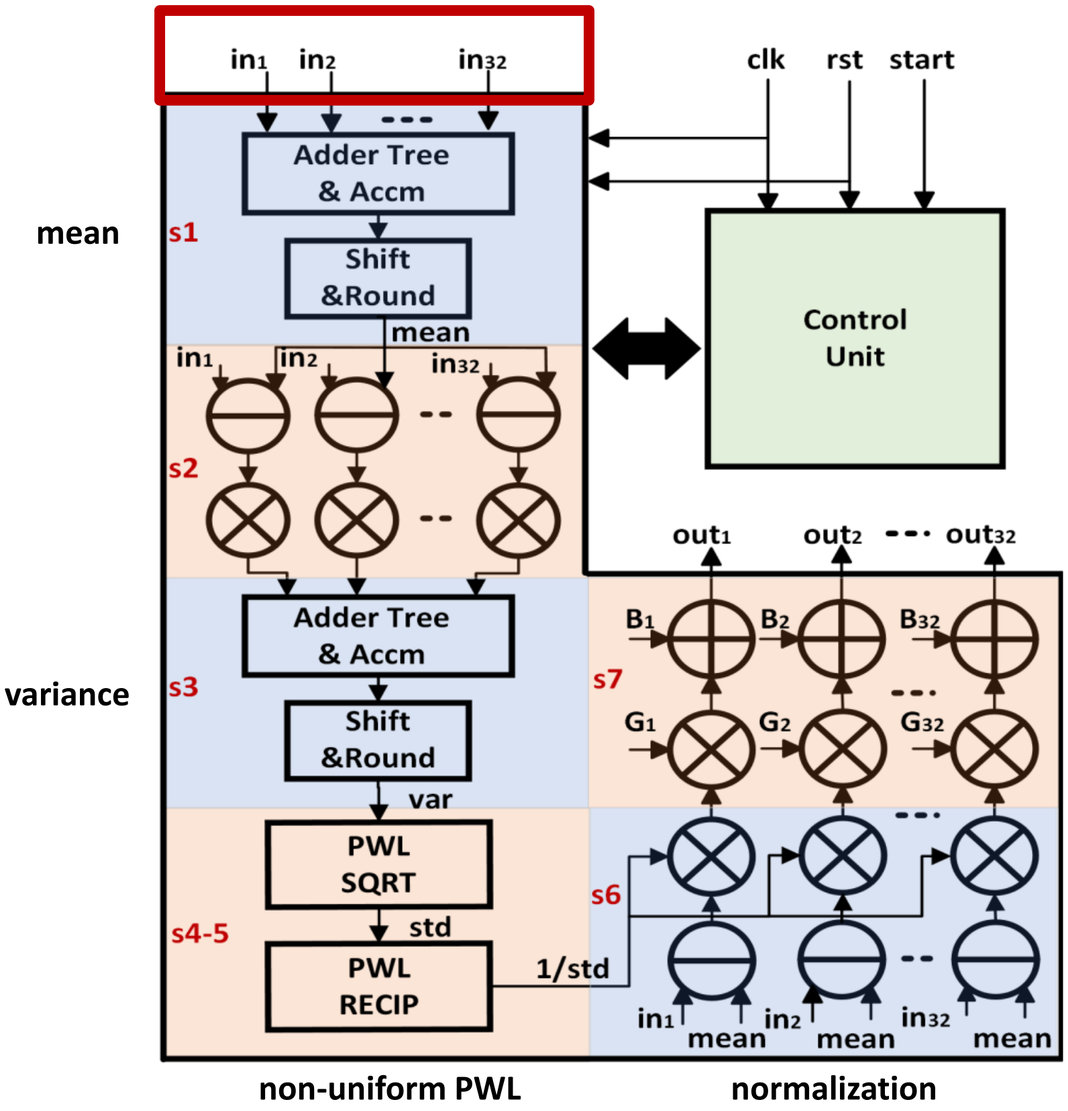
LayerNorm Approximation via Software-Hardware

2025.7.08.

Sangyun Kim
Eunju Kim

CONTENTS

Standard Layer Normalization(PWL)



Input Data Type

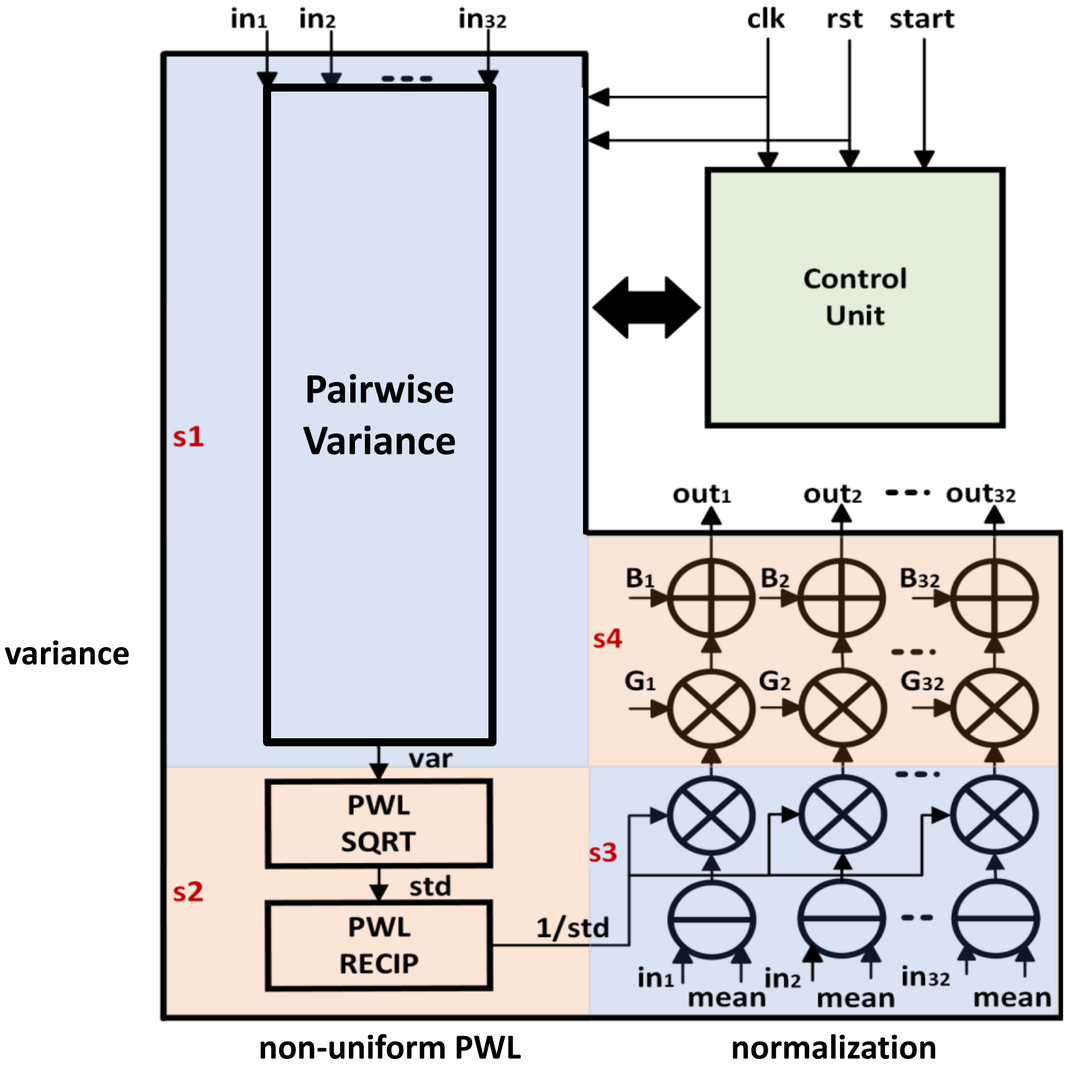
Q8.8 (8 integer, 8 fractional)

Data Layout

Each batch: 32 elements

Each element: 16-bit fixed point

PWL-Based LayerNorm with Pairwise Variance Estimation



Allows **merging s1–s3** into a single block.

Replaces s4 with s2 for full pipelining.

Pairwise Variance Algorithm

Previous Pairwise Variance Code

```
# 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
```

- Same Algorithm
  - 16 groups and only 1 group remains
  - each step calculate μ, n, δ_i, intVar_i.
  - **update μ and n** and store.

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

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

- **Not consider appoximation**

Pairwise Variance Algorithm(s1)

Revise Pairwise Variance Code

```python
def pairwise_variance_q8_8(x_q, G=16):
    D = x_q.shape[-1]
    splits = np.split(x_q, G, axis=-1)

    n_list = [s.shape[-1] for s in splits]
    mu_list = [np.sum(s, axis=-1, keepdims=True) // s.shape[-1] for s in splits]
    M_list = [
        np.sum(((s - mu).astype(np.int64)) ** 2, axis=-1, keepdims=True)
        for s, mu in zip(splits, mu_list)
    ]

    while len(mu_list) > 1:
        next_mu, next_M, next_n = [], [], []
        for i in range(0, len(mu_list), 2):
            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]
            n_total = n1 + n2
            delta = μ1 - μ2  * (1)
            delta_sq = (delta.astype(np.int64)) ** 2

            n_total_shift = int(np.log2(n_total))
            delta_term = (delta_sq * n1 * n2) >> n_total_shift    * (2)

            M12 = M1 + M2 + delta_term
            mu12 = (μ1 * n1 + μ2 * n2) >> n_total

            next_mu.append(mu12)
            next_M.append(M12)
            next_n.append(n_total)
        mu_list, M_list, n_list = next_mu, next_M, next_n

    var_q16 = M_list[0] // D
    return q8_8_to_float(var_q16 >> 8)
```

Revise Part of Code

- Approximated to match the Q8.8 input format.
- Only **addition**(**subtraction**), **multiplication**, and **bit-shift** operations are used.
- **Division is eliminated.**

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

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

Pairwise_variance computes global variance of input

LUT(Look-Up Table)

LUT Parameter Generation Code

```python
# Generate data
x_vals = np.linspace(0.01, 128, 2000)   # Input domain: avoid zero for stability
sqrt_vals = np.sqrt(x_vals)             # True sqrt values
recip_vals = 1 / sqrt_vals              # True reciprocal sqrt values

# linear model(sqrt)
sqrt_model = pwlf.PiecewiseLinFit(x_vals, sqrt_vals)
sqrt_breaks = sqrt_model.fit(8)         # Fit with 8 segments
np.savez(                               # Save model parameters as .npz
    "pwl_sqrt.npz",
    breaks=sqrt_model.fit_breaks,
    slopes=sqrt_model.slopes,a
    intercepts=sqrt_model.intercepts, b
)

# linear model(recip)
recip_model = pwlf.PiecewiseLinFit(x_vals, recip_vals)
recip_breaks = recip_model.fit(8)       # Fit with 8 segments
np.savez(                               # Save model parameters as .npz
    "pwl_recip.npz",
    breaks=recip_model.fit_breaks,
    slopes=recip_model.slopes, a
    intercepts=recip_model.intercepts,b
)
```

- Input domain from **[0.01, 128]** into **8 segments.**
- Python LUT parameters saved as **".npz".**
- Verilog will be converted to **".mem".**

Each file returns the **coefficients a_i and b_i**

for the piecewise linear approximation

of the **square root** and **reciprocal square root** functions

over the **input variance.**

PWL approximation (S2)

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

- Input **variance** $\boldsymbol{x}$(result of Pairwise), **PWL model** approximates the **sqrt** and **1/sqrt** functions.
- **LUT** performs the **actual computation** using slope(a_i) and intercept(b_i) values from the **PWL.**
- result the **LUT outputs** an **approximated value of 1 / sqrt(variance)**.

$$\boldsymbol{y} = a_i x + b_i \rightarrow \boldsymbol{y} = \frac{1}{\sqrt{\sigma_i^2 + \epsilon}}$$

Final Normalization Step(S3)

```
def approx_layernorm_q8_8(x_q, eps=1e-5):
    mu_q = np.mean(x_q, axis=-1, keepdims=True)
    x_centered = x_q - mu_q
    var_approx = pairwise_variance_q8_8(x_q)
    sqrt_val = pwl_approx(var_approx + eps, sqrt_breaks, sqrt_slopes, sqrt_intercepts)
    recip_val = pwl_approx(sqrt_val, recip_breaks, recip_slopes, recip_intercepts)
    x_float = q8_8_to_float(x_centered)
    return x_float * recip_val
```

- Using the results from S1 and S2.
- The input data is normalized by **subtracting the mean**

  and **multiplying** by the reciprocal of standard deviation.

$$\widehat{x_{i,j}} = \frac{x_{i,j} - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}}, \qquad for\ i = 1,2,\dots,d_{token}$$

LayerNorm Approximation: Performance Comparison

- Accuracy drop was evaluated on 10 samples from **SST-2, QNLI** and **MNLI** using our LN approximation.
- Both the original paper's results and our implementation were **compared** against the **PyTorch baseline.**

Setting

Input data samples: 10 each from **SST-2, QNLI, and MNLI**

Data type: **Q8.8** (8 integer bits, 8 fractional bits)

Model input dimension: **768** (GPT-2 compatible)

```
bert_sst2_embed.npy : 99.4362%
bert_qnli_embed.npy : 99.5774%
bert_mnli_embed.npy : 99.3592%
bert_sst2_embed_100.npy : 99.5190%
distilbert_sst2_embed.npy : 99.3935%
distilbert_qnli_embed.npy : 99.5161%
distilbert_mnli_embed.npy : 99.6359%
distilbert_sst2_embed_100.npy : 99.4406%
```

Accuracy Difference on NLP Tasks (PyTorch, %) : Paper vs. Ours						
	SST2(Paper)	SST2(Our)	QNLI(Paper)	QNLI(Our)	MNLI(Paper)	MNLI(Our)
Bert Base	99.62%	99.44%	99.97%	99.57%	99.65%	99.35%
Distill Bert	99.87%	99.33%	99.47%	99.34%	99.73%	99.55%

LayerNorm Approximation: Performance Comparison

- When the number of samples for SST-2 was increased from 10 to 100

Setting

Input data samples: 10 vs 100 samples from **SST-2**

Data type: **Q8.8** (8 integer bits, 8 fractional bits)

Model input dimension: **768** (GPT-2 compatible)

Accuracy Difference on NLP Tasks (PyTorch, %) : Paper vs. Ours			
	SST2(Paper)	SST2(Our: 10)	SST2(Our: 100)
Bert Base	99.62%	99.44%	99.48%
Distill Bert	99.87%	99.33%	99.40%

- Accuracy improved with more samples, showing that 10 samples were too few.
- Mismatch may come from unspecified LUT settings, `x_vals = np.linspace(0.01, 128, 2000)` but error stays under 1%, confirming correct implementation.

DFX ARCHITECHTURE

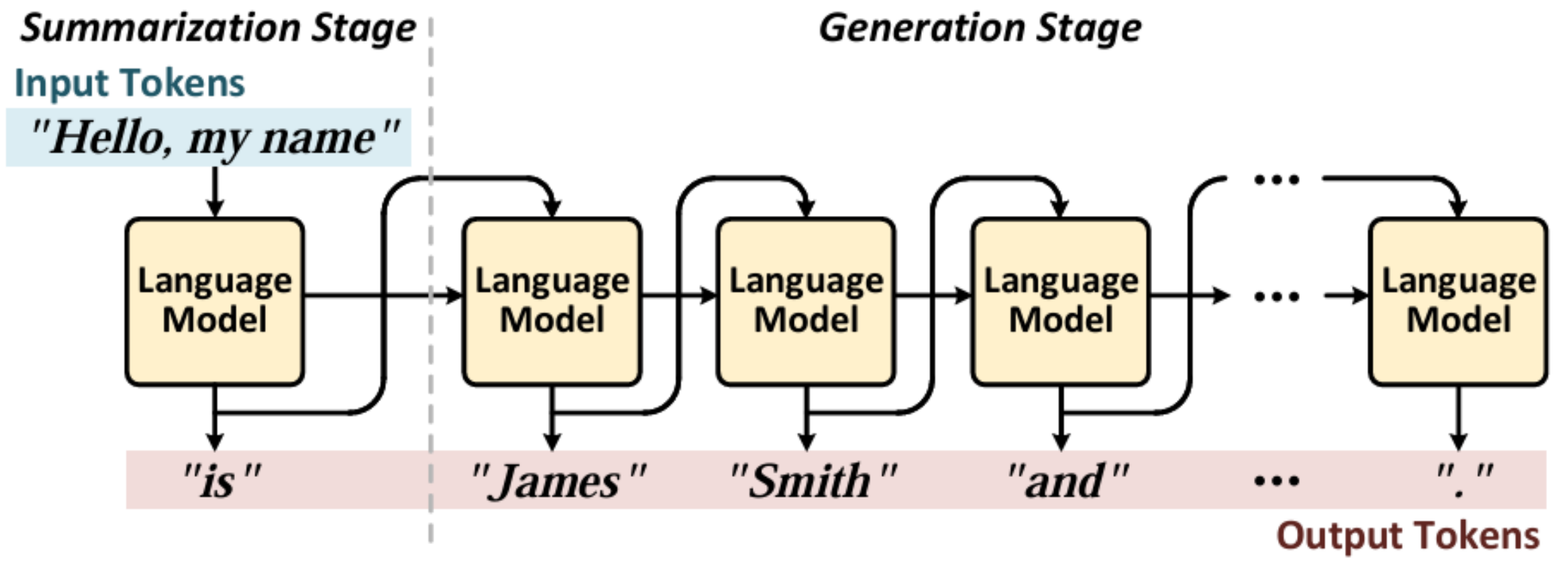


Illustration of transformer-based text generation

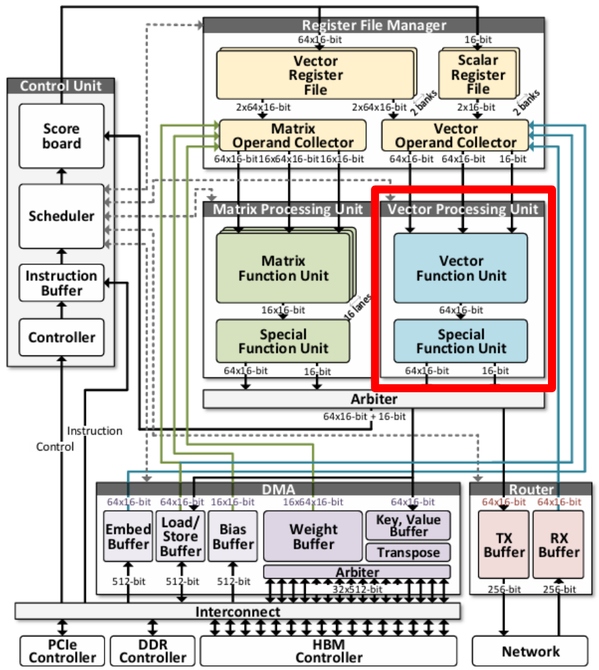
- **Two stages** of Transformer-based LLM such as GPT

  → **Summarization Stage** processes **well-suited for parallel architectures** like GPUs.

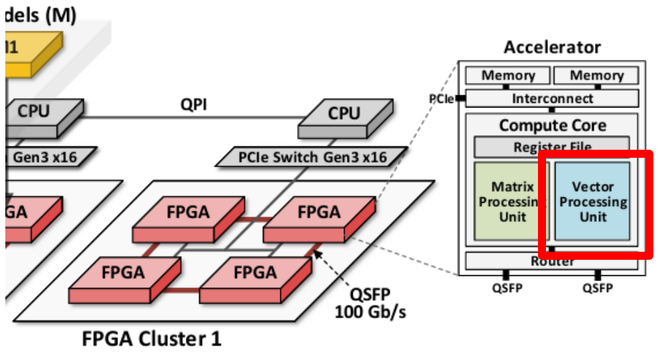
  → **Generation Stage** follows an end-to-end **sequential architectures**,

  resulting in **latency** due to limited parallelism.

DFX ARCHITECHTURE(LN)



DFX core microarchitecture



accelerator's microarchitecture

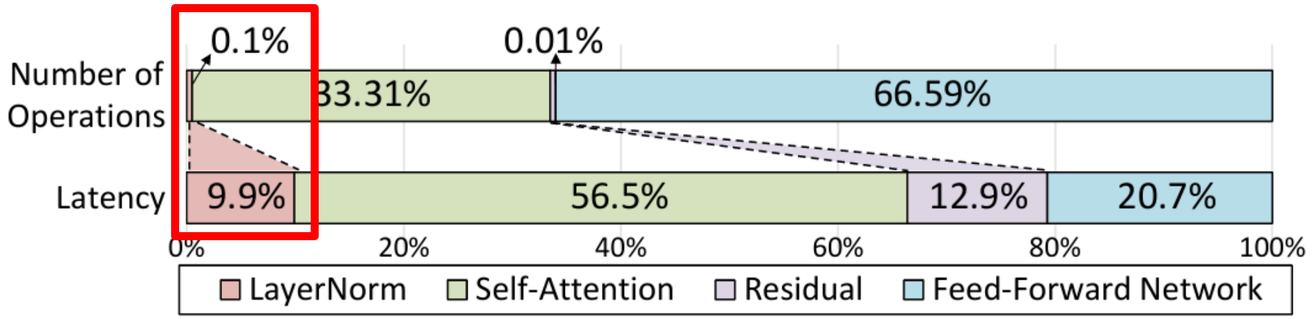
DFX Architecture Key Features

- DFX compute core is optimized for single token processing
- **Efficient tiling scheme and data flow**
- To address the increasing model size, DFX uses model parallel
- Transformer-based model modifications and expansions

How is the Data Flow Optimized?

LN operations are accelerated by **parallelizing** the **VPU** using **FPGAs**, enabling a custom accelerator architecture.

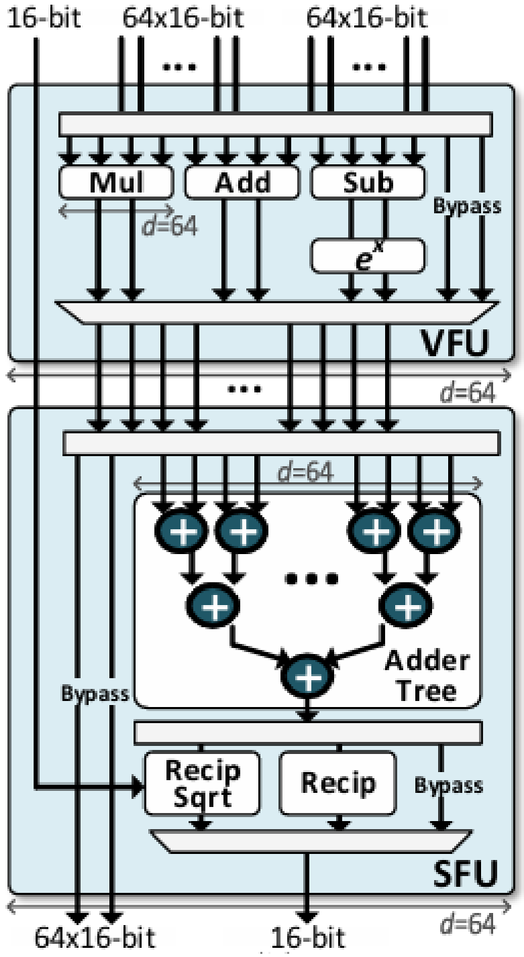
Why Apply to DFX Architecture?



GPT-2 latency and number of operations breakdown on GPU.

- LayerNorm accounts for only **0.1% of operations**
- But contributes to **9.9% of total latency**.

Modules Required for LayerNorm Acceleration



Vector Processing Unit

Module	Used in Equation	How?
Adder(Subtractor)	$x_i - \mu, etc.$	Using Verilog module
Multiplier	$(x_i - \mu)^2, etc.$	Using Verilog module
Accummulator	$\sum \boldsymbol{x_i}$	Using Verilog module
Reciprocal	$\mu = \frac{1}{N}\sum x_i$	Using LUT
Reciprocal Sqrt	$\frac{x_i - \mu}{\sqrt{\sigma_i^2 + \epsilon}}$	Using LUT

- To implement the **Pairwise** and **PWL approximations**, each function is **modularized**.

Layer Normalization: Equation & Required Modules

$$y_i = \gamma_i \cdot \frac{x_i - \mu}{\sigma} + \beta_i$$

$$\mu = \frac{1}{N}\sum_{j=1}^{N} x_j$$

$$\sigma = \sqrt{\frac{1}{N}\sum_{j=1}^{N}\left(x_j - \mu\right)^2}$$

To implement Layer Normalization,

we need the followings:

- **Accumulator**
- **Multiplier**
- **Reciprocal square root**
- **Adder(Subtractor)**

- **Mean (μ):** Accumulator, Multiplier
- **Standard deviation(σ):**

Accumulator, Multiplier, Reciprocal square root

- **formula:**

Subtractor**,** Multiplier, Adder

→ In Week 2, we implemented Accumulator, Multiplier, and Adder

Accumulator

```
module accumulator #(
    parameter DATA_WIDTH = 16,
    parameter N = 16
)(
    input                       clk,
    input                       rst,
    input                       en,
    input                       valid_in,
    input      [DATA_WIDTH-1:0] in_data,
    output reg [DATA_WIDTH+4:0] sum_out,
    output reg                  done
);

    reg [3:0] counter;

    always @(posedge clk or posedge rst) begin
        if (rst) begin
            sum_out <= 0;
            counter <= 0;
            done    <= 0;
        end else if (en && valid_in) begin
            sum_out <= sum_out + in_data;
            counter <= counter + 1;
            if (counter == N-1)
                done <= 1;
        end
    end

endmodule
```

Port Descriptions

in	clk	Clock input
	rst	Reset
	en	Enable accumulation
	valid_in	Valid input flag
	in_data	16-bit unsigned input (hexadecimal format)
out	sum_out	accumulated output(20-bit)
	done	High when N inputs have been processed

- When en & valid_in are high:
  - in_data is added to sum_out
  - counter increments by 1
  - done goes high after 16 inputs (counter = 15)
- sum_out stores the accumulated result

Mean Calculation

In top_tb.v :

```
29        reg [20:0] mean_out;

64        wait (done);
65        @(posedge clk);
66        mean_out = sum_out >> 4;
67        $display(">>> DONE at T=%0t | sum=%5h | mean=%3h",
68                $time, sum_out, mean_out);
```

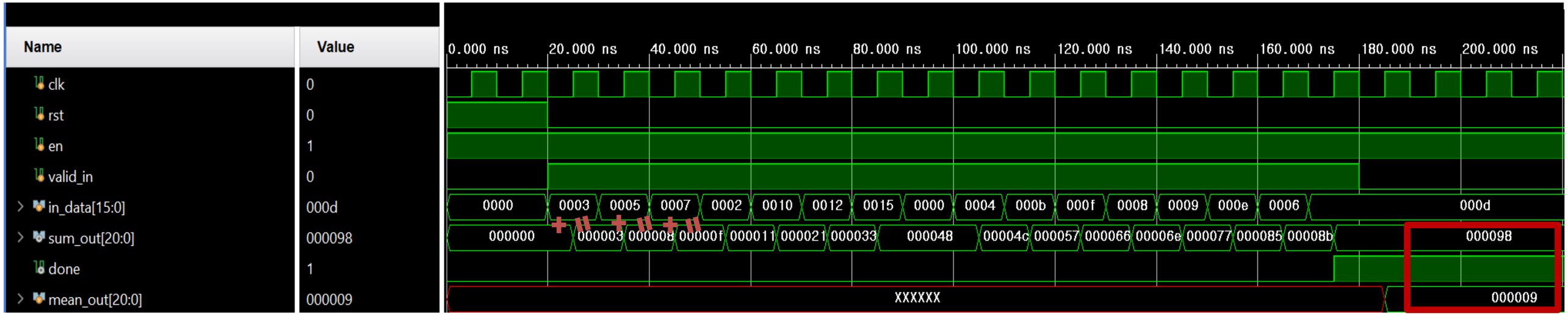
- When `done` goes high (on the rising edge of the clock)

- Compute the mean = sum_in >> 4

- The result is equivalent to dividing by 2^4(number of inputs)

- Stores the mean in `mean_out`

Simulation result: Accumulator & Mean



- The accumulated sum of `in_data` values is correctly produced through `sum_out`
- Mean: computed once when done goes high (sum >> 4)

Carry Look-ahead Adder

```
1   // N-bit Carry Look-Ahead (CLA) Adder (default: 4-bit)
2   module cla_adder #(parameter N = 4)(
3       input  [N-1:0] x,
4       input  [N-1:0] y,
5       input          cin,
6       output [N-1:0] sum,
7       output         cout
8   );
9
10  // Internal wires for propagate (p), generate (g), and carry (c)
11      wire [N-1:0] p, g;
12      wire [N:0]   c;
13
14      assign c[0] = cin; // Initialize carry-in
15      genvar i;
16
17  // Compute carry generate & carry propagate function
18      generate
19          for (i = 0; i < N; i = i + 1) begin : pg_cla
20              assign p[i] = x[i] ^ y[i];
21              assign g[i] = x[i] & y[i];
22          end
23      endgenerate
```

```
25  // Compute the carry for each stage
26      generate
27          for (i = 1; i < N + 1; i = i + 1) begin : carry_cla
28              assign c[i] = g[i-1] | (p[i-1] & c[i-1]);
29          end
30      endgenerate
31
32  // Compute the sum
33      generate
34          for (i = 0; i < N; i = i + 1) begin : sum_cla
35              assign sum[i] = p[i] ^ c[i];
36          end
37      endgenerate
38
39  // Final carry out
40      assign cout = c[N];
41
42  endmodule
```

Port Descriptions

in	x	First operand(4-bit)
	y	Second operand(4-bit)
	cin	Carry-in(1-bit)
out	sum	Sum output(4-bit)
	cout	Final carry-out(1-bit)

- Compute propagate (p = x XOR y) and generate (g = x AND y) signals for each bit
- Use generate & propagate signals to calculate carry-in for each bit position
- Compute each sum bit using: sum = p XOR carry
- Output final carry-out as cout = c[N]

Multiplier

```
1  `timescale 1ns/1ps
2
3  // Unsigned N-bit Array Multiplier (default: 4-bit)
4  module unsigned_multiplier_gen #(parameter N = 4) (
5      input [N-1:0] x,
6      input [N-1:0] y,
7      output [2*N-1:0] prod
8  );
9
10 //Internal wires for Partial Sums and Carries
11 wire sum [N-1:0][N-1:0];
12 wire carry [N-1:0][N-1:0];
13
14 genvar i, j;
15
16 generate for(i = 0; i < N; i = i + 1) begin : multiplier
17     if(i == 0)
18         for(j = 0; j < N; j = j + 1) begin
19             assign sum [j][i] = x[j] & y[i];
20             assign carry [j][i] = 0;
21         end
22     else
23         for(j = 0; j < N; j = j + 1) begin
24             if(j == 0)
25                 assign {carry[j][i], sum[j][i]} = sum[j+1][i-1] + (x[j]&y[i]);
26             else if (j == N-1)
27                 assign {carry[j][i], sum[j][i]} = (x[j]&y[i]) + carry[N-1][i-1] + carry[j-1][i];
28             else
29                 assign {carry[j][i], sum[j][i]} = (x[j]&y[i]) + sum[j+1][i-1] + carry[j-1][i];
30         end
31 end endgenerate
```

```
33 //Generate prod bits
34 generate for (i = 0; i < N; i = i + 1) begin: low_bit_multiplier
35     assign prod[i] = sum[0][i];
36 end endgenerate
37 generate for (i = 1; i < N; i = i + 1) begin: high_bit_multiplier
38     assign prod[N-1+i] = sum[i][N-1];
39 end endgenerate
40 assign prod[2*N-1] = carry[N-1][N-1];
41
42 endmodule
```

Port Descriptions

in	x	Multiplicand(4-bit)
	y	Multiplier(4-bit)
out	prod	Product of x and y(8-bit)

- **Partial Product Generation**

- Bitwise AND (x[j] & y[i]) generates partial products

- Partial products are added row by row with carry propagation

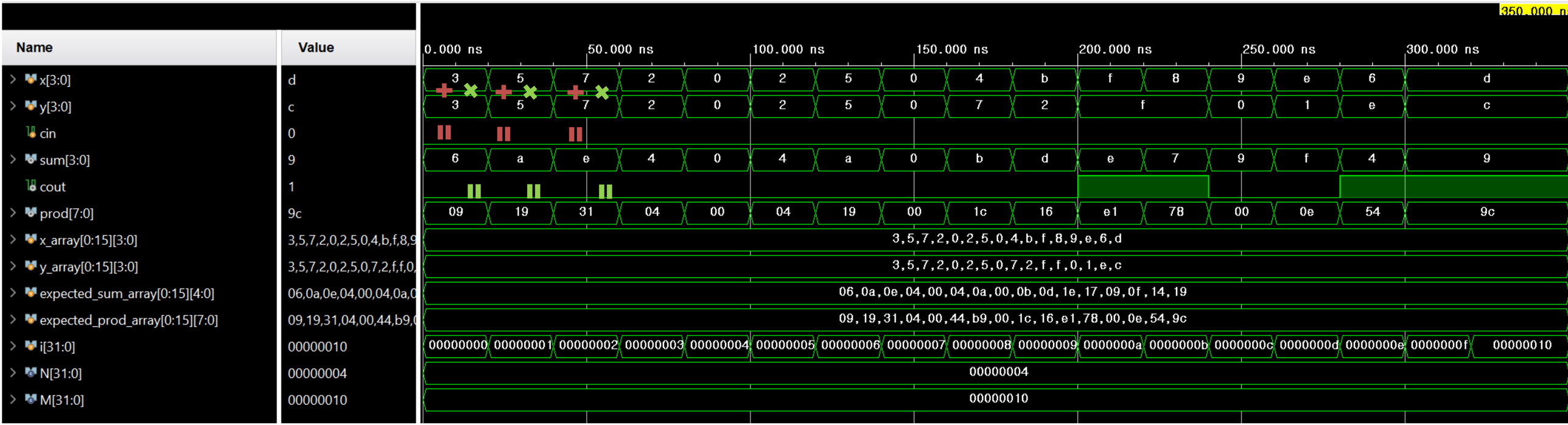
- **Generate product bits**

- prod[0:N-1]: from the first sum row

- prod[N:2N-2]: from the last column of each row

- prod[2N-1]: final carry

Simulation result: CLA Adder & Multiplier



- Sum and product results match the expected values
- CLA Adder and Multiplier operate correctly, as verified by simulation

Plans for Next

- Arithmetic module implementation (Int & Q8.8)
- Pairwise & PWL using custom operators
- Final computation using Python-generated LUT
- Latency analysis & HW optimization direction