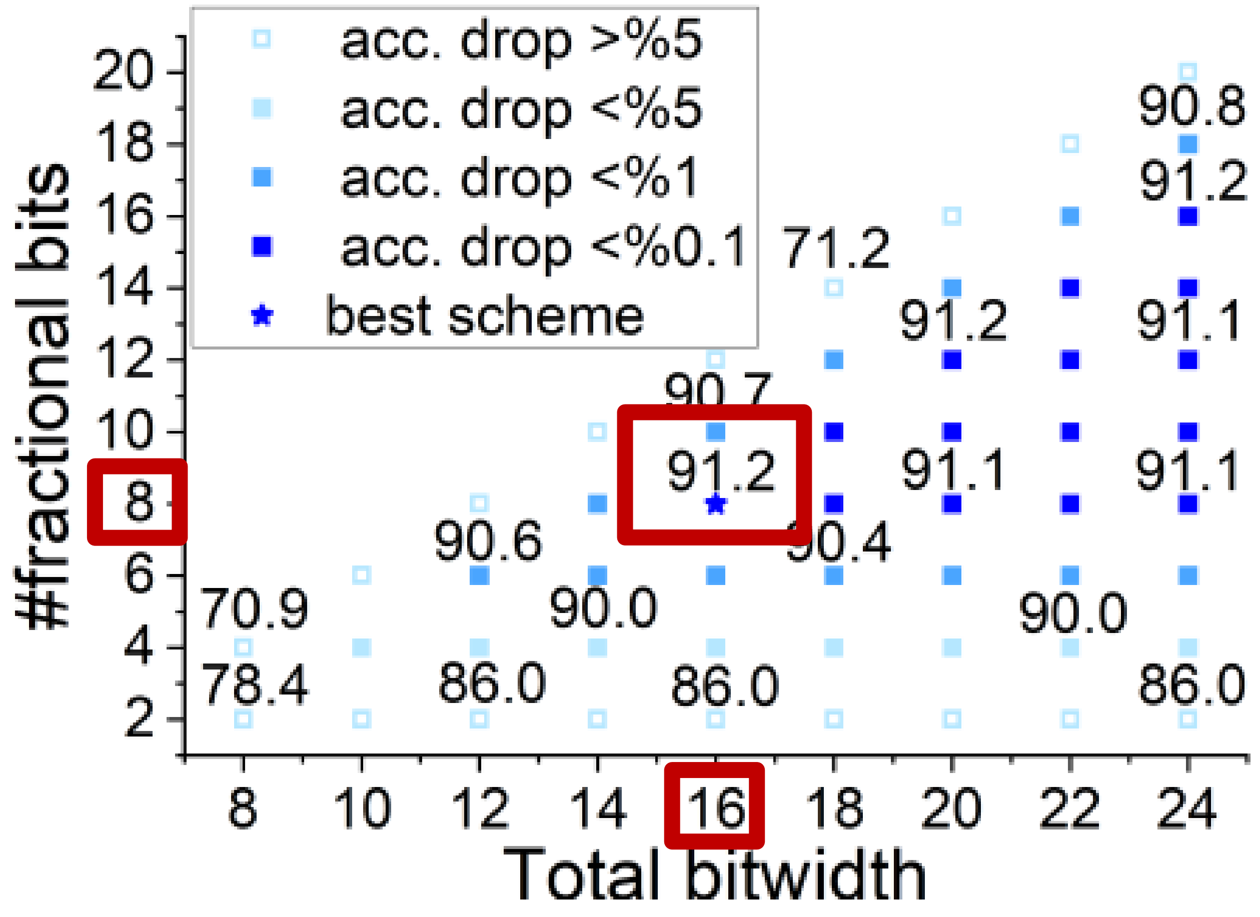
→ **hardware inefficient** due to the **square root** and **division operations**

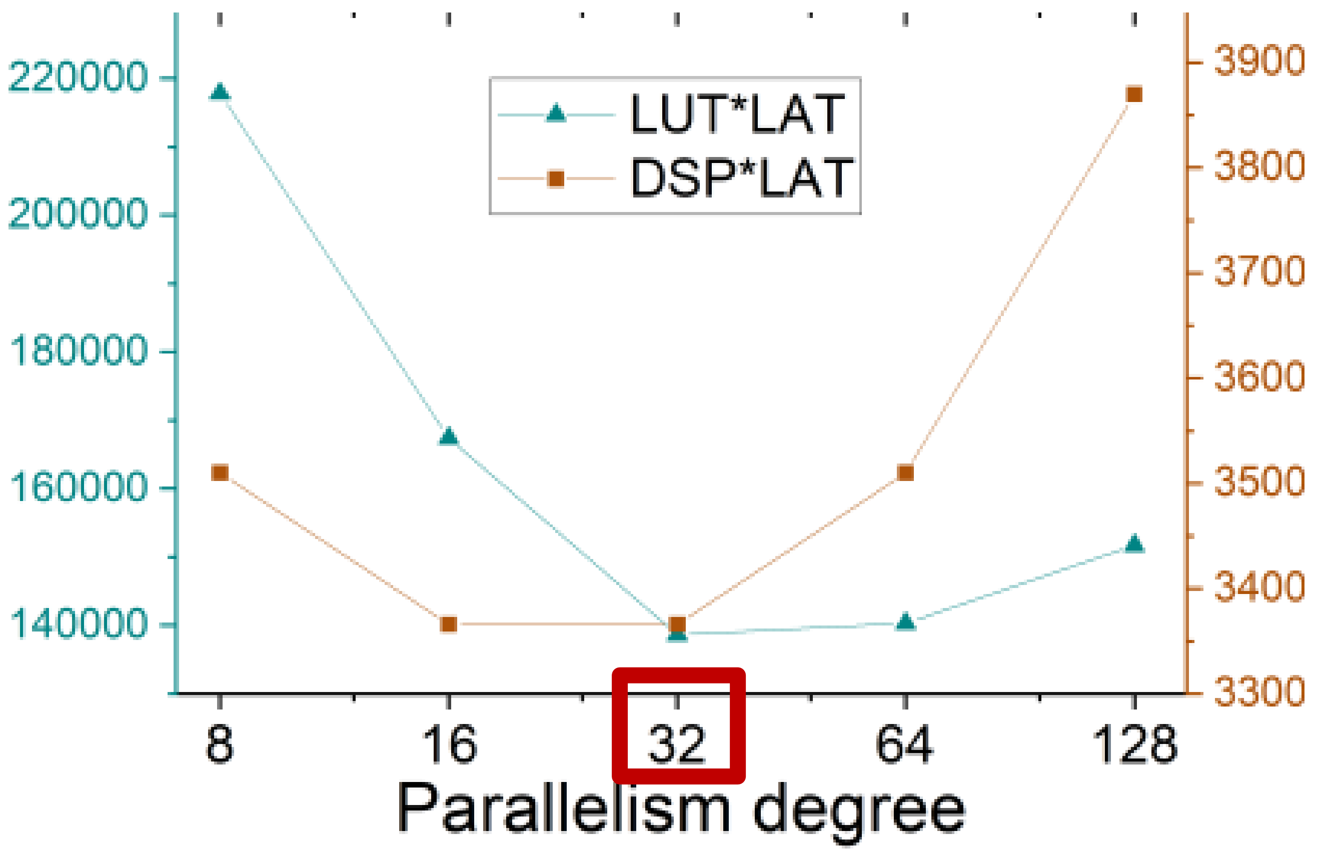
→ **time consuming** since it needs **three iterations** on the input data

What This Paper Proposes?

- Hardware-efficient LN approximation **without fine-tuning**
- **Key contributions:**
  - Fixed-point quantization & Parallelism exploration
  - Standard Layer Normalization(PWL)
  - Pairwise Variance Algorithm

Fixed-point quantization & Parallelism exploration

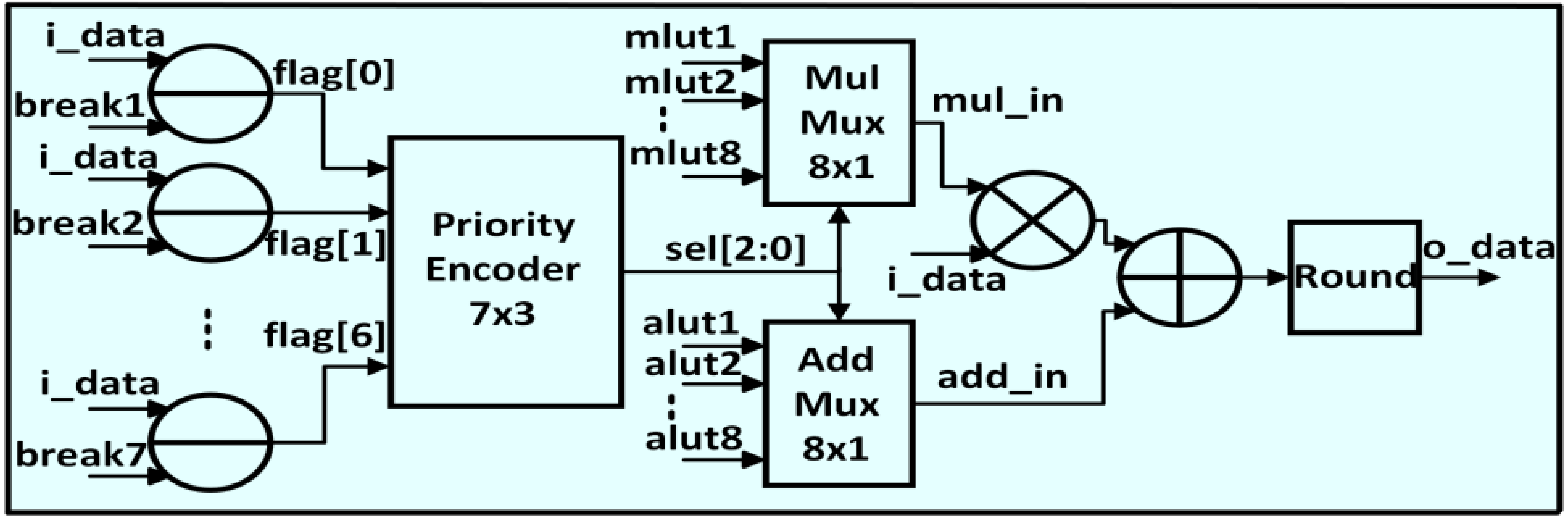




- Replaces 32-bit float with **16-bit fixed point** (8 integer + 8 fractional bits)
- Maintains accuracy (≤ 0.28% drop)

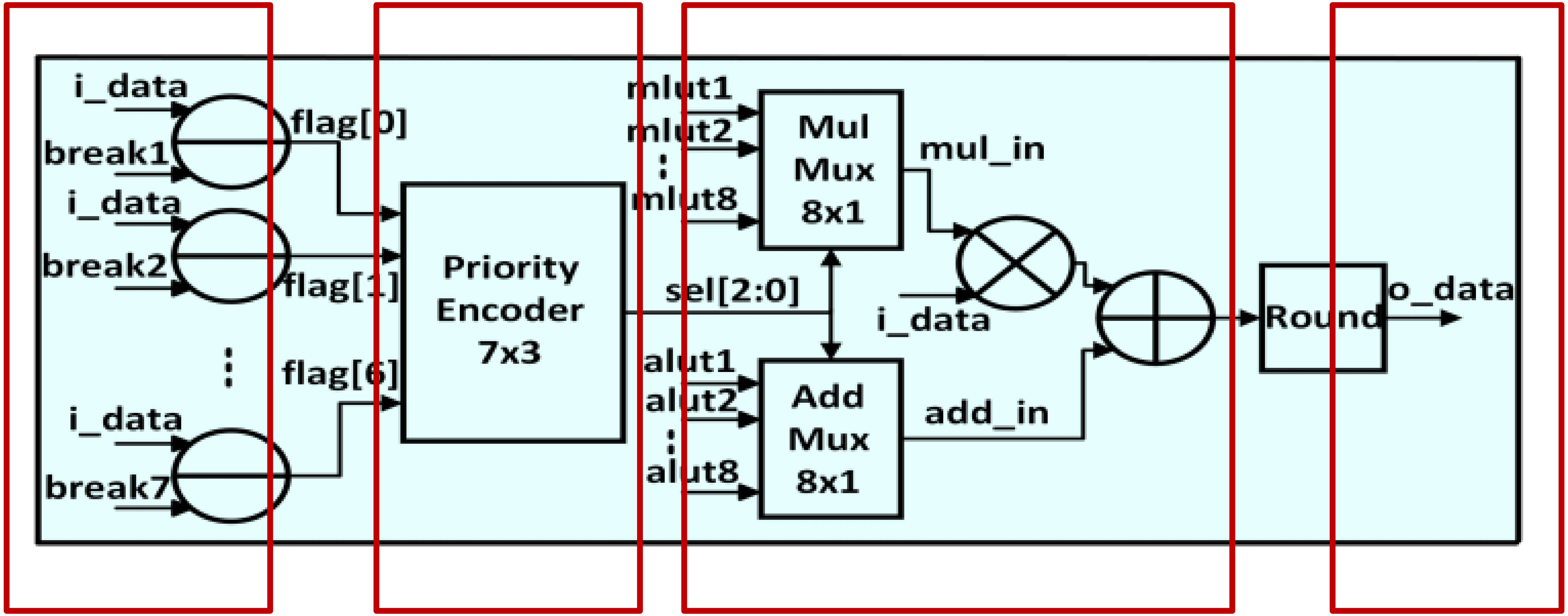
- The **best area-time trade-off** is found to be the **32-input** design

Standard Layer Normalization(PWL)



- Sqrt and reciprocal approximated using **7 non-uniform breakpoints (8 line segments)**
- Executes in **1 clock cycle**

Standard Layer Normalization(PWL)



Breakpoints
(→ 8 segments)

Input data **x**
→ Select one of
8 segments using
a 3-bit index [2:0]

PWL Approximation
using the function **pwl_approx()**

Output $\boldsymbol{y} = a_3x + b_3$

Standard Layer Normalization(PWL)

- **using pwlf python lib**
- **pwlf library**: Automatically finds optimal breakpoints for piecewise linear fitting

```
# PWL fit
x_vals = np.linspace(0.01, 128, 1000)
sqrt_vals = np.sqrt(x_vals)
recip_vals = 1 / sqrt_vals

sqrt_model = pwlf.PiecewiseLinFit(x_vals, sqrt_vals)
sqrt_breaks = sqrt_model.fit(8)
sqrt_slopes = sqrt_model.slopes  a
sqrt_intercepts = sqrt_model.intercepts  b

recip_model = pwlf.PiecewiseLinFit(x_vals, recip_vals)
recip_breaks = recip_model.fit(8)
recip_slopes = recip_model.slopes
recip_intercepts = recip_model.intercepts
```

Input : 0.01 ~ 128
Precomputing sqrt and reciprocal

Fits a piecewise linear function with 8 segments

Standard Layer Normalization(PWL)

```
# PWL approximation
def pwl_approx(x, breakpoints, slopes, intercepts):
    x = np.clip(x, breakpoints[0], breakpoints[-1])
    out = np.zeros_like(x)
    for i in range(len(slopes)):
        mask = (x >= breakpoints[i]) & (x < breakpoints[i + 1])
        out[mask] = slopes[i] * x[mask] + intercepts[i]
    out[x >= breakpoints[-1]] = slopes[-1] * x[x >= breakpoints[-1]] + intercepts[-1]
    return out
```

If input the data $\boldsymbol{x}$ **in breackpoint** $[\boldsymbol{x_3}, \boldsymbol{x_4})$

Then output the result $\boldsymbol{y = a_3 x + b_3}$

→ Output can be calculated in a **linear approximation**

→ Reduced latency

Standard Layer Normalization(PWL)

Result Evaluation

Latency

```
===== Timing (ms) =====
[Sqrt Exact]   0.0268 ms
[Sqrt PWL]     0.2562 ms
[Recip Exact]  0.0106 ms
[Recip PWL]    0.1365 ms
```

PWL > True

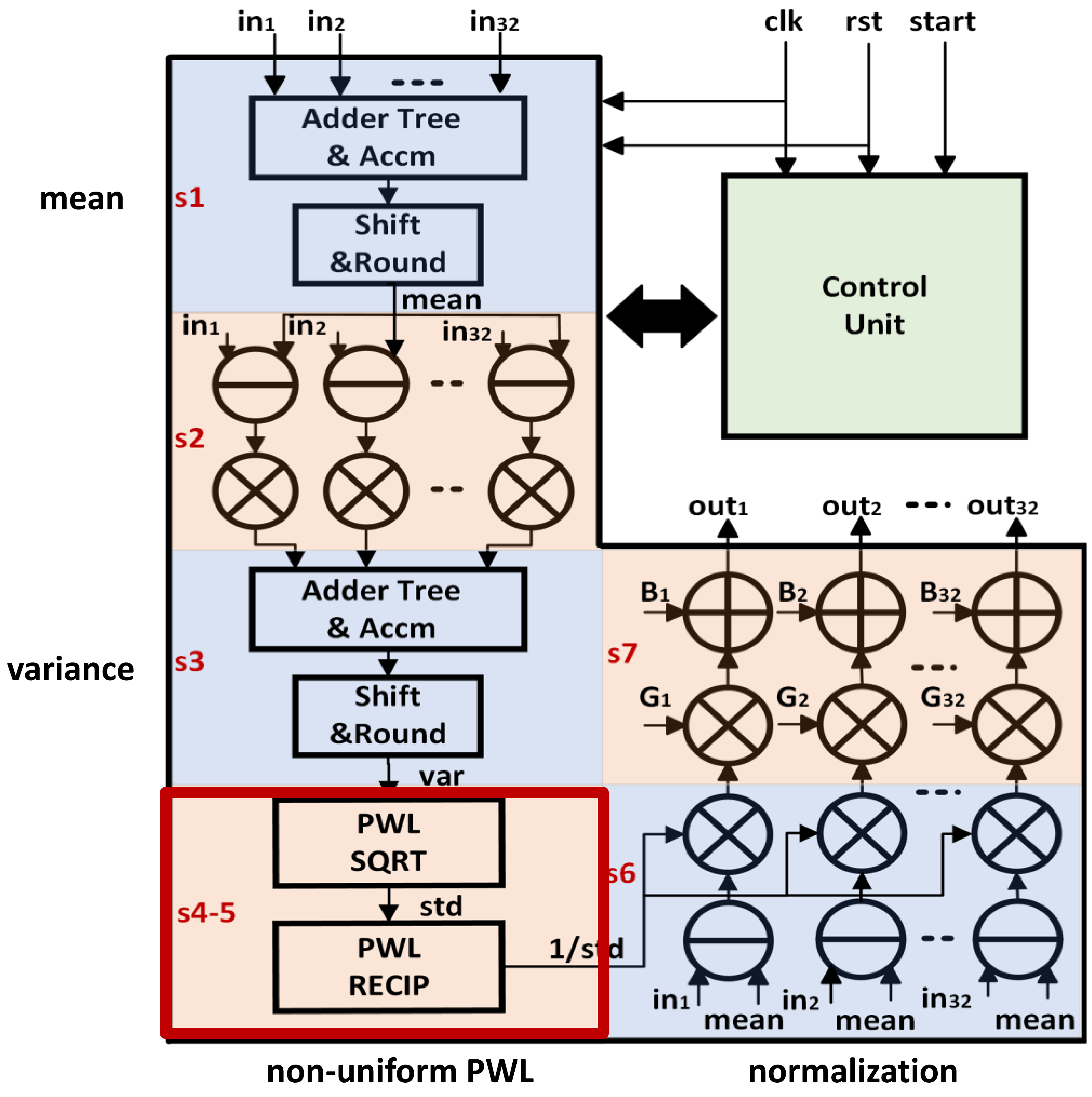
Latency increased due to **software (not FPGA)** implementation.

Accuracy

```
===== Accuracy (% Error) =====
[Sqrt PWL]     Mean Accuracy: 99.1760%
[Recip PWL]    Mean Accuracy: 97.9223%
```

sqrt and reciprocal PWL approximations achieved over **95% accuracy.**

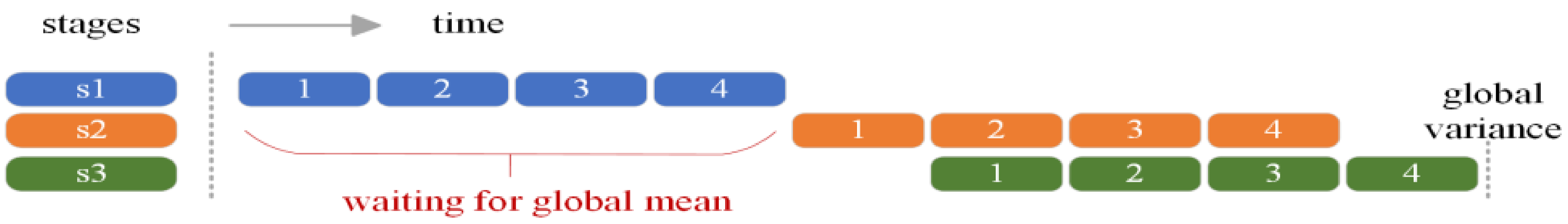
Standard Layer Normalization(PWL)



Total number of input: 512

Each batch: 32 elements

Each element: 16-bit fixed point



s2 must wait for the result of s1 until all inputs are presented

Pairwise Variance Algorithm

Algorithm:

the input data can be split into groups → $\delta_{1(2)} = \mu_{1(3)} - \mu_{2(4)}$ $\quad \delta = \frac{\mu_1 + \mu_2}{2} - \frac{\mu_3 + \mu_4}{2}$

the variance of each group can be calculated separately → $intVar_{1(2)} = \sigma^2_{1(3)} + \sigma^2_{2(4)} + \delta^2_{1(2)} \times \frac{n_{1(3)} \times n_{2(4)}}{n_{1(3)} + n_{2(4)}}$

intermediate values are used to calculate the global variance → $\sigma^2 = intVar_1 + intVar_2 + \delta^2 \times \frac{(n_1 + n_2) \times (n_3 + n_4)}{n_1 + n_2 + n_3 + n_4}$

Pairwise Variance Algorithm

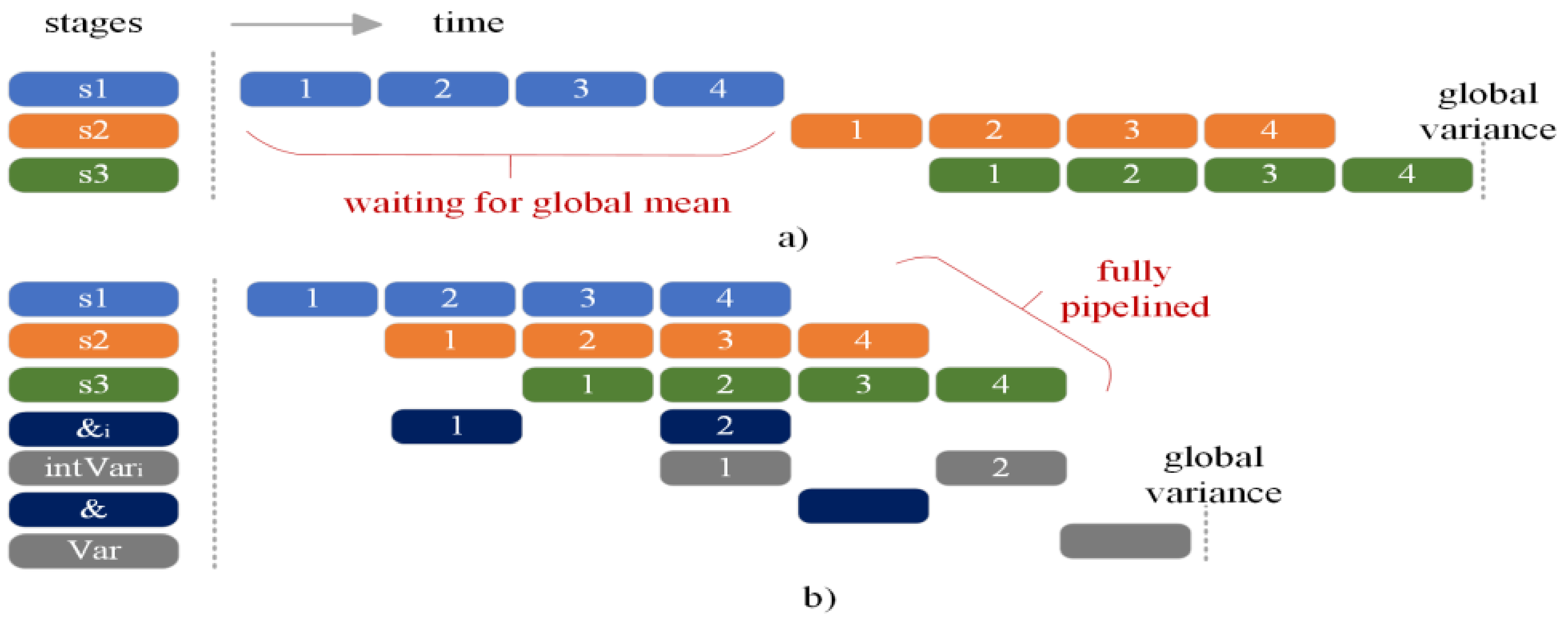


Fig. 5. Exemplified timing diagrams for a) standard variance b) pairwise variance calculation.

- Avoids waiting for global mean → reduces iteration count from 3 to 2
- Enables **fully pipelined architecture** for s1–s3 stages

→ Only one more clock cycle to calculate the global variance

* No change to the remaining hardware components of s4-7

Pairwise Variance Algorithm

Standard variance calculation

```
# Variance methods
def true_variance(x):
    mean = x.mean(axis=-1, keepdims=True)
    return ((x - mean) ** 2).mean(axis=-1, keepdims=True)
```

$*(1)$

: Two sequential divisions

→ major reason of latency

$*(1)\ \sigma_i^2 = \frac{1}{d_{model}} \sum_{j=1}^{d_{model}} (x_{i,j} - \mu_i)^2$

One-pass variance algorithm

```
def one_pass_variance(x):
    mean = x.mean(axis=-1, keepdims=True)
    mean_sq = (x**2).mean(axis=-1, keepdims=True)
    return mean_sq - mean**2
```

$*(2)$

: Reduces the computation to a single pass

→ lowers the latency

→ Increase the risk of floating-point errors

$*(2)\ \sigma_i^2 = \frac{1}{d_{model}} \sum_{j=1}^{d_{model}} x_{i,j}^2 - \mu_i^2$

Pairwise Variance Algorithm:

- Input is split into 16 groups
- Variance is calculated in a hierarchical fashion:16 → 8 → 4 → 2 → 1

→ Fully pipelined architecture leads to reduced overall latency

Pairwise Variance Algorithm

Pairwise Variance Algorithm

```python
def pairwise_variance(x):
    # Number of groups to split into (must be a power of two)
    G = 16
    D = x.shape[-1]
    # Ensure the last dimension is divisible by G
    assert D % G == 0, f"Last dim {D} must be divisible by {G}"

    # Split the input tensor into G equal parts along the last axis
    splits = np.split(x, G, axis=-1)

    # Compute per-group counts, means, and sum of squared deviations:
    # M_i = sum_j (x_ij - mu_i)^2 = n_i * var_i
    n_list = [s.shape[-1] for s in splits]  # number of elements in each group
    mu_list = [s.mean(axis=-1, keepdims=True) for s in splits]  # group means (mu_i)
    M_list = [
        np.var(s, axis=-1, keepdims=True, ddof=0)
        * n  # group sum of squared deviations (M_i)
        for s, n in zip(splits, n_list)
    ]
```

Number of groups: 16

N_list: Number of elements in each group

Mu_list: Group means

M_list: Variance * n

Pairwise Variance Algorithm

Pairwise Variance Algorithm

```
# Iteratively merge groups in pairs: 16 -> 8 -> 4 -> 2 -> 1
while len(mu_list) > 1:
    next_mu, next_M, next_n = [], [], []
    for i in range(0, len(mu_list), 2):
        # Grab two adjacent groups
        μ1, μ2 = mu_list[i], mu_list[i + 1]
        M1, M2 = M_list[i], M_list[i + 1]
        n1, n2 = n_list[i], n_list[i + 1]

        # Compute merged sum of squared deviations:
        # M_12 = M1 + M2 + (mu1 - mu2)^2 * (n1 * n2) / (n1 + n2)
        delta = μ1 - μ2  *(1)
        M12 = M1 + M2 + delta**2 * (n1 * n2) / (n1 + n2)  *(2)
        next_M.append(M12)

        # Compute merged mean and count:
        # mu_12 = (n1 * mu1 + n2 * mu2) / (n1 + n2)
        next_mu.append((μ1 * n1 + μ2 * n2) / (n1 + n2))
        next_n.append(n1 + n2)

    # Prepare for next iteration
    mu_list, M_list, n_list = next_mu, next_M, next_n
```

- Start with 16 groups and iteratively merge them until only 1 group remains
- At each step, take adjacent values of μ, n, and M to compute δ_i and $intVar_i$
- Then, **update μ and n** and store them in the list for the next iteration.

$* (1)\ \delta_{1(2)} = \mu_{1(3)} - \mu_{2(4)}$

$* (2)\ intVar_{1(2)} = \sigma^2_{1(3)} + \sigma^2_{2(4)} + \delta^2_{1(2)} \times \dfrac{n_{1(3)} \times n_{2(4)}}{n_{1(3)} + n_{2(4)}}$

Pairwise Variance Algorithm

Pairwise Variance Algorithm

```
# After merging, compute final biased variance:
M_total = M_list[0]  # total sum of squared deviations
n_total = n_list[0]  # total number of elements
var_total = M_total / n_total  # biased variance = M_total / n_total
return var_total
```

M_list = variance * n

→ **Total variance =** $\frac{\mathrm{M_{total}}}{n_{total}}$

Pairwise Variance Algorithm

Result Evaluation

Latency

```
===== Timing (ms) =====
[True Var]      1.1087 ms
[One-Pass Var] 0.1928 ms
[Pairwise Var] 2.2948 ms
```

One-Pass < True < Pairwise

Accuracy

```
===== Accuracy (% Error vs PyTorch) =====
[True Var]     100.0000%
[One-Pass Var] 100.0000%
[Pairwise Var] 100.0000%
```

One-Pass = True = Pairwise

Conclusion

- Proposed two LN hardware accelerators:
  - Standard Layer Normalization(PWL)

    → reduced the latency almost 3x
  - Pairwise variance algorithm

    → reduced the latency almost 4x

    → additional **27%** over standard LN (slightly more hardware resources)
- No fine-tuning required
- No retraining required