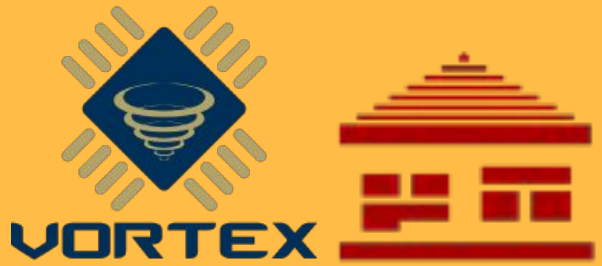
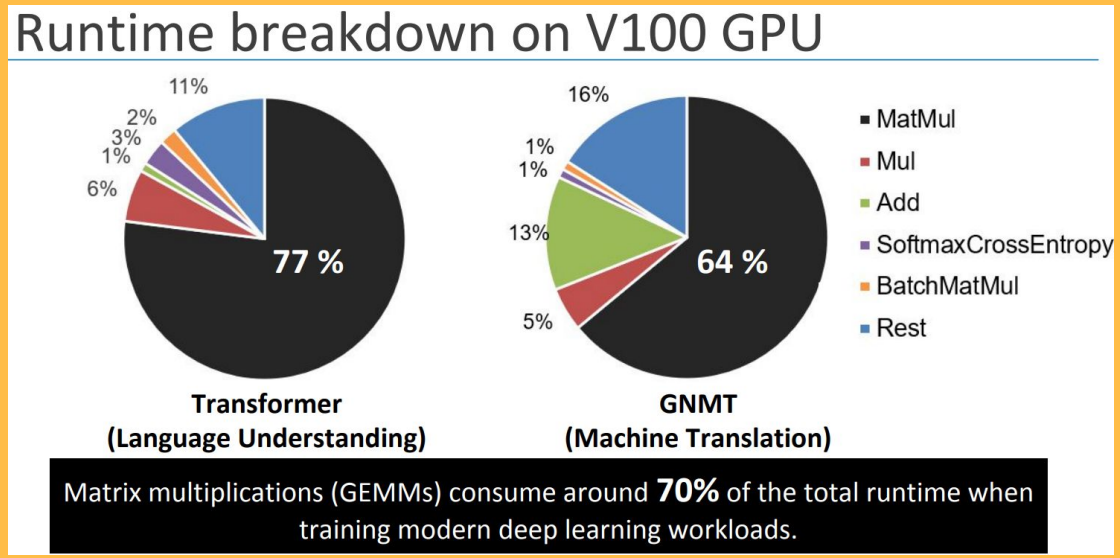
A Configurable Mixed-Precision Fused Dot Product Unit for GPGPU Tensor Computation

Nikhil Rout, Blaise Tine @ Vortex Workshop, MICRO 2025
October 18, 2025



IMPORTANCE OF MATMUL

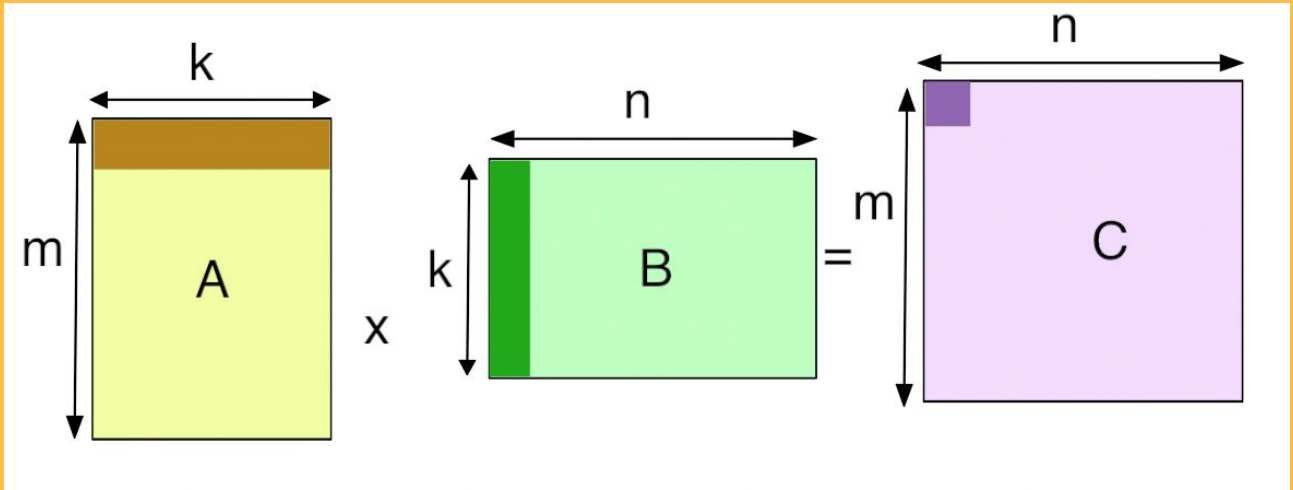
- Fundamental operation in
  - Deep Neural Networks
  - Transformers/Natural Language Processing Models



https://ieeexplore.ieee.org/document/9065523

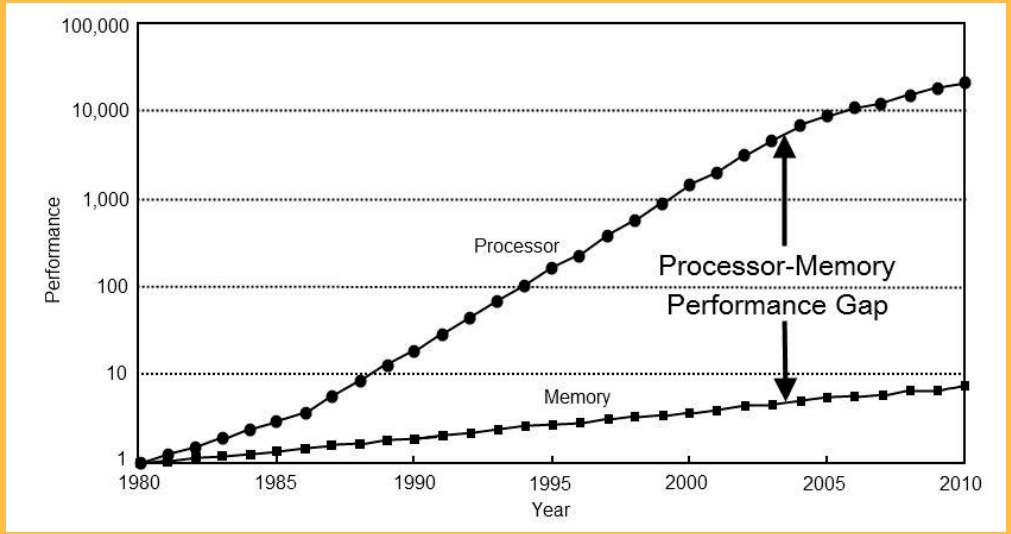
IMPORTANCE OF MATMUL

- Matrix multiplication is compute-intensive
  - For a matrix multiplication of size MxK * KxN
  - Compute cost is O(M K N) FLOPS



MOTIVATION FOR TENSOR CORES

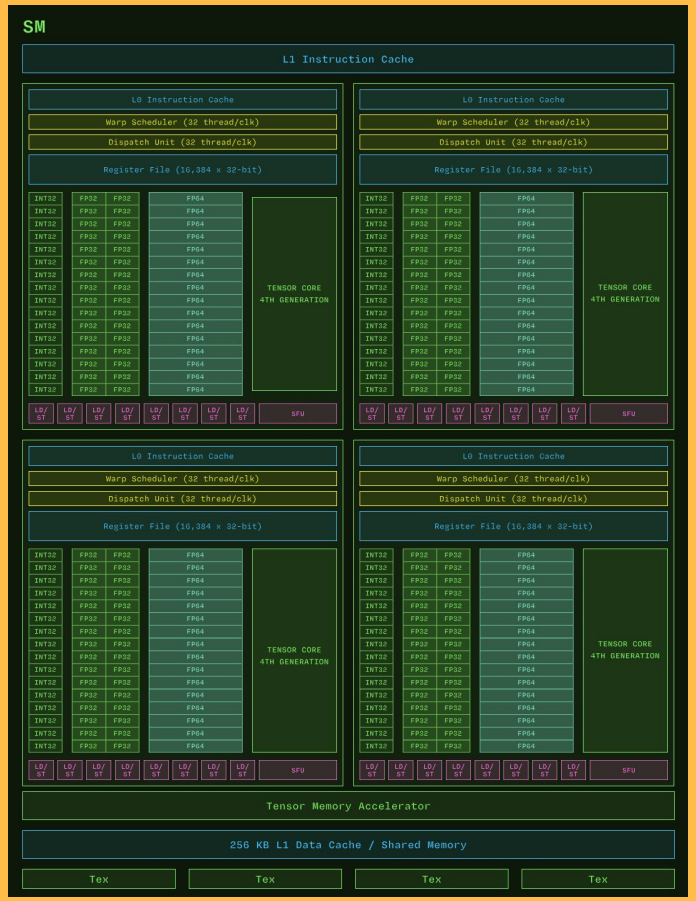
- Memory BW is our biggest bottleneck; grows much slower than compute BW
  - Solution?
    - Introduce MatMul specific functional unit within GPGPU pipeline
    - Reduced Register Pressure (don't need to store intermediates)
    - Optimizes loaded memory utilization
    - Enables fusing operations → reduce critical path & rounding error

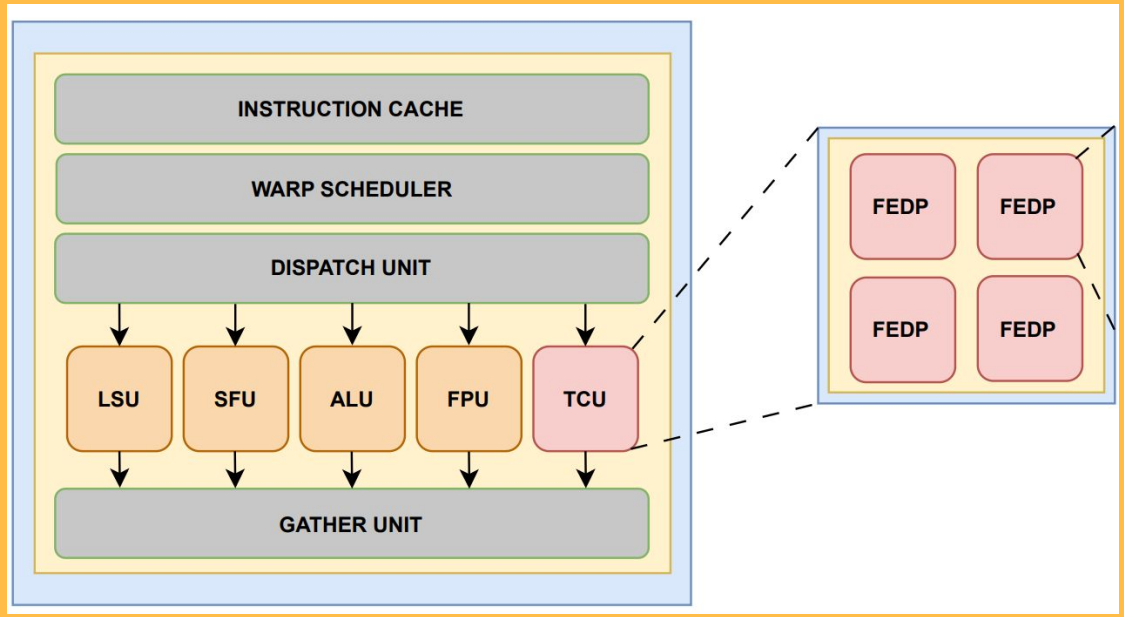


MOTIVATION FOR TENSOR CORES

- SM ALU/FPU only,
  - MUL R0, A0, B0
  - MUL R1, A1, B1
  - MUL R2, A2, B2
  - MUL R3, A3, B3
  - ADD R4, R0, R1
  - ADD R5, R2, R3
  - ADD R6, R4, R5
- TCU extended SM
  - FEDP_MMA R6, A0, B0, A1, B1, A2, B2, A3, B3
- Reduced Register Pressure: (R0, R1, R2, R3, R4, R5)

PLACEMENT OF TENSOR CORES IN A GPGPU SUB-CORE (SM)





https://modal.com/gpu-glossary/device-hardware/tensor-core

Systolic Array based MatMul

+ Low Memory BW, efficient for Large Matrices (256x256) → Training (TPU)
- Rigid uArch → Poor resource utilization/occupancy when operating on small, irregular or sparse matrices; needs (2N - 1) cycles
- Isolated Accelerator-like approach fits into Dataflow Processors

- Multiply two 3x3 matrices (inputs)
  - Keep the final result in PE accumulators

$$\begin{bmatrix} c_{00} & c_{01} & c_{02} \\ c_{10} & c_{11} & c_{12} \\ c_{20} & c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix} \times \begin{bmatrix} b_{00} & b_{01} & b_{02} \\ b_{10} & b_{11} & b_{12} \\ b_{20} & b_{21} & b_{22} \end{bmatrix}$$

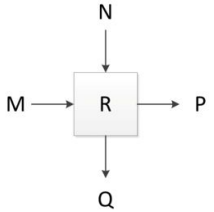
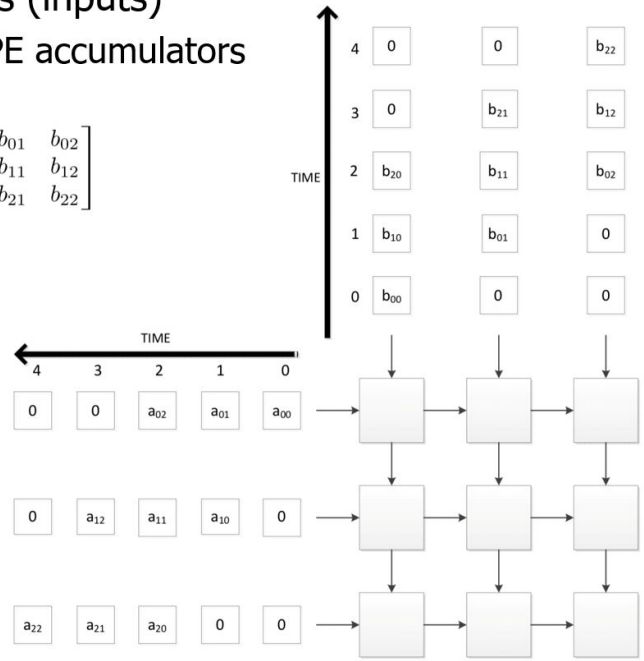


Figure 1: A systolic array processing element

P = M

Q = N

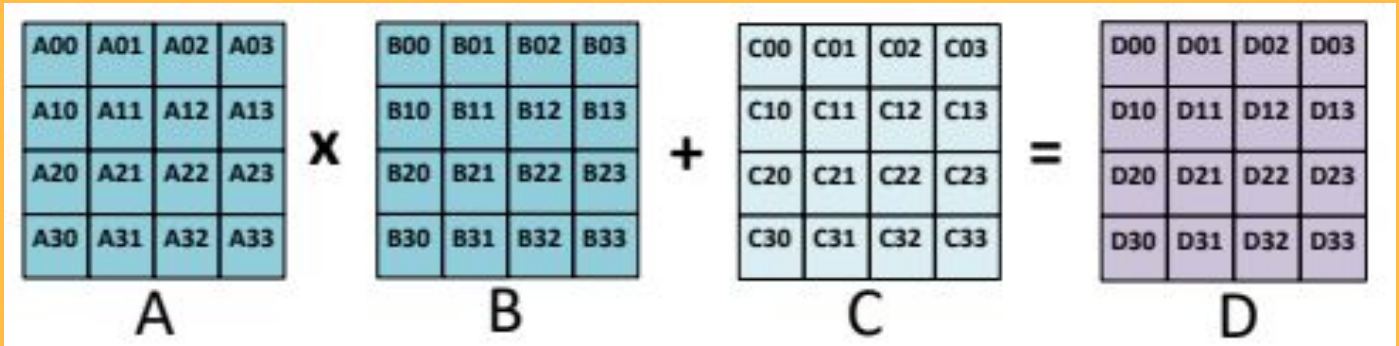
R = R + M*N



Credits: Prof. Onur Mutlu

Fused-Efficient Dot Product (FEDP) Array based MatMul

- Fused Efficient Dot Product (FEDP) Arrays:
  - + Efficient for Small as well as Large Matrices → Inference/Training
  - + GPU memory BW is bound to warp size 16/32T anyway
  - + Large tiles are simply time/space multiplexed
  - + Fixed 4-cycle Latency (Independent of Tile Size)
  - + Flexible support for mixed-precision, irregular and sparse workloads
  - + Tightly-Coupled Functional Unit-like approach fits in to GPGPU Arch



$$Y_i = A_{i+0}B_{i+0} + A_{i+1}B_{i+1} + .. + A_{i+n}B_{i+n} + C_i$$

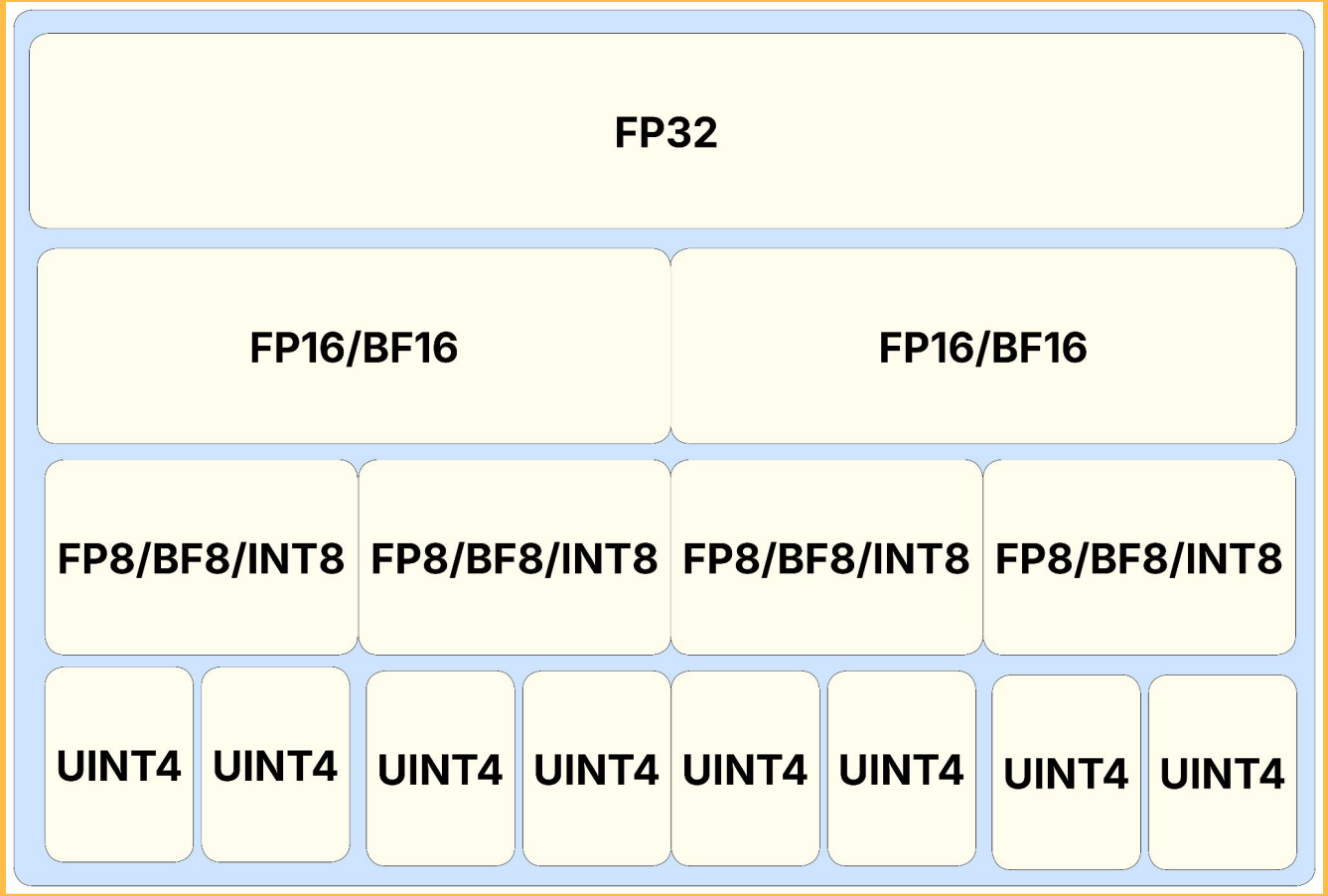
WMMA TILE/CORE SIZE CALCULATION

Per-warp scheduling: ***mmadd rd, rs1, rs2, rs3***

- Register operands spawn the whole warp
- Max register capacity = 32 x 32 bit = 1024 bit
- You can now store a destination tile of 8x4 floats
- Possible tile configurations
  - f32 → f32: M=8, N=4, K=4
  - f16 → f32: M=8, N=4, K=8
  - f16 → f16: M=8, N=8, K=8
  - int8 → i32: M=8, N=4, K=16
  - int4 → i32: M=8, N=4, K=32

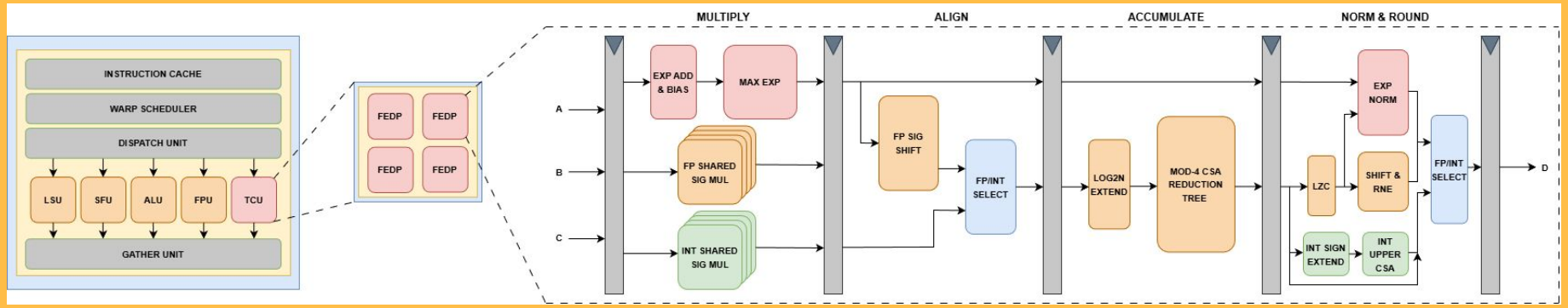
Credits: Prof. Blaise Tine

REGISTER OPERAND PACKING SCHEME



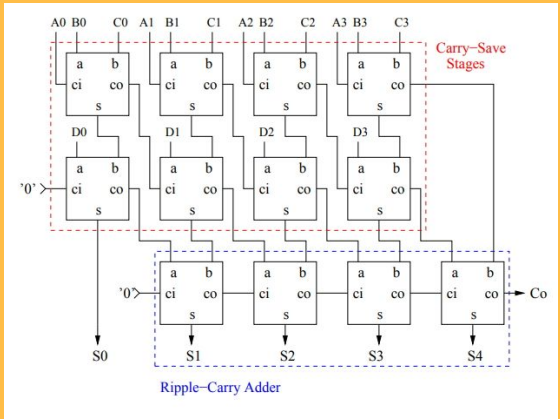
FEDP MICROARCHITECTURE CHALLENGES

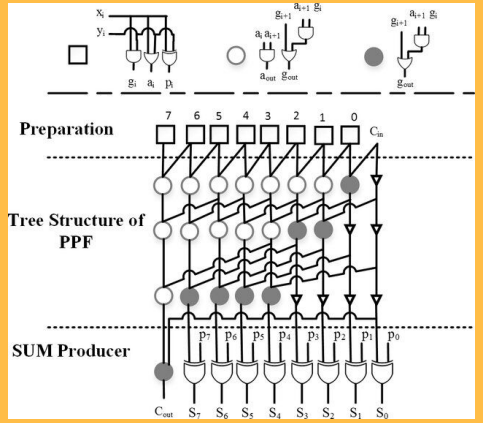
- Low latency (NVIDIA Volta FEDP had a 4 stage pipeline)
- Mixed-Precision FP support for efficient DL workload specific operations
- Fuse Integer Pipeline within Floating-Point Datapath maximizing resource reuse with minimal overhead
- Primary Goal? Meet 300MHz operational freq timing req
- Open-Source Baseline: BSG modified SystemVerilog Berkeley HardFloat (RocketChip, Gemmini, Virgo, BSG Manycore)



FEDP MICROARCHITECTURE: LIBRARY MODULES

- Floating-point FEDP datapath requires several multi-operand additions:
  - Carry Save Adder (CSA) - **(O(log2(N) + O(W))**
- Fast final 2-operand adder:
  - Kogge-Stone Adder (KSA) - Parallel prefix tree structure
- Fast NxN-bit multiplier:
  - Wallace Tree Multiplier (WTMUL) - Fast partial product reduction
- Additional details in Backup Slides

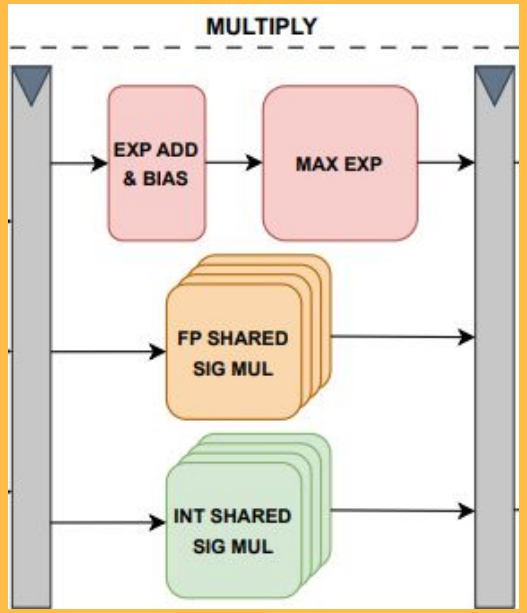




				a_3	a_2	a_1	a_0
			x	b_3	b_2	b_1	b_0
p_{70}	p_{60}	p_{50}	p_{40}	p_{30}	p_{20}	p_{10}	p_{00}
p_{61}	p_{51}	p_{41}	p_{31}	p_{21}	p_{11}	p_{01}	x
p_{52}	p_{42}	p_{32}	p_{22}	p_{12}	p_{02}	x	x
p_{43}	p_{33}	p_{23}	p_{13}	p_{03}	x	x	x
z_7	z_6	z_5	z_4	z_3	z_2	z_1	z_0

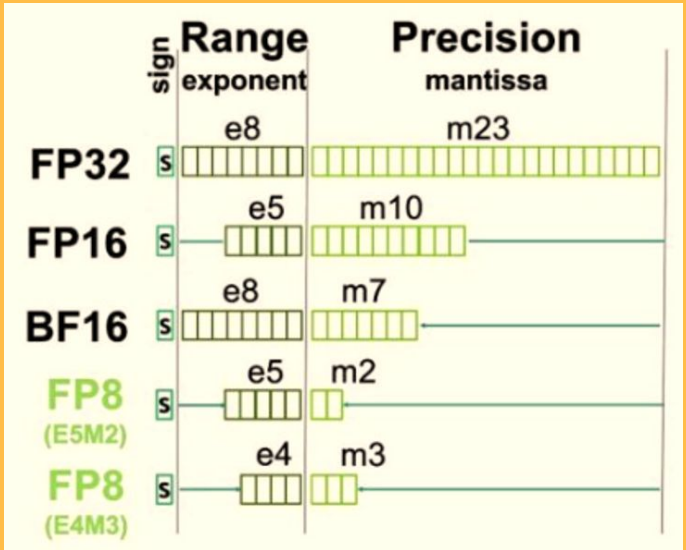
FEDP Microarchitecture: Shared Low-Precision Multiplication

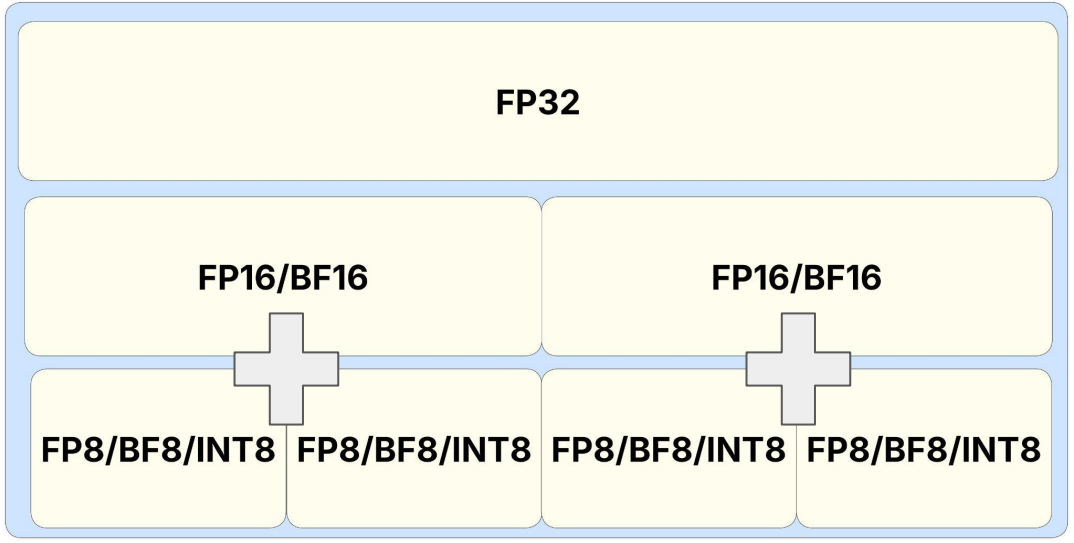
- Dedicated Wallace Tree multipliers approach requires:
  - One 11x11 (fp16), One 8x8 (bf16) Two 8x8(int8), Six 4x4 (fp8, uint4) and Two 3x3 (bf8)
  - → Thirteen variable-width multipliers
- Instead we share “class-wise” multipliers to save area based on format,
  - One 11x11 (fp16/bf16) – (extensible: tf32)
  - Two 8x8 (fp8/bf8/int8) – (extensible: uint8)
  - Four 4x4 (uint4) – (extensible: int4, fp4)
  - → Only **seven** variable-width multipliers
- Overhead incurred by radix-4 booth recoding isn’t worth it at our target bit-widths (4-11) although no. of partial products are halved



FEDP Microarchitecture: Shared Low-Precision Multiplication

- Formats that pack > 2 operands per register need additional additions to maintain the same width and format compatibility later in the pipeline





- All multiplications converge in an E8M25 intermediate representation for following stages in pipeline

FEDP Microarchitecture: Exponent Add & Bias

- Each FP format has a Exponent bias based on Exponent Bit-Width
- For eg,
  - FP32 Exp → 8-bits → Bias = (2^(8-1) -1) = 127
  - FP16 Exp → 5-bits → Bias = (2^(5-1) -1) = 15
- FP Multiplication result calculation requires input exponents addition
- Mixed-Precision operation requires bias adjustment

$$X \times Y = (X_s \times Y_s) \times 2^{X_E + Y_E}$$

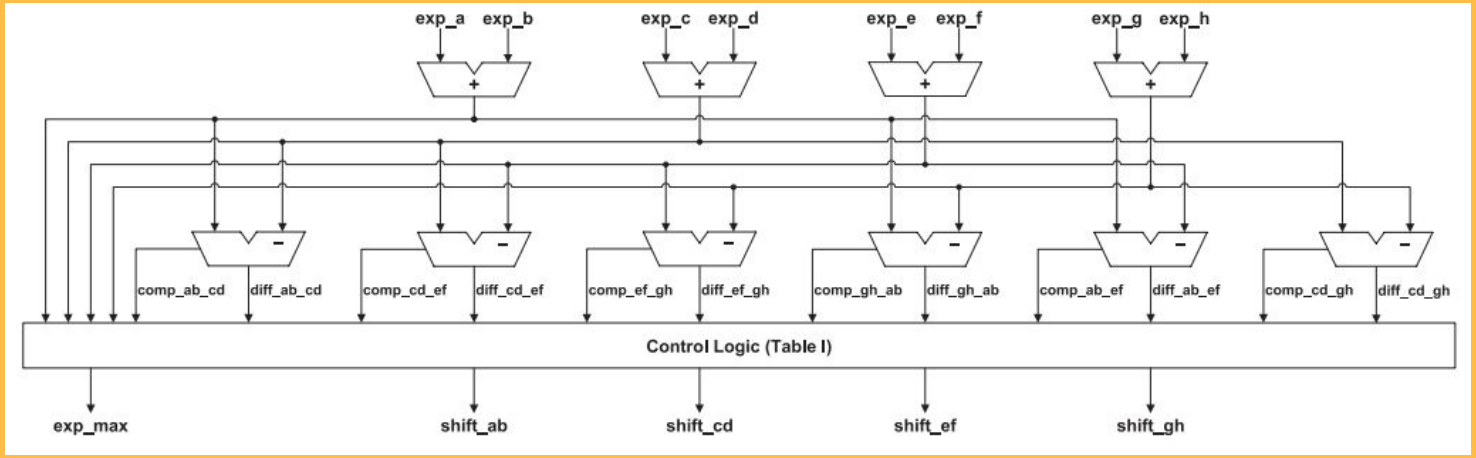
$$EXP_{FP32} = EXP_A + EXP_B + BIAS_{FP32} - (2 \times BIAS_{FP16}) + 1$$

FEDP Microarchitecture: Maximum Exponent Identification

- Compute all N×N pairwise exponent sign and difference matrices
  O(1) depth vs O(log2(N)) tree-comparator structure.

$$\begin{bmatrix} 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

V2 > v1 > v3 > v4



- Don’t need to re-calculate shift amounts, simply invert subtractor results in consideration

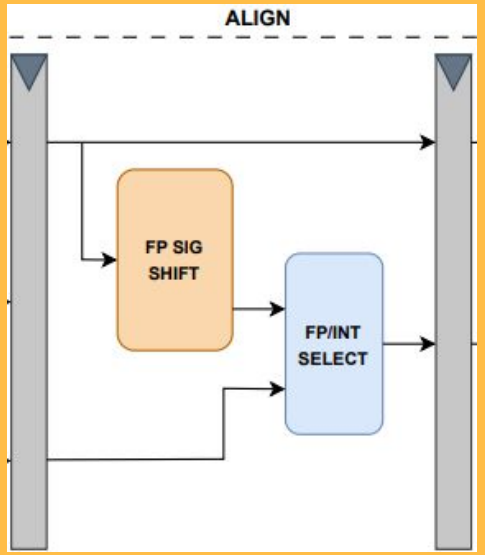
https://ieeexplore.ieee.org/document/7416176

FEDP Microarchitecture: Significand Alignment

- E8M25 raw significands require shifting before accumulation

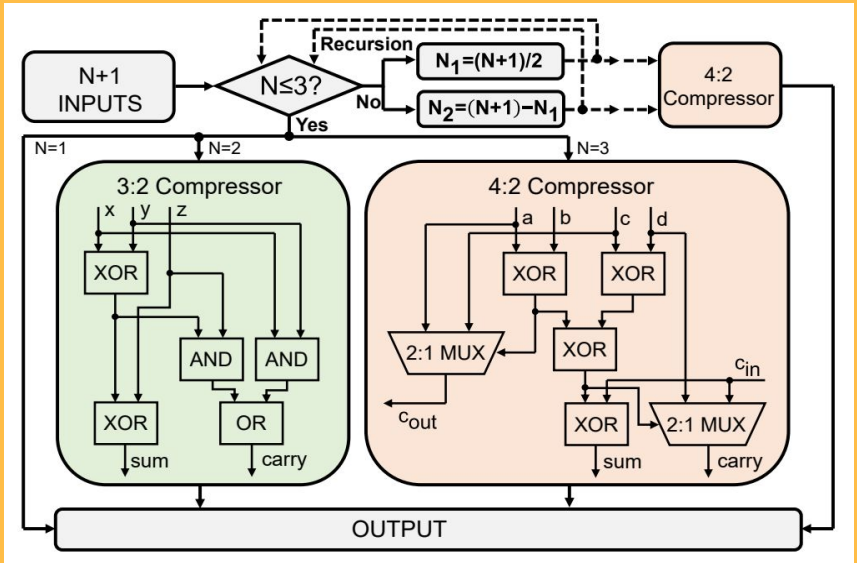
$$X + Y = (X_S \times 2^{X_E - Y_E} + Y_S) \times 2^{Y_E}$$

- Shift amounts were already calculated in the previous stage's maximum exponent logic



FEDP Microarchitecture: High-Precision Accumulation

- N-operand (25+log2(N))-bit CSA accumulates aligned significand products
- Dealing with signed-values requires an additional log2(N) bit sign-extension before feeding into CSA
- Addend “C” accumulation is integrated into this large accumulation → smaller rounding error/critical path

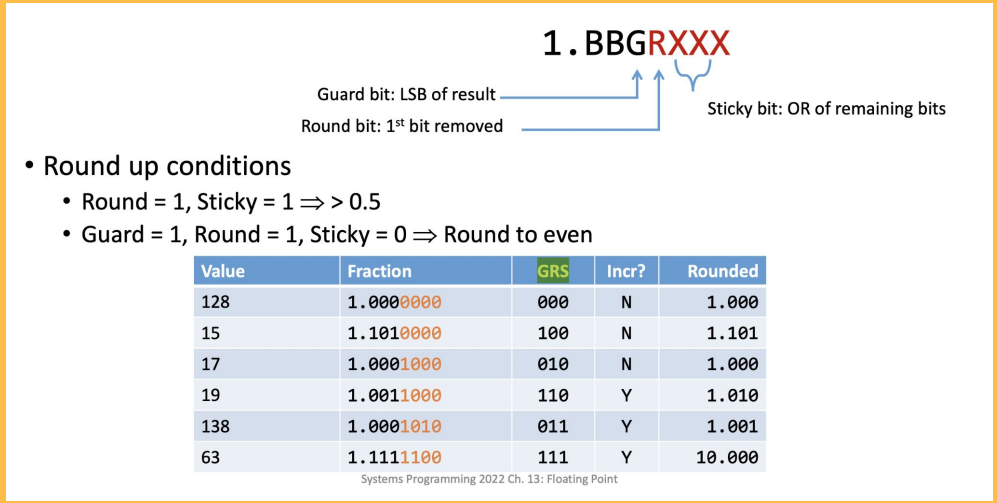


The CSA utilizes a recursive 4:2 compressor structure with 3:2 fallback to deal with odd number of operands cases

https://ieeexplore.ieee.org/document/10182007

FEDP Microarchitecture: Rounding & Normalization

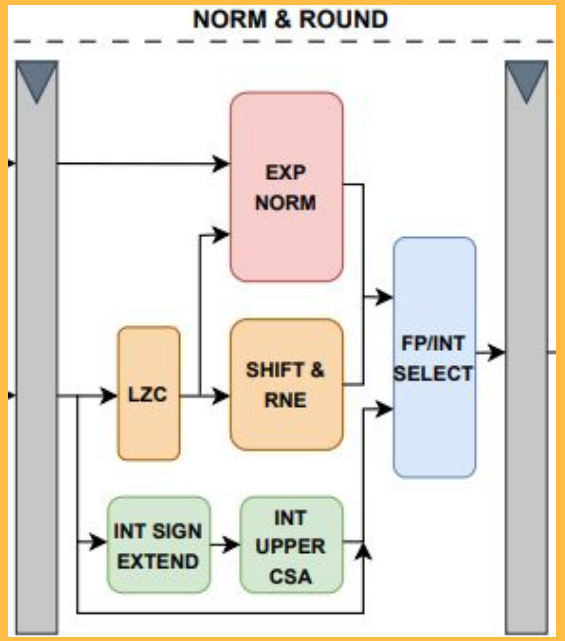
- Standard Leading Zero Counter (LZC) normalization and Round-to-nearest-even (RNE) rounding are performed.



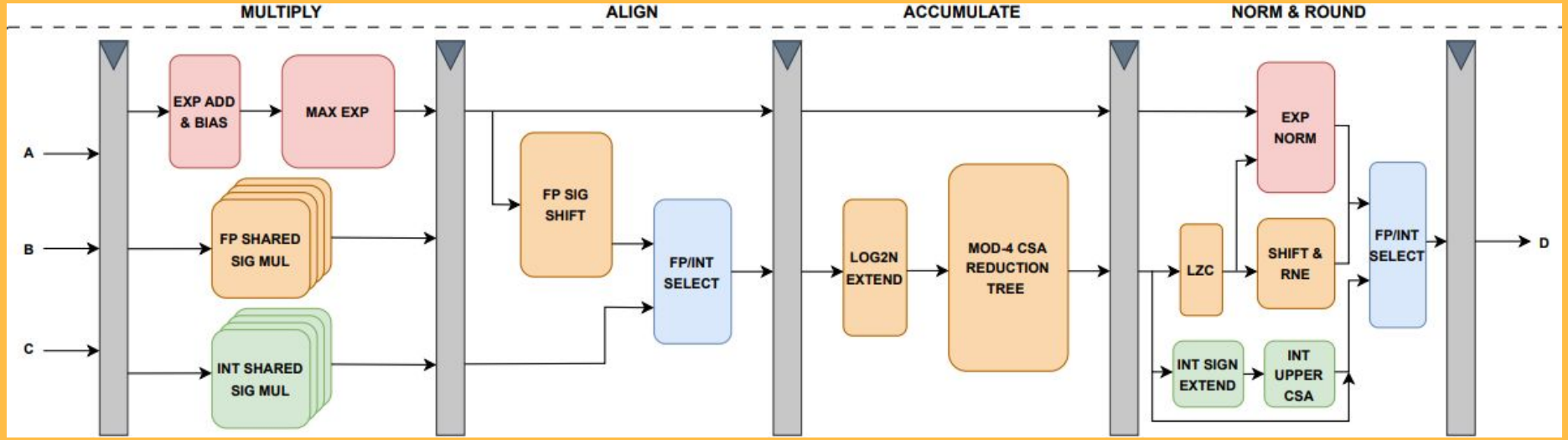
round_up = guard_bit & (round_bit | sticky_bit | lsb)

FEDP Microarchitecture: Fusing the Integer Pipeline

- Many components required for Integer dot product operations are already found in the existing FP datapath → Fusing both pipelines eliminates arbiter overhead and separate scheduling units
- Products zero/sign-extended to 25-bits to match accumulator requirements and enable reuse of intermediate registers
- 32-bit addend C is partitioned into lower 25 bits (processed in FP accumulation) and upper 7 bits (propagated separately)
- Final 32-bit integer output is assembled by concatenating 25-bit accumulation with upper 7 bits computed (CSA) in parallel to FP normalization/rounding stage



FEDP Microarchitecture: Putting everything together

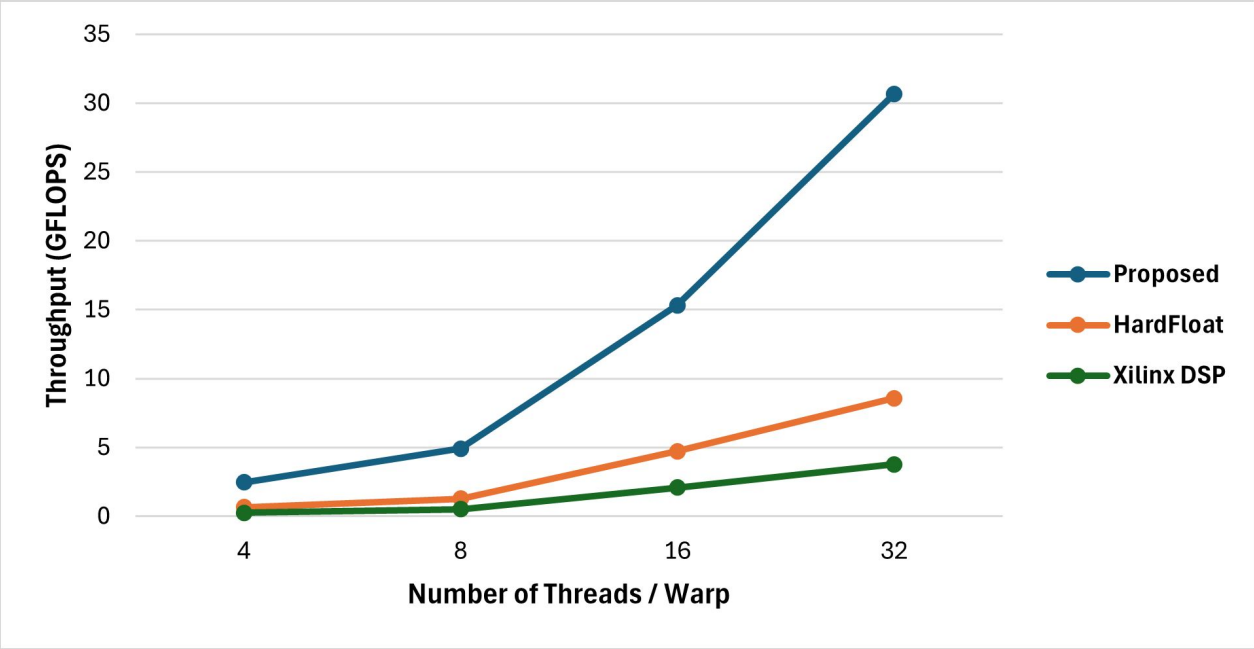


FEDP MICROARCHITECTURE: EVALUATION ENVIRONMENT

- RTL implementations of these modules can be found at https://github.com/vortexgpgpu/vortex/tree/bug_fixes/hw/rtl/tcu/drl
- Library modules can be found at https://github.com/vortexgpgpu/vortex/tree/bug_fixes/hw/rtl/libs
- Bash commands for running tests can be found at https://github.com/vortexgpgpu/vortex/tree/bug_fixes/ci/regression.sh.in

The following evaluations of our FEDP design are made against equivalent Berkeley HardFloat and Xilinx DSP IP based discrete implementations, targeting 300MHz clock frequency on the AMD Xilinx Alveo U55C FPGA across a range of Threads/Warp Configurations (4, 8, 16, 32)

FEDP Microarchitecture: Performance Scaling



Proposed FEDP Throughput = (**2.453–30.662) GFLOPS!**

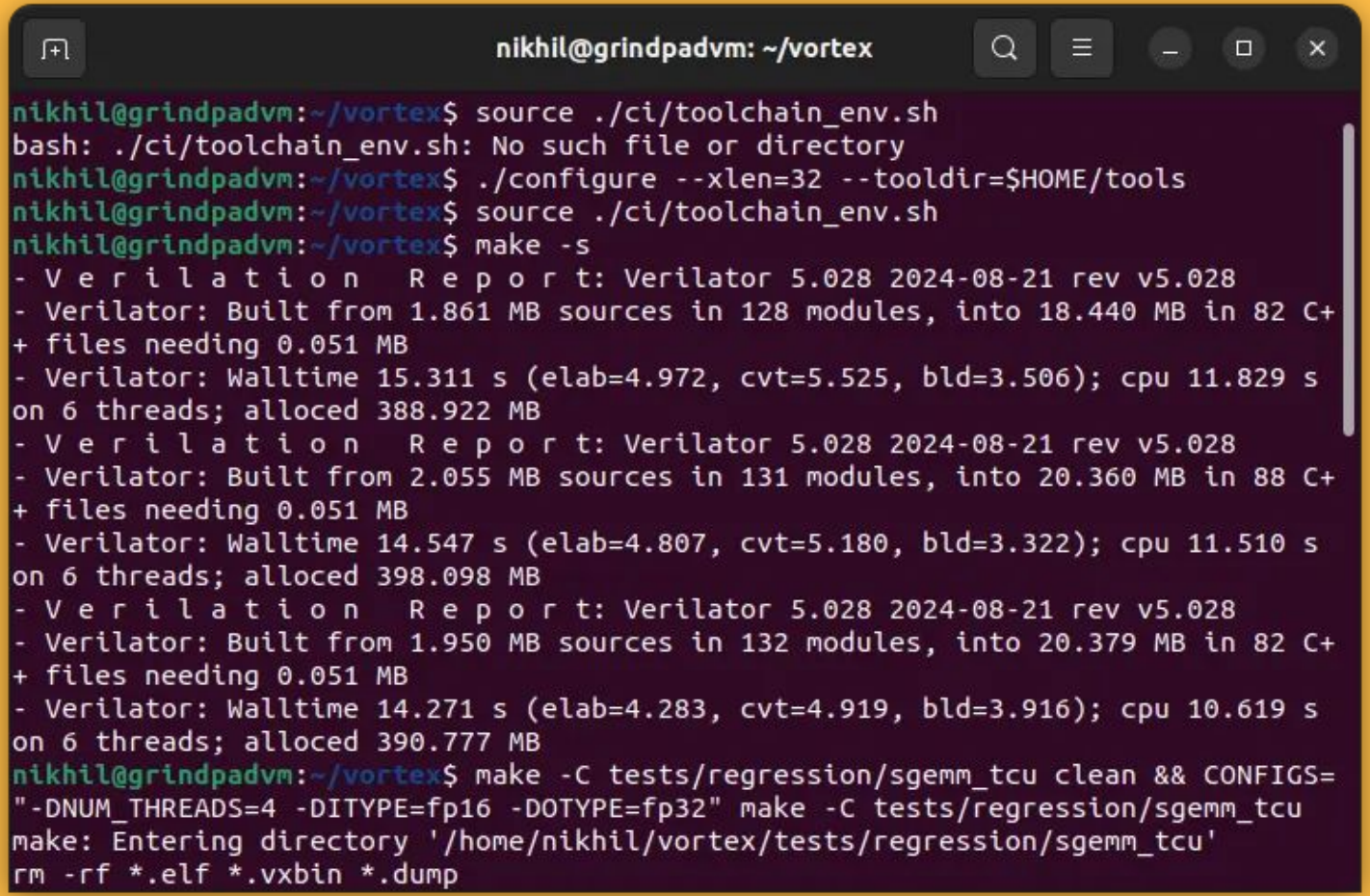
~ 3.7x Berkeley HardFloat

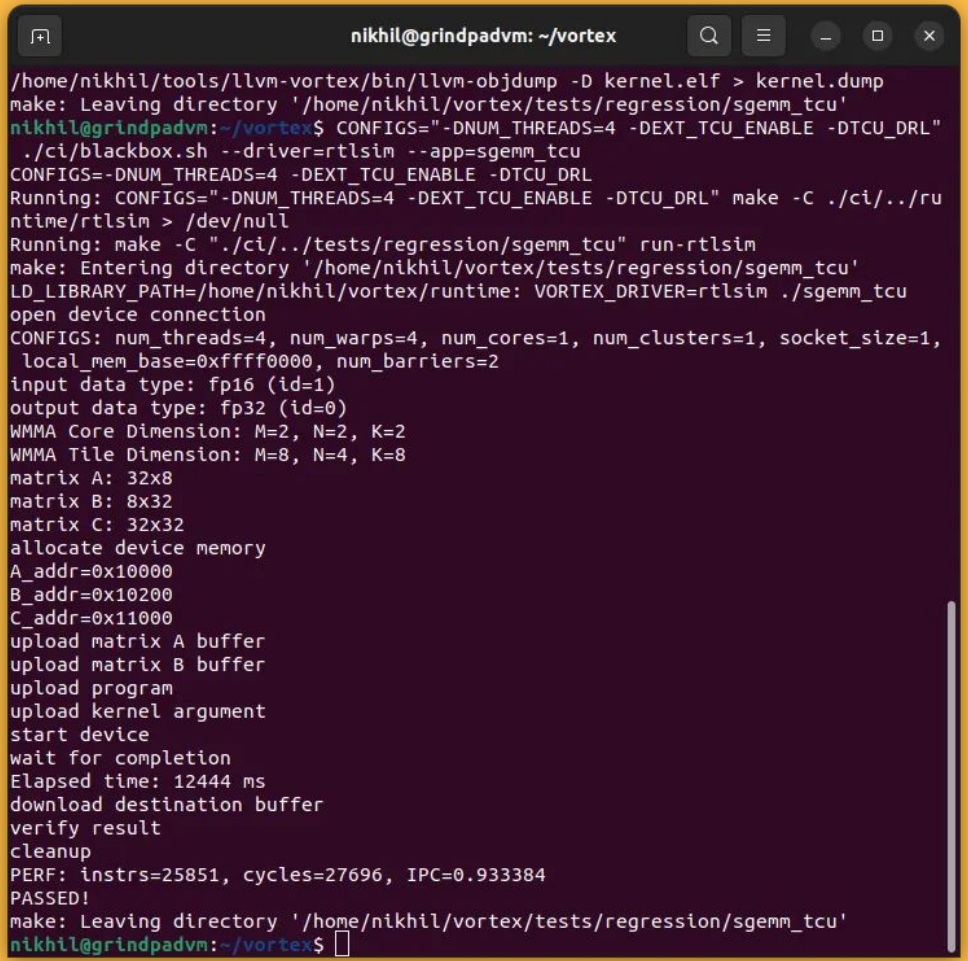
FEDP Microarchitecture: Area Costs

	Backend	N = 4	N = 8	N = 16	N = 32
LUTs	Xilinx DSP	6216	12414	49236	98581
	HardFloat	18400	37002	144001	291207
	Proposed	**10945**	**21899**	**95336**	**188077**
FFs	Xilinx DSP	9107	18063	70738	141314
	HardFloat	6163	12153	46850	93190
	Proposed	**2364**	**4624**	**14967**	**29769**
DSPs	Xilinx DSP	64	128	512	1024
	HardFloat	16	32	128	256
	Proposed	**0**	**0**	**0**	**0**

Proposed <60% Berkeley HardFloat

SGEMM TCU DEMO


nikhil@grindpadvm: ~/vortex
nikhil@grindpadvm:~/vortex$ source ./ci/toolchain_env.sh
bash: ./ci/toolchain_env.sh: No such file or directory
nikhil@grindpadvm:~/vortex$ ./configure --xlen=32 --tooldir=$HOME/tools
nikhil@grindpadvm:~/vortex$ source ./ci/toolchain_env.sh
nikhil@grindpadvm:~/vortex$ make -s
- V e r i l a t i o n   R e p o r t: Verilator 5.028 2024-08-21 rev v5.028
- Verilator: Built from 1.861 MB sources in 128 modules, into 18.440 MB in 82 C+
+ files needing 0.051 MB
- Verilator: Walltime 15.311 s (elab=4.972, cvt=5.525, bld=3.506); cpu 11.829 s
on 6 threads; alloced 388.922 MB
- V e r i l a t i o n   R e p o r t: Verilator 5.028 2024-08-21 rev v5.028
- Verilator: Built from 2.055 MB sources in 131 modules, into 20.360 MB in 88 C+
+ files needing 0.051 MB
- Verilator: Walltime 14.547 s (elab=4.807, cvt=5.180, bld=3.322); cpu 11.510 s
on 6 threads; alloced 398.098 MB
- V e r i l a t i o n   R e p o r t: Verilator 5.028 2024-08-21 rev v5.028
- Verilator: Built from 1.950 MB sources in 132 modules, into 20.379 MB in 82 C+
+ files needing 0.051 MB
- Verilator: Walltime 14.271 s (elab=4.283, cvt=4.919, bld=3.916); cpu 10.619 s
on 6 threads; alloced 390.777 MB
nikhil@grindpadvm:~/vortex$ make -C tests/regression/sgemm_tcu clean && CONFIGS=
"-DNUM_THREADS=4 -DITYPE=fp16 -DOTYPE=fp32" make -C tests/regression/sgemm_tcu
make: Entering directory '/home/nikhil/vortex/tests/regression/sgemm_tcu'
rm -rf *.elf *.vxbin *.dump


nikhil@grindpadvm: ~/vortex
/home/nikhil/tools/llvm-vortex/bin/llvm-objdump -D kernel.elf > kernel.dump
make: Leaving directory '/home/nikhil/vortex/tests/regression/sgemm_tcu'
nikhil@grindpadvm:~/vortex$ CONFIGS="-DNUM_THREADS=4 -DEXT_TCU_ENABLE -DTCU_DRL"
./ci/blackbox.sh --driver=rtlsim --app=sgemm_tcu
CONFIGS=-DNUM_THREADS=4 -DEXT_TCU_ENABLE -DTCU_DRL
Running: CONFIGS="-DNUM_THREADS=4 -DEXT_TCU_ENABLE -DTCU_DRL" make -C ./ci/../runtime/rtlsim > /dev/null
Running: make -C "./ci/../tests/regression/sgemm_tcu" run-rtlsim
make: Entering directory '/home/nikhil/vortex/tests/regression/sgemm_tcu'
LD_LIBRARY_PATH=/home/nikhil/vortex/runtime: VORTEX_DRIVER=rtlsim ./sgemm_tcu
open device connection
CONFIGS: num_threads=4, num_warps=4, num_cores=1, num_clusters=1, socket_size=1, local_mem_base=0xffff0000, num_barriers=2
input data type: fp16 (id=1)
output data type: fp32 (id=0)
WMMA Core Dimension: M=2, N=2, K=2
WMMA Tile Dimension: M=8, N=4, K=8
matrix A: 32x8
matrix B: 8x32
matrix C: 32x32
allocate device memory
A_addr=0x10000
B_addr=0x10200
C_addr=0x11000
upload matrix A buffer
upload matrix B buffer
upload program
upload kernel argument
start device
wait for completion
Elapsed time: 12444 ms
download destination buffer
verify result
cleanup
PERF: instrs=25851, cycles=27696, IPC=0.933384
PASSED!
make: Leaving directory '/home/nikhil/vortex/tests/regression/sgemm_tcu'
nikhil@grindpadvm:~/vortex$

FUTURE WORK IN PROGRESS/CONSIDERATION

- Split 4:2 CSA structure into parallel MOD-4 groups for reducing large Ns
  - Eg. 8 thread config → needs 9 operands to be summed
- No difference in latency of dense vs sparse FEDP, but power consumption can be optimized by enable gating modules/clock gating intermediate registers
- Add E8M0 registers for per-operand scaling, and some additional scaling multipliers for OCP Microscaling MXFP8 formats dot product support

The dot product of two MX-compliant format vectors A: $\left\{X^{(A)}, \left[P_i^{(A)}\right]_{i=1}^{k}\right\}$ and B: $\left\{X^{(B)}, \left[P_i^{(B)}\right]_{i=1}^{k}\right\}$ of length k is a scalar number C. The following semantics *must* be minimally supported:

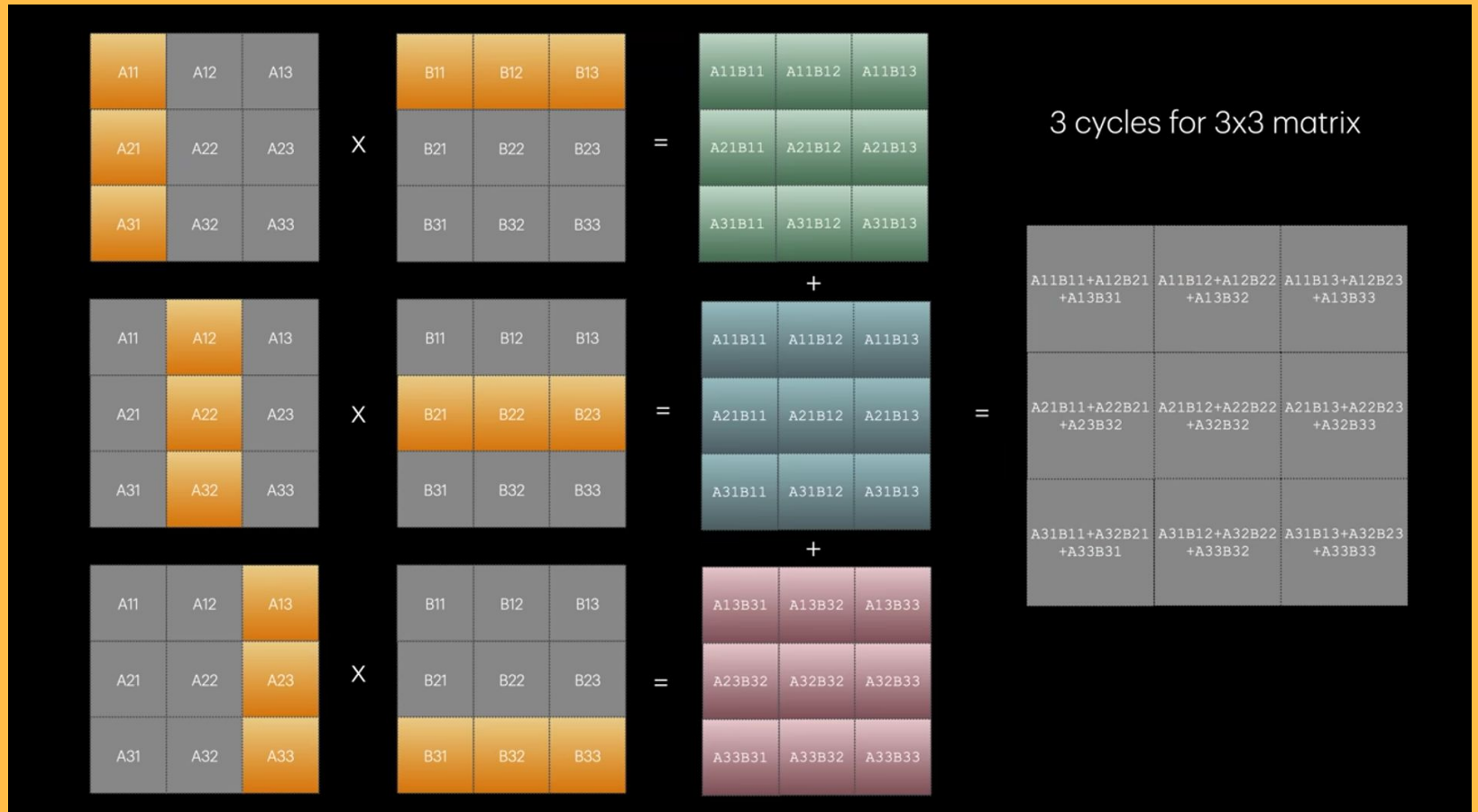
$$C = Dot(A,B) = X^{(A)}X^{(B)}\sum_{i=1}^{k}\left(P_i^{(A)} \times P_i^{(B)}\right)$$

Where:

- $X^{(A)}$, $X^{(B)}$ are the block scales of vectors A and B respectively.

BACKUP/ADDITIONAL SLIDES

Outer Product Matmul



Inner vs Outer Product

- An outer product based uarch would enable lesser MIO BW (only requires reading each row/colum once)
- However it requires storing full tile size intermediate results before accumulation → very expensive
- Adjacent Rows/Columns can be accessed relatively easily from shared memory → inner product based uarch should have the same throughput

OCP Microscaling MX Format Support

- Only select Nvidia Blackwell and AMD CDNA4 archs support MX formats atm
- Since, we already have FP8 support, we only need to add E8M0 registers for per-operand scaling, and some additional scaling multipliers

The dot product of two MX-compliant format vectors A: $\left\{X^{(A)}, \left[P_i^{(A)}\right]_{i=1}^{k}\right\}$ and B: $\left\{X^{(B)}, \left[P_i^{(B)}\right]_{i=1}^{k}\right\}$ of length k is a scalar number C. The following semantics *must* be minimally supported:

$$C = Dot(A,B) = X^{(A)}X^{(B)}\sum_{i=1}^{k}\left(P_i^{(A)} \times P_i^{(B)}\right)$$

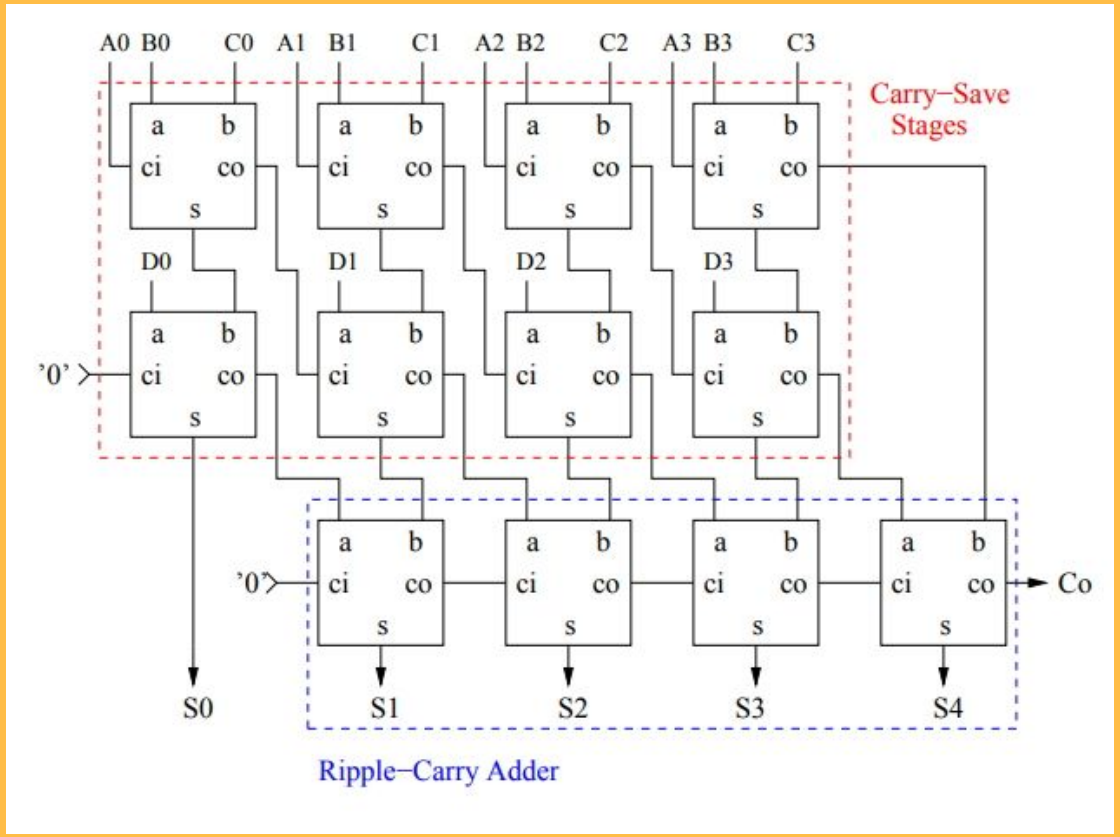
Where:

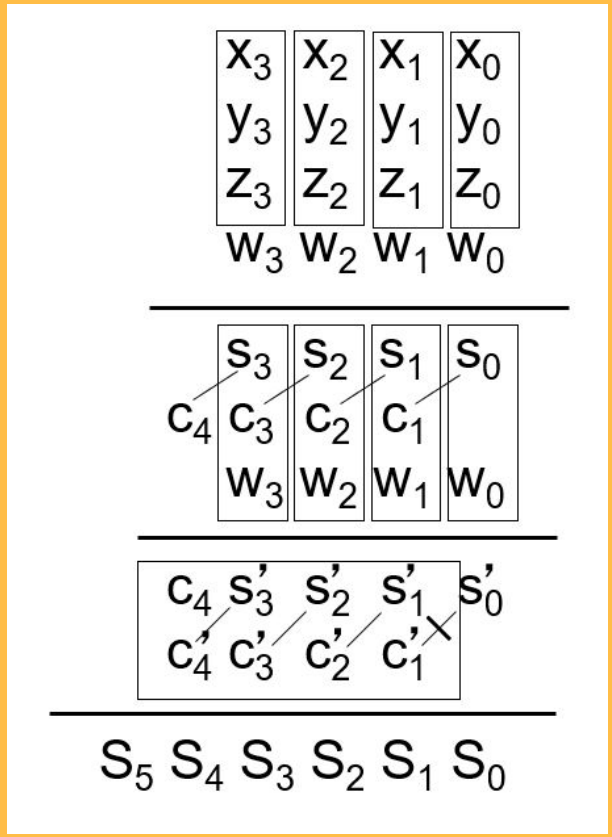
- $X^{(A)}$, $X^{(B)}$ are the block scales of vectors A and B respectively.

FEDP MICROARCHITECTURE LIBRARY MODULES

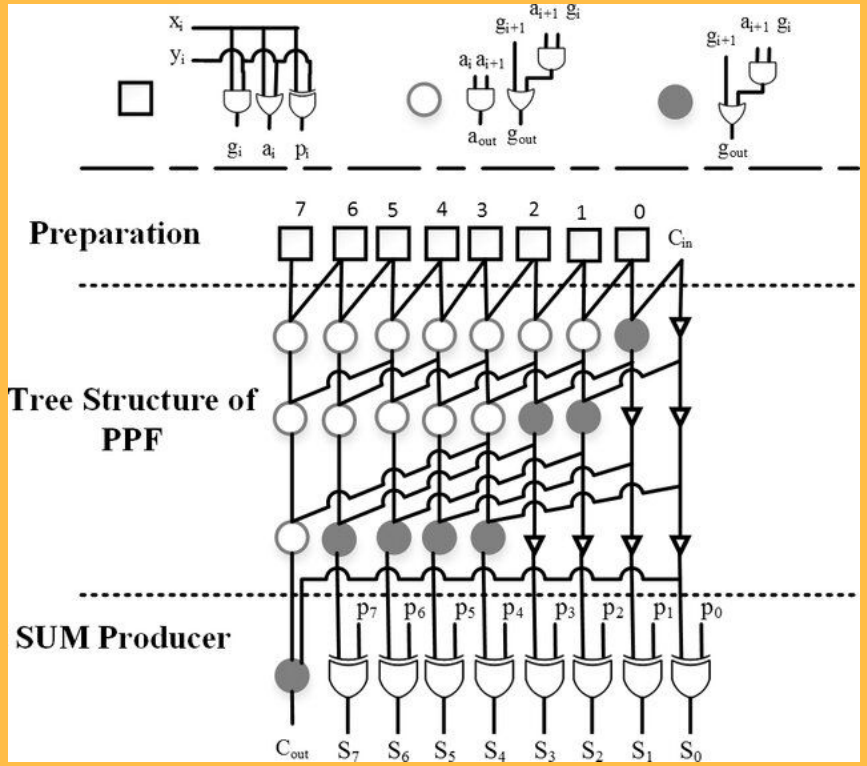
- Floating-point FEDP datapath requires several multi-operand additions:
  - Summing multiple partial products from multiplier AND array
  - Summing multiple exponents for result exponent add and bias
  - Summing multiple sub-product
- Assuming we have N operands of W bits each:
  - Traditional Reduction Tree Adders have (O(log2(N)) x O(W)) complexity
  - Carry Save Adders have approximately **(O(log2(N) + O(W))** complexity
- Operands in a CSA reduce vertically in parallel without prior carry dependencies and only require one complete W-bit addition at the end

FEDP Microarchitecture Library Modules: Carry Save Adder (CSA)





FEDP Microarchitecture Library Modules: Kogge-Stone Adder (KSA)



Gi=Ai & Bi

Pi = Ai ^ Bi

$$
\begin{aligned}
c_1 &= G_0 + c_0 P_0, \\
c_2 &= G_1 + G_0 P_1 + c_0 P_0 P_1, \\
c_3 &= G_2 + G_1 P_2 + G_0 P_1 P_2 + c_0 P_0 P_1 P_2, \\
c_4 &= G_3 + G_2 P_3 + G_1 P_2 P_3 + G_0 P_1 P_2 P_3 + c_0 P_0 P_1 P_2 P_3
\end{aligned}
$$

FEDP Microarchitecture Library Modules: Wallace-Tree Multiplier

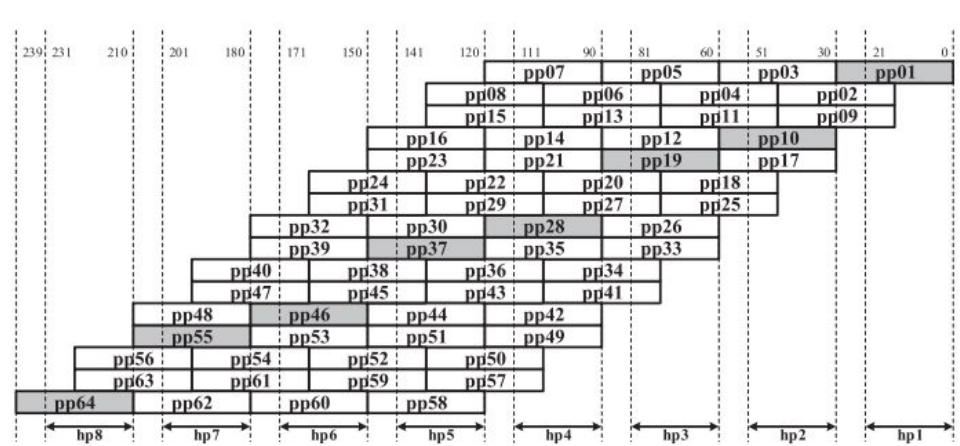
- Wallace-Tree Multipliers reduce the multiple partial products from the multipliers AND array effectively using a CSA

				a_3	a_2	a_1	a_0
			x	b_3	b_2	b_1	b_0
p_{70}	p_{60}	p_{50}	p_{40}	p_{30}	p_{20}	p_{10}	p_{00}
p_{61}	p_{51}	p_{41}	p_{31}	p_{21}	p_{11}	p_{01}	x
p_{52}	p_{42}	p_{32}	p_{22}	p_{12}	p_{02}	x	x
p_{43}	p_{33}	p_{23}	p_{13}	p_{03}	x	x	x
z_7	z_6	z_5	z_4	z_3	z_2	z_1	z_0

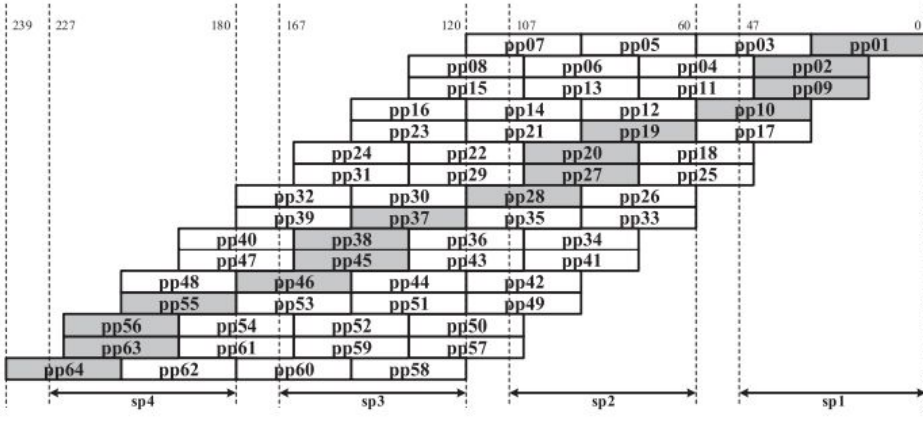
- Radix-4 booth recoding overhead outweighs the benefit of halving the number of partial products at our target 4-11 bit widths.

Shared Multiplier from Efficient FMA

- Unified mini-multiplier partial product grid for mantissa multiplication. Activate partial products as required based on format



(a) Half-precision mode



(b) Single-precision mode

KEY TAKEAWAYS FROM Efficient FMA

- Example executing fp32 24-bit mantissa multiplication using 15x15 bit multipliers
  - The four 15×15 multiplications produce:
    - pp0 = A_sub0 × B_sub0 (positions 0-29)
    - pp1 = A_sub0 × B_sub1 (positions 15-44)
    - pp2 = A_sub1 × B_sub0 (positions 15-44)
    - pp3 = A_sub1 × B_sub1 (positions 30-59)
  - Accumulate PPs using 4:2 CSA
- Advantages to this approach:
  - One large multiplier for all formats
  - Efficient resource utilization
  - Regularity in design → better synthesis
  - Parallel LZ(A)C subnormal to normal number during format transition

KEY TAKEAWAYS FROM Efficient FMA

- Drawbacks to this approach:
  - A lot of subword boundary management
    - Sign Extension: Extending sum vectors with 1's to the MSB position of the larger product (like in VX_tcu_drl_acc currently)
    - Carry Suppression: Discarding any carry propagation through bit positions at mid-point zero-padded junctions (Eg. [59:48] position wrt 48-bit fp32 significand mul)
  - Still requires zero-padding for smaller formats
    - Alternatively, if a smaller bit size multiplier is chosen to avoid zero-padding, a larger dimension partial product grid is required and larger supported formats will require a lot of csa accumulation stages