ARITH 2025

Double-Word Decomposition in a Combined FP16, BF16 and FP32 Dot Product Add Operator

Orégane Desrentes

Benoît Dupont de Dinechin

Florent de Dinechin

Context

A short and biased history of floating point units

- ... with a focus on adders and multipliers

A short and biased history of floating point units

- ... with a focus on adders and multipliers $r = \circ(x + y)$ $r = \circ(x \times y)$
- In 1985, IEEE-754 standardizes formats and rounding $\circ(\ldots)$

A short and biased history of floating point units

- ... with a focus on adders and multipliers $r = \circ(x + y)$ $r = \circ(x \times y)$
- In 1985, IEEE-754 standardizes formats and rounding $\circ(\dots)$
- In the 90s, FMA (fused multiply add): $r = \circ(x \times y + z)$
  - two operations in one instruction: *faster*
  - one single rounding: *more accurate*

A short and biased history of floating point units

- ... with a focus on adders and multipliers $r = \circ(x + y)$ $r = \circ(x \times y)$
- In 1985, IEEE-754 standardizes formats and rounding $\circ(\dots)$
- In the 90s, FMA (fused multiply add): $r = \circ(x \times y + z)$
- These days: $r = \circ\left(z + \sum_{i=0}^{N-1} x_i \times y_i\right)$ (our ARITH 2023 paper)
  - for some fixed N (typically $N =$8 to 32, and growing)
  - with **mixed-precision** for accurate chaining of such operations: x_i and y_i in a small format, z and r in a wider format (typically 8/16 bits, or 16/32 bits)

A short and biased history of floating point units

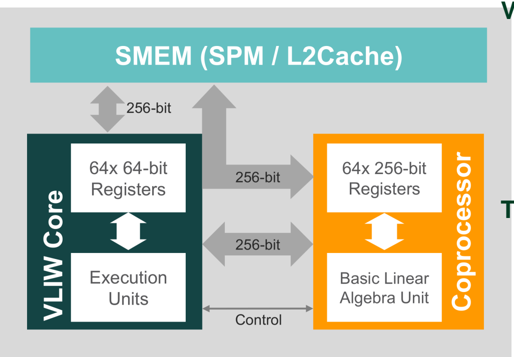
- ... with a focus on adders and multipliers $r = \circ(x + y)$ $r = \circ(x \times y)$
- In 1985, IEEE-754 standardizes formats and rounding $\circ(\ldots)$
- In the 90s, FMA (fused multiply add): $r = \circ(x \times y + z)$
- These days: $r \approx z + \sum_{i=0}^{N-1} x_i \times y_i$ (GPUs, ML/AI accelerators)
  - for some fixed N (typically $N =$8 to 32, and growing)
  - with **mixed-precision** for accurate chaining of such operations:
    x_i and y_i in a small format, z and r in a wider format (typically 8/16 bits, or 16/32 bits)

A short and biased history of floating point units

- ... with a focus on adders and multipliers $r = \circ(x + y)$ $r = \circ(x \times y)$
- In 1985, IEEE-754 standardizes formats and rounding $\circ(\dots)$
- In the 90s, FMA (fused multiply add): $r = \circ(x \times y + z)$
- These days: $r \approx z + \sum_{i=0}^{N-1} x_i \times y_i$ (GPUs, ML/AI accelerators)
  - for some fixed N (typically N =8 to 32, and growing)
  - with **mixed-precision** for accurate chaining of such operations:
    x_i and y_i in a small format, z and r in a wider format (typically 8/16 bits, or 16/32 bits)
  - presented as a small **matrix-matrix-accumulate (MMA)** unit: $D = C + A \times B$
    (some call it "tensor operator")
    to **balance compute (fast) and data access (slower)**:
    $O(n^3)$ compute for $O(n^2)$ data access, where n is the matrix dimension

Example: MPPA3 V2 Coolidge™ processing element (PE)

A 6-issue 64-bit VLIW core with a tightly-coupled tensor coprocessor



VLIW Core

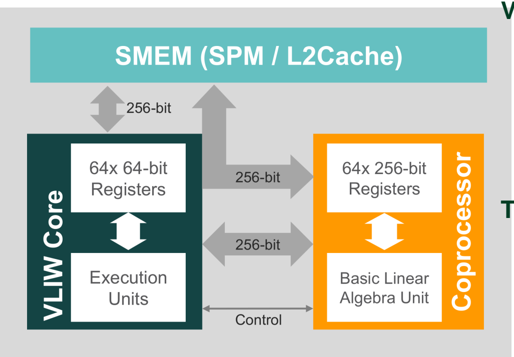
- Scalar: 32-bit and 64-bit INT & FP
- SIMD: 8× 8-bit, 4× 16-bit, 2× 32-bit
- 128-bit and 256-bit SIMD operations in a VLIW bundle
- 256-bit load/store unit with masking

Tensor Coprocessor

- 256-bit load/store unit with masking
- Matrix zip/unzip & transpose
- Blocks of 256-bit registers used as circular buffer or as lookup table

Example: MPPA3 V2 Coolidge™ processing element (PE)

A 6-issue 64-bit VLIW core with a tightly-coupled tensor coprocessor



VLIW Core

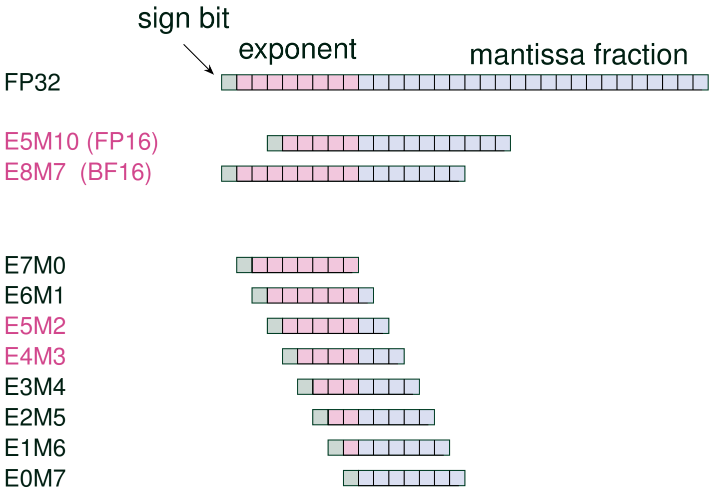
- Scalar: 32-bit and 64-bit INT & FP
- SIMD: 8$\times$ 8-bit, 4$\times$ 16-bit, 2$\times$ 32-bit
- 128-bit and 256-bit SIMD operations in a VLIW bundle
- 256-bit load/store unit with masking

Tensor Coprocessor

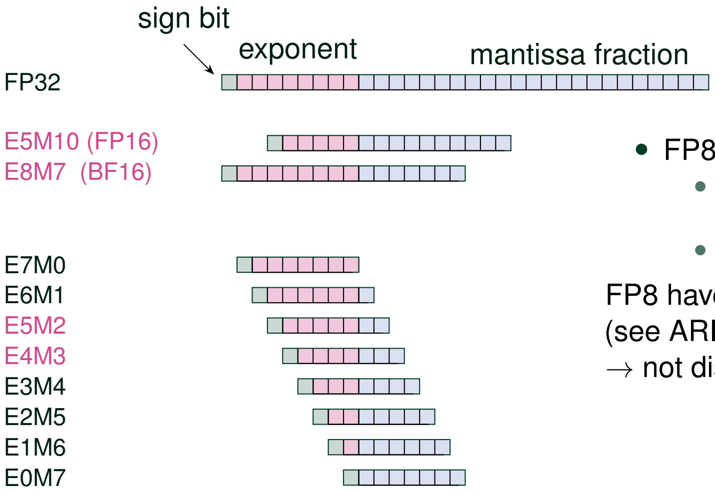
- 256-bit load/store unit with masking
- Matrix zip/unzip & transpose
- Blocks of 256-bit registers used as circular buffer or as lookup table

In short: all the tensor blah blah is really about (clever) data shuffling,
the arithmetic is based on **Mixed-Precision Dot Product and Add** operators.

Arithmetic formats for machine learning



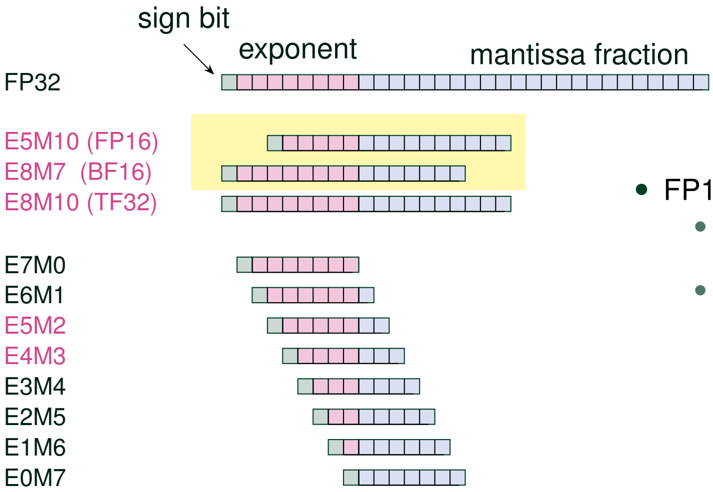
Arithmetic formats for machine learning



- FP8 (aka E5M2 and E4M3)
  - sum-of-products computed in a much wider format (Kulisch-like)
  - internal size that fits all: E5M3

FP8 have a separate unit in Coolidge (see ARITH 2023)
$\rightarrow$ not discussed in this work

Arithmetic formats for machine learning



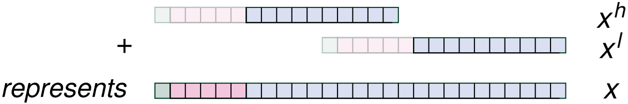
- FP16/BF16
  - internal size that fits all inputs: E8M10 (aka NVIDIA TF32)
  - sum-of-products computed in a much wider format

Oh, a new toy! Let us play with it!

The toy: this tensor core computing the mixed-precision (16/32-bit) MMA: $R = C + AB$.
Can we use it to compute a matrix product in FP32?
Of course we can (almost).

An old idea: Double-Word Arithmetic

Basic idea: represent a high-precision value x as a pair of FP values $x^h + x^l$



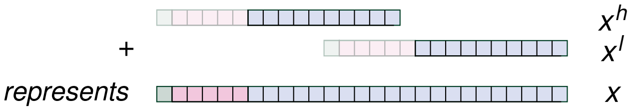
(each mantissa includes an implicit leading 1)

Oh, a new toy! Let us play with it!

The toy: this tensor core computing the mixed-precision (16/32-bit) MMA: $R = C + AB$.
Can we use it to compute a matrix product in FP32?
Of course we can (almost).

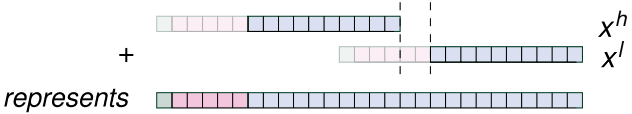
An old idea: Double-Word Arithmetic

Basic idea: represent a high-precision value x as a pair of FP values $x^h + x^l$



(each mantissa includes an implicit leading 1)

... plus classical trick: the sign bit of x^l provides an extra bit.



Example:
double-FP16 $\approx$ E5M21
(FP32 is E8M23)

In terms of software

To accelerate an FP32 matrix multiplication $R = AB$,

- decompose A and B as the unevaluated sum of FP16 matrices:

$$A \simeq A^h + A^l \tag{1}$$

$$B \simeq B^h + B^l \tag{2}$$

  Algorithm for this:

$$A^h = \textbf{toFP16}(A) \qquad \text{(rounding here)} \tag{3}$$

$$A^l = \textbf{toFP16}(A - \textbf{toFP32}(A^h)) \qquad \text{(and here).} \tag{4}$$

- then use a mixed-precision MMA

$$R \approx A^h B^h + A^h B^l + A^l B^h + A^l B^l \tag{5}$$

In terms of dot products

A mixed-precision dot-product-add of size N on FP16 vectors (x_i, y_i): $r = a + \sum_{i=0}^{N-1} x_i y_i$

a	$+$	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}	x_{11}	x_{12}	x_{13}	x_{14}	x_{15}
		y_0	y_1	y_2	y_3	y_4	y_5	y_6	y_7	y_8	y_9	y_{10}	y_{11}	y_{12}	y_{13}	y_{14}	y_{15}

accelerates a dot-product-and-add of size $N/4$ on FP32 data (u_i, v_i): $r = a + \sum_{i=0}^{N/4-1} u_i v_i$

- Decompose $u_i = u_i^h + u_i^l$ and $v_i = v_i^h + v_i^l$
- Now $\forall i \quad u_i v_i = (u_i^h + u_i^l) \times (v_i^h + v_i^l) = u_i^h v_i^h + u_i^h v_i^l + u_i^l v_i^h + u_i^l v_i^l$
- Use the mixed-precision FP16 dot-product-and-add

a	$+$	u_0^h	u_0^h	u_0^l	u_0^l	u_1^h	u_1^h	u_1^l	u_1^l	u_2^h	u_2^h	u_2^l	u_2^l	u_3^h	u_3^h	u_3^l	u_3^l
		v_0^h	v_0^l	v_0^h	v_0^l	v_1^h	v_1^l	v_1^h	v_1^l	v_2^h	v_2^l	v_2^h	v_2^l	v_3^h	v_3^l	v_3^h	v_3^l

Executive summary so far

... If we have P TFlop/s of FP16 tensor operator, we can use it to compute P/4 TFlop/s of quasi-FP32 matrix operations.

Executive summary so far

... If we have P TFlop/s of FP16 tensor operator, we can use it to compute P/4 TFlop/s of quasi-FP32 matrix operations.

Quasi-FP32 is not FP32

1. range issues: FP16 exponent is more limited than FP32
2. precision issues: emulation of 1+21 mantissa bits, whereas FP32 is 1+23.

Double BF16 instead solves range issues, but makes precision issues worse...
Triple BF16 lowers performance to X/9 TFlop/s (and 9 is not a hardware-friendly factor)

Executive summary so far

... If we have P TFlop/s of FP16 tensor operator, we can use it to compute P/4 TFlop/s of quasi-FP32 matrix operations.

Quasi-FP32 is not FP32

1. range issues: FP16 exponent is more limited than FP32
2. precision issues: emulation of 1+21 mantissa bits, whereas FP32 is 1+23.

Double BF16 instead solves range issues, but makes precision issues worse...
Triple BF16 lowers performance to X/9 TFlop/s (and 9 is not a hardware-friendly factor)

This paper:

- Proposition of **minimal hardware support** solving both problems
- (with decomposition of x into $x^h + x^l$ in hardware)
- evaluation of its overhead.

The proposed architecture

Proposed minimum intermediate format: E9S12

- Assumption: We know how to build an exact or accurate sum-of-ExMy-plus-FP32.
- What is the smallest format ExMy accomodating FP16, or BF16, or half-FP32 ?
  First idea: E8M11 (12-bit significand with the implicit 1).

Proposed minimum intermediate format: E9S12

- Assumption: We know how to build an exact or accurate sum-of-ExMy-plus-FP32.
- What is the smallest format ExMy accomodating FP16, or BF16, or half-FP32 ?
  First idea: E8M11 (12-bit significand with the implicit 1).

Why S12 and not M11?

Because E8M11 tends to imply a normalized format with an implicit 1.
Not mandatory here! When decomposing a FP32 $x = x^h + x^l$, why round x^h to the nearest? Why normalize x^l? Just splitting the bit vector is cheaper.
In short, S12 stands for a 12-bit significand, possibly un-normalized.

Proposed minimum intermediate format: E9S12

- Assumption: We know how to build an exact or accurate sum-of-ExMy-plus-FP32.
- What is the smallest format ExMy accomodating FP16, or BF16, or half-FP32 ?
  First idea: E8M11 (12-bit significand with the implicit 1).

Why S12 and not M11?

Because E8M11 tends to imply a normalized format with an implicit 1.
Not mandatory here! When decomposing a FP32 $x = x^h + x^l$, why round x^h to the nearest? Why normalize x^l? Just splitting the bit vector is cheaper.
In short, S12 stands for a 12-bit significand, possibly un-normalized.

Why E9 and not E8?

The decomposition of an FP32 with a very small exponent entails an even smaller exponent for x^l: we need one more exponent bit to capture this case. This will be cheap.

In details

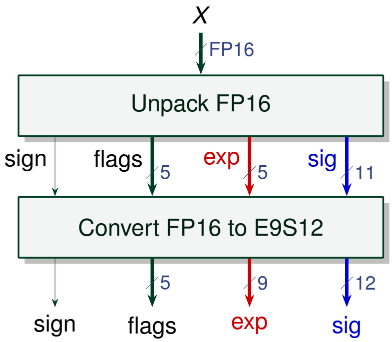
E9S12 consists of

- a sign bit
- 5 flag bits (isNormal, isInf, isNaN, isSigNaN, isZero)
- 9 exponent bits, with a fancy bias of 139=127+12
- 12 significand bits, not necessarily normalized

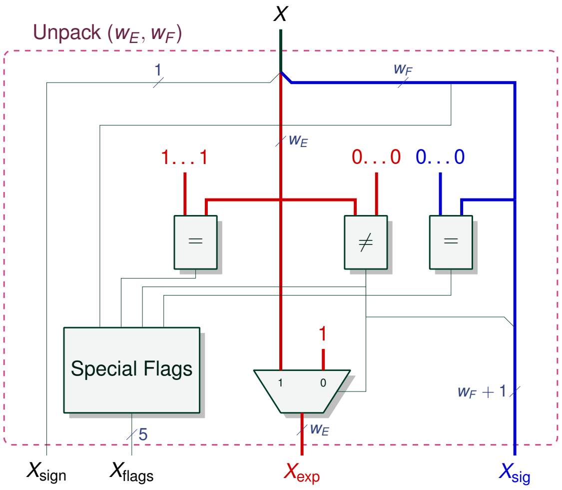
Conversion in two steps:

- unpack the floating-point format
  - extract exponent and fraction fields,
  - decode all the flags
  - prepend implicit bit to significand
- convert to E9S12
  - zero-pad significand
  - adjust exponent bias

Example for FP16

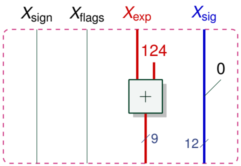


In more details, it is just simple

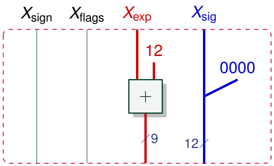


Conversion in two steps:

- unpack the floating-point format
  - extract exponent and fraction fields,
  - decode all the flags
  - prepend implicit bit to significand
- convert to E9S12
  - adjust exponent bias
  - zero-pad significand

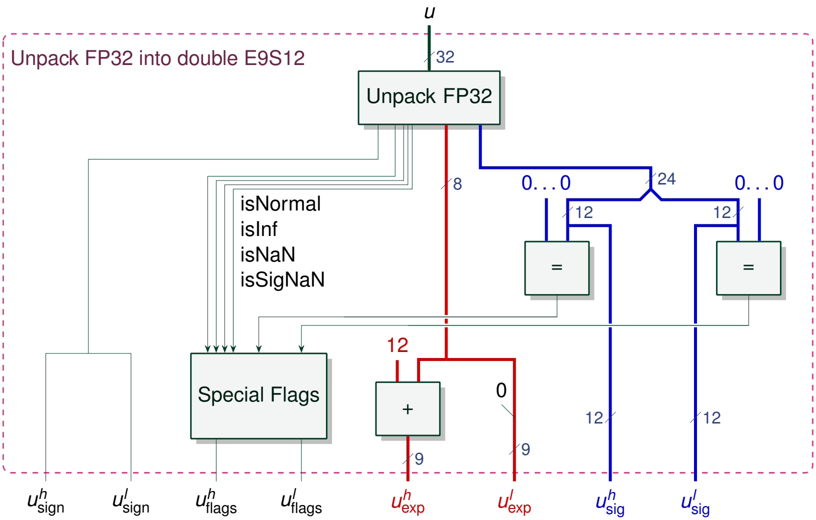


Convert FP16 to E9S12



Convert BF16 to E9S12

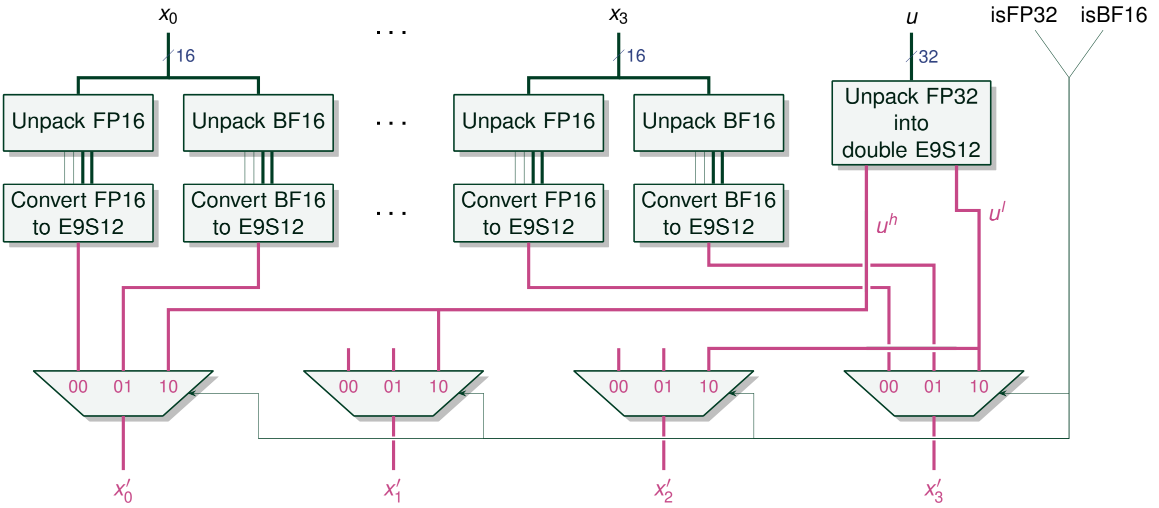
Unpacking FP32 into a double-E9S12



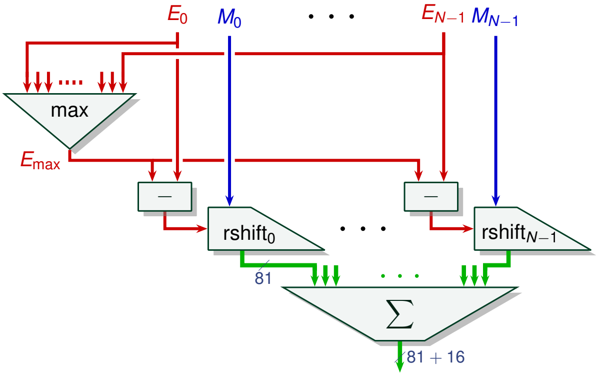
Overall multiplexing of each group of 4 inputs

$$\boxed{a} + \begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|}
\hline
u_0^h & u_0^h & u_0^l & u_0^l & u_1^h & u_1^h & u_1^l & u_1^l & u_2^h & u_2^h & u_2^l & u_2^l & u_3^h & u_3^h & u_3^l & u_3^l \\
\hline
v_0^h & v_0^l & v_0^h & v_0^l & v_1^h & v_1^l & v_1^h & v_1^l & v_2^h & v_2^l & v_2^h & v_2^l & v_3^h & v_3^l & v_3^h & v_3^l \\
\hline
\end{array}$$

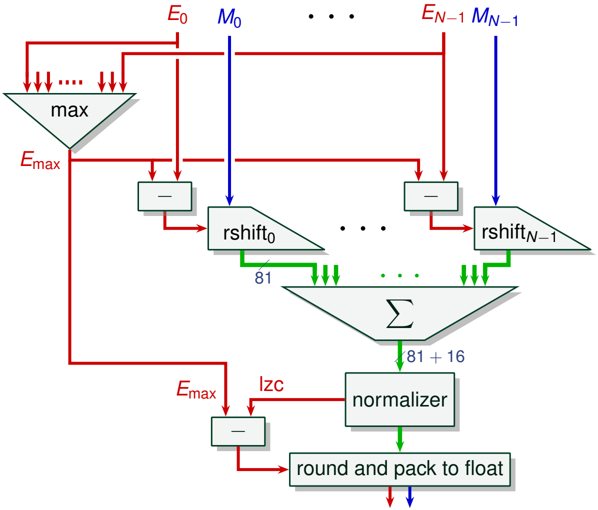
Overall multiplexing of each group of 4 inputs



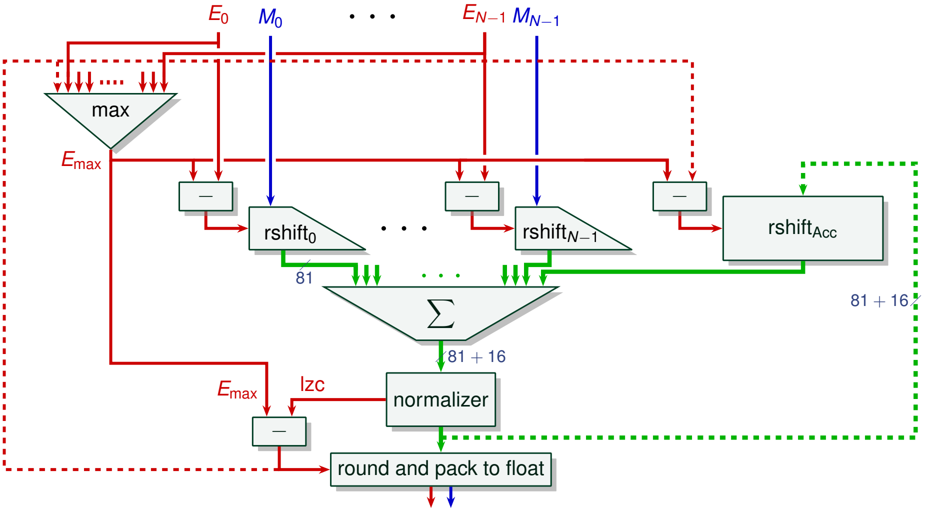
The sum of product itself



The sum of product itself



The sum of product itself (variant that registers the large accumulator)



Choices of parameters

- Correctly rounded FP16 dot product of size N requires $L = 81 + \log_2(N)$ bits.
- We round it up to $L = 81 + 16$
  so we can iterate on the operator for N up to 65000.

This is much more accurate than the competition...

Executive summary

With this architecture:

- Single correct rounding of the exact FP16 dot product with FP32 add
- Full subnormal support (easy since E9M12 is not necessarily normalized)
  - NVIDIA flushes TF32 subnormals to zero
- Results accurate to 81+16-bit for the other dot-products-and-add variants

There is a cheaper variant where we loop back an FP32

- Support of FP32 FMA (Fused Multiply and Add)
- But less accurate when using the operator iteratively

Evaluation

A trade-off between area and power

Combinatorial operators running a 333MHz to emulate pipeline at 1GHz

Operator	Area (μm^2)	Power (mW) Leakage	Power (mW) Total
Baseline: $DP16_{FP16}A_{FP32}$	1796	.000477	1.83
$DP16_{FP16/BF16}A_{FP32}$	2343	.000602	2.11
$DP4_{FP32}A_{FP32}$	1865	.000476	1.87
$DP4_{FP32}A_{FP32}$ + $DP16_{FP16/BF16}A_{FP32}$	4208	.001078	2.11
proposed	2504	.000657	2.62

- FP32 support adds 40% area to a $DP16_{FP16}A_{FP32}$
- Compared to two separate accelerators, the combined operator
  - saves 68% of area
  - saves power when idle (leakage)
  - but consumes more power when active: 25% for 16-bit vectors, or 43% for FP32.

Questions ?