Software

LEVERAGING THE BFLOAT16 AI DATATYPE FOR HIGH PRECISION COMPUTE

Alexander Heinecke, Greg Henry, Ping Tak Peter Tang

Legal Disclaimer & Optimization Notice

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more complete information visit www.intel.com/benchmarks.

INFORMATION IN THIS DOCUMENT IS PROVIDED "AS IS". NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

What is BFLOAT16

Int16 based training (KNM/1st Nervana):

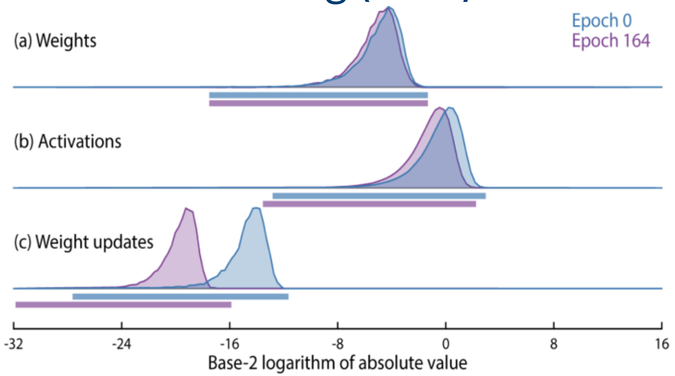
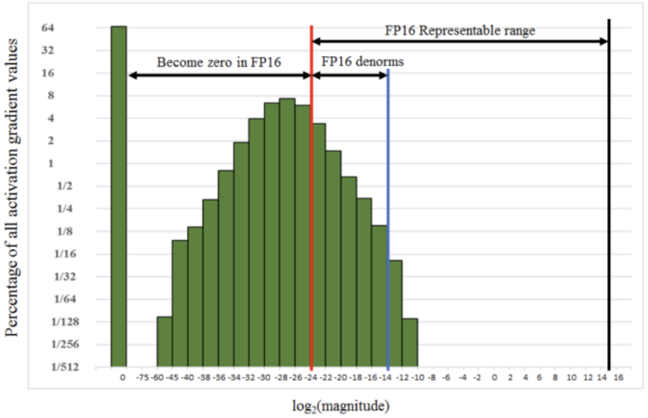


Figure 4. Distributions of tensor scales in a deep neural network and their evolution during training.

FP16 based training (GPUs):



FP32 | s | 8 bit exp | 23 bit mantissa

FP16 | s | 5 bit exp | 10 bit mantissa

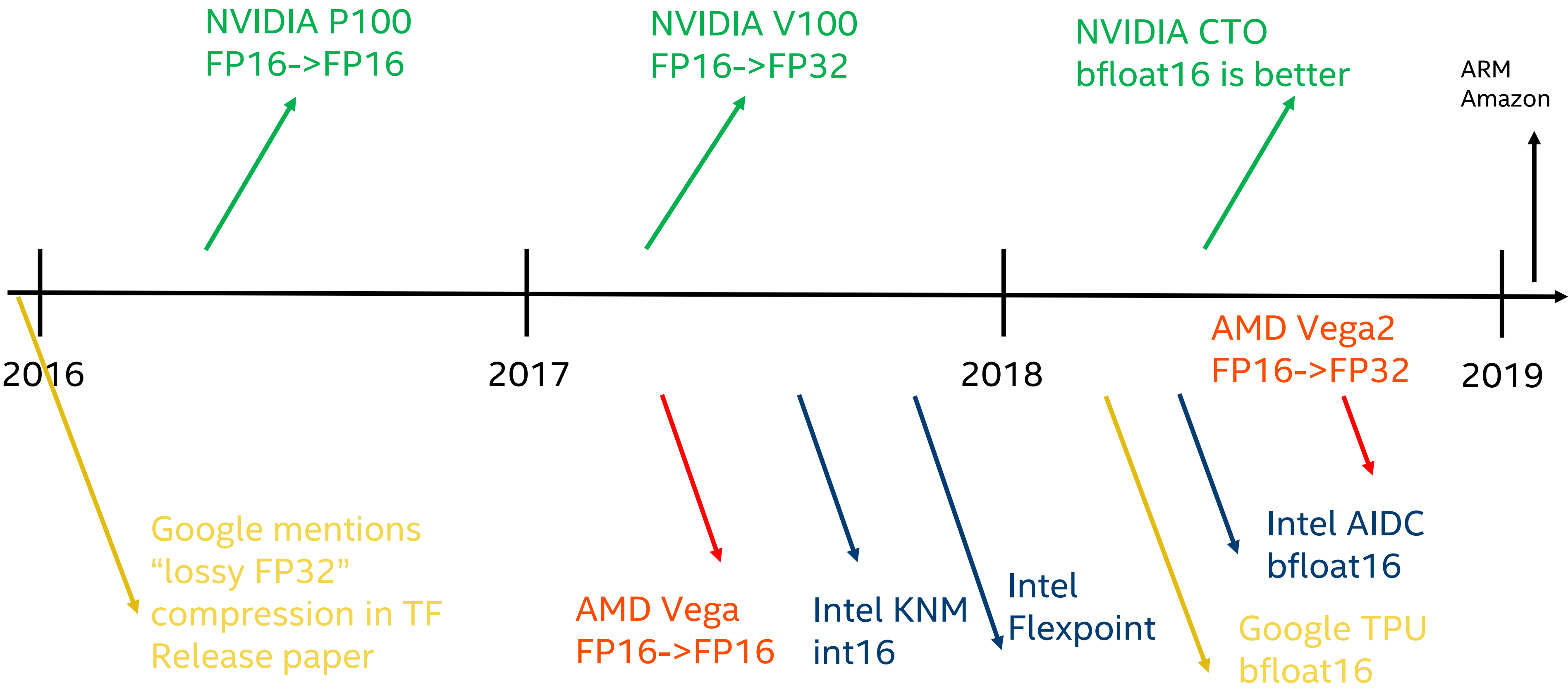
bfloat16 | s | 8 bit exp | 7bit mantissa

The best compromise among FP16, int16, Flexpoint and it even reuses a lot of inference Hardware!

Int16-KNM | s | “15 bit mantissa” | 8 bit shared exp

Flexpoint | s | “15 bit mantissa” | 5 bit shr.exp

The History Of BFLOAT16



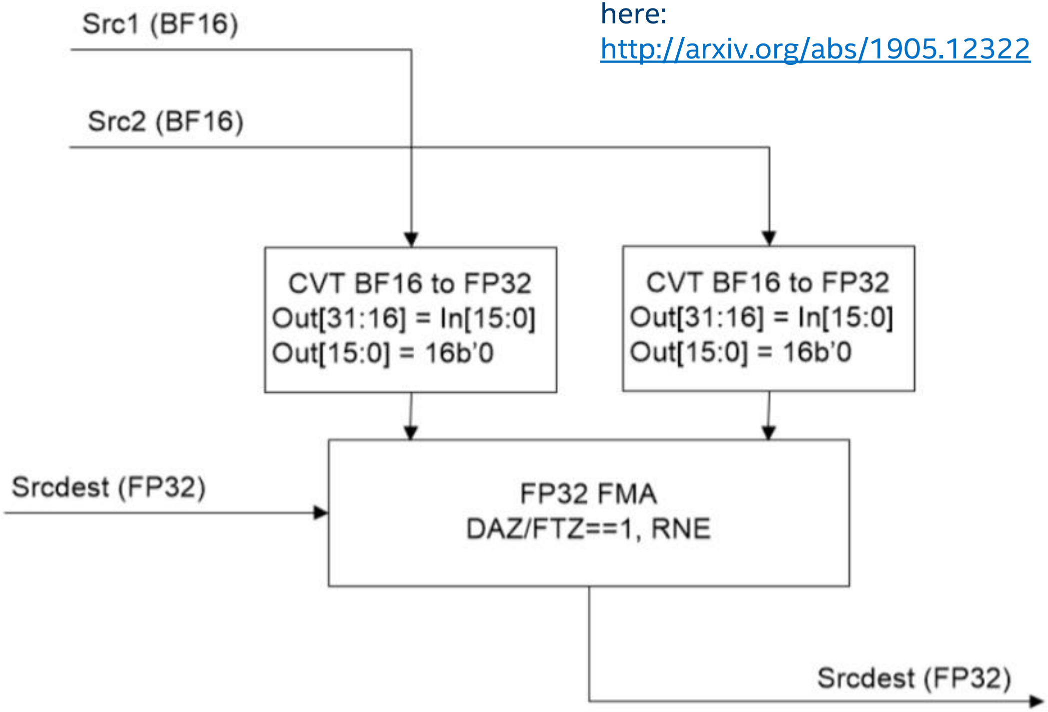
Bfloat16 on Intel Architecture

BF16 instructions for Cooper Lake now public at: https://software.intel.com/en-us/intel-architecture-instruction-set-extensions-programming-reference

3 Instructions:

- VNNI-Style dot-product with FP32 accumulate
- 2 FP32->BF16 converts (RNE) for 1024bits and 512bits inputs

Our bfloat16 evaluation for deep learning applications can be found here: http://arxiv.org/abs/1905.12322



What about combining data formats?

- Concept behind double-double: pair two doubles together and do “careful calculations” to keep track of over 100+ bits of accuracy.
- Why not do the same thing with bfloat16?
- We can have a single BF16, denoted bx1 and have roughly 8-bits of accuracy
- We can have a pair of BF16s, denoted bx2, and roughly 16-bits of accuracy
- We can have a triplet of BF16s, denoted bx3, and roughly 24-bits of accuracy
  - This is comparable to FP32!
  - This is not identical to FP32!
- Especially interesting for inner-products as we have a FP32 accumulator

Creating Doubles or Triplets Very Easy (Compared to Double-Double Algorithms)

$$
\begin{aligned}
b^{(0)} &= (\mathcal{B}_{16})a \\
b^{(1)} &= (\mathcal{B}_{16})((\mathcal{F}_{32})(a - (\mathcal{F}_{32})b^{(0)})) \\
b^{(2)} &= (\mathcal{B}_{16})((\mathcal{F}_{32})(a - (\mathcal{F}_{32})b^{(0)} - (\mathcal{F}_{32})b^{(1)}))
\end{aligned}
$$

We can write every single precision number using this split:

- $b^{(0)}$ is bfloat16 and contains the most significant bits
- $b^{(1)}$ is bfloat16 and contains the next significant bits
- $b^{(2)}$ is bfloat16 and contains the least significant bits

Let's have a more detailed look at dot-products

$$z = \mathbf{x}^T\mathbf{y} \quad \hat{z} = |\mathbf{x}|^T|\mathbf{y}| \quad \varepsilon_f = 2^{-24} \quad \gamma_{f,k} = \frac{k\varepsilon_f}{1-k\varepsilon_f}$$

The standard computation in FP32 is as follows:

$Z \leftarrow 0$
For $\ell = 1, 2, \ldots, n$:
  $Z \leftarrow \mathtt{FMA}(x_\ell, y_\ell, Z)$
End

The error bound is standard in this case, namely

$$|Z - z| \leqslant \gamma_{f,n}\hat{z}$$

→ "n rounding errors"

$Z^{(i,j)} \leftarrow 0$
For $\ell = 1, 2, \ldots, n$:
  $Z^{(i,j)} \leftarrow \mathtt{FMA}(x_\ell{}^{(i)}, y_\ell{}^{(j)}, Z^{(i,j)})$
End

$$\varepsilon_b = 2^{-8}$$

$$\gamma_{b,k} = \frac{k\varepsilon_b}{1-k\varepsilon_b}$$

$$\begin{aligned}
Z^{(0)} &\leftarrow Z^{(0,0)} \\
Z^{(1)} &\leftarrow Z^{(0,1)} + Z^{(1,0)} \\
Z^{(2)} &\leftarrow Z^{(0,2)} + (Z^{(1,1)} + Z^{(2,0)}) \\
Z^{(3)} &\leftarrow Z^{(1,2)} + Z^{(2,1)} \\
Z^{(4)} &\leftarrow Z^{(2,2)}
\end{aligned}$$

$$\begin{aligned}
|Z_2 - z| &\leqslant |Z_2 - (z^{(0)} + z^{(1)} + z^{(2)})| + |z^{(3)} + z^{(4)}| \\
&\leqslant |Z_2 - (z^{(0)} + z^{(1)} + z^{(2)})| + 1.01\varepsilon_b^3\hat{z} \\
&\leqslant |Z_2 - (Z^{(0)} + Z^{(1)} + Z^{(2)})| \\
&\quad + \sum_{i=0}^{2} |Z^{(i)} - z^{(i)}| + 1.01\varepsilon_b^3\hat{z} \\
&\leqslant \gamma_{f,2}\sum_{i=0}^{2}|Z^{(i)}| + 1.01\gamma_{f,n}\hat{z} + 1.01\varepsilon_b^3\hat{z} \\
&\leqslant 1.01(\gamma_{f,n+2} + \varepsilon_f^3)\hat{z}.
\end{aligned}$$

→ A split into 3 BF16 is slightly worse, but should not be relevant in practice

Doing only the most relevant computations

Number of Bfloat16s used in the split	Number of Multiplies to Do
1 a1*b1	1 (bin 1 only)
2 (a1,a2)*(b1,b2)	3 (bins 1 and 2 only)
3 (a1,a2,a3)*(b1,b2,b3)	6 (bins 1, 2, and 3 only)
4 (a1,a2,a3,a4)*(b1,b2,b3,b4)	10 (bins 1-4 only)

Add up the terms in this order: from the highest bin (likely smallest numbers) to the lowest bin (likely biggest numbers)

But why do all this extra work??

- Because BFloat16 might be many times faster than FP32 in AVX-512.
- Suppose for the sake of argument, that it's 16x over FMA-based FP32 in AVX-512.
  - 3 bfloats using 6 multiplies will have ~6x the flops
  - There's still potential speed-up over FP32
  - Similar accuracy but faster (assuming the splits can be done relatively free)

BF16 Speed-Up Over FP32	Combined BF16 For FP32
8	1.3 = 8/6
16	2.7 = 16/6
32	5.2 = 32/6

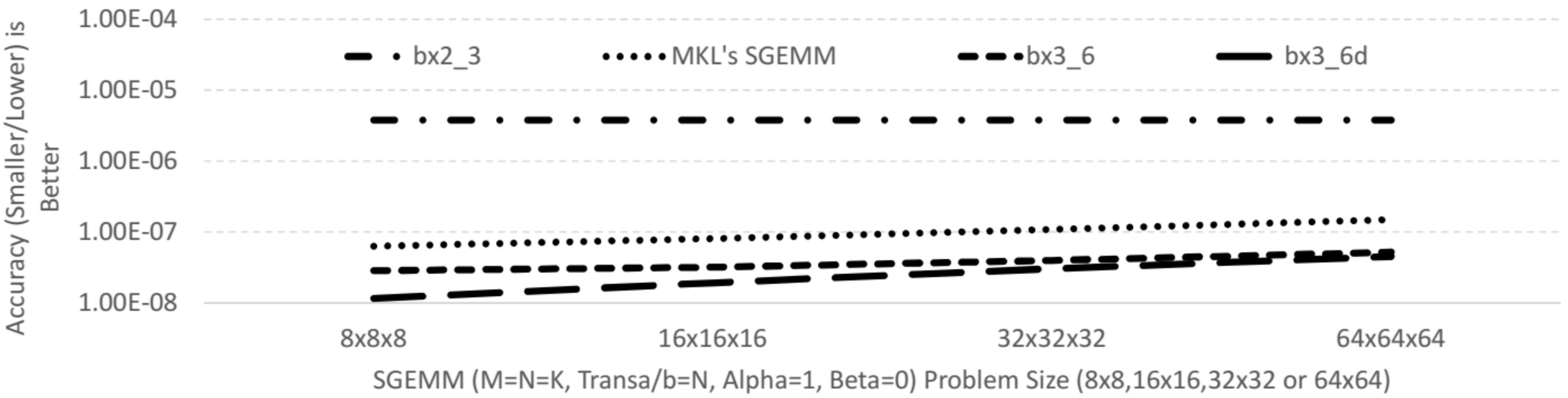
BF16x3 Can be Invisible to SGEMM (FP32 matrix-matrix multiply) users

- Different numeric results, but in many cases greater accuracy
- Time in BF16x3-GEMM is: the decomposition, matrix multiplication, adding up the results at the end.
  - The middle matrix-multiply step is O(n^3) and must be done 6 times
  - The first and last step are O(n^2) and must be done once
  - Hopefully most of the time will be in GEMM…
- The data can "start" in FP32 format, and "finish" in FP32 format
- The results may be faster… Again, BF16 might be 8x-32x faster than FP32
  - 6x more computations executed at 16x faster? A win!

Okay, so it works but what do the experiments say?

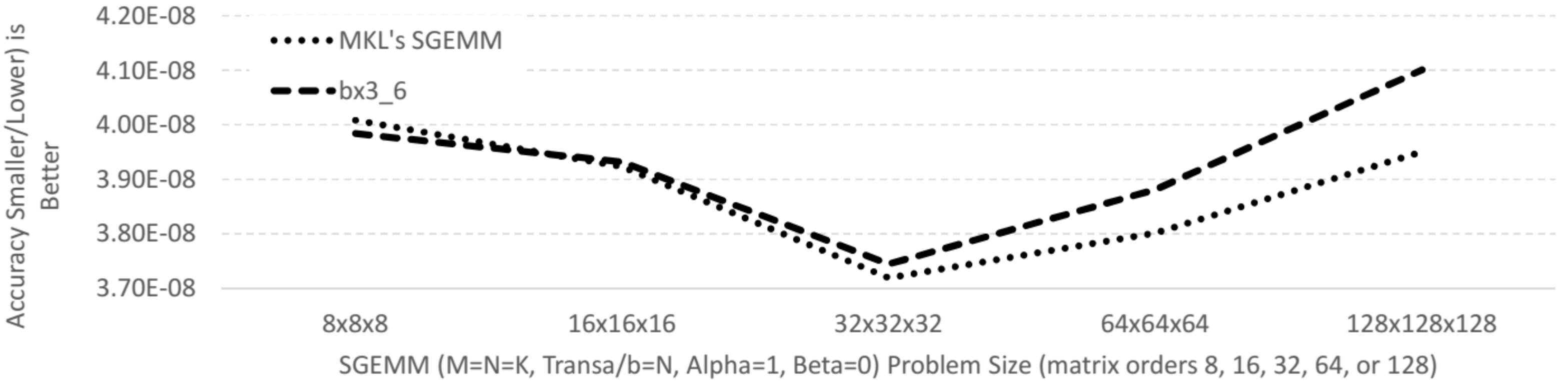
- Tested lots of dot products with varying ranges
- Tested lots of GEMM variants and cases
  - Small Range (Uniformly Random Distribution in a close range like [-1,1]
  - Huge Range (Uniformly Random Exponents)
  - Medium Range (Gaussian Distribution on the exponents)
- Tested LU decomposition

Matrix Multiply - GEMM

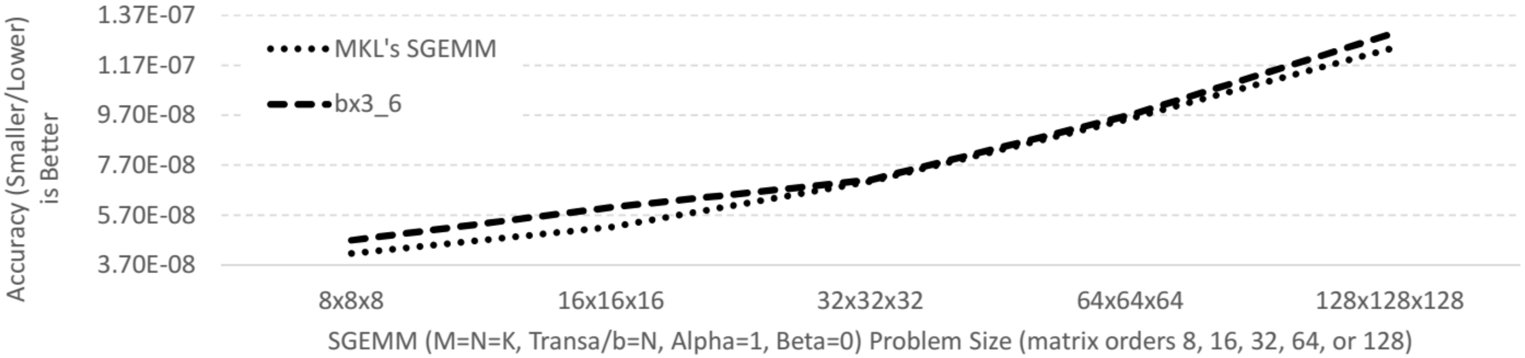


Various GEMM average relative error vs. DGEMM (||*A* x *B* – DGEMM||/||DGEMM||) over 1000 runs compared to original fp64 data in [-1.0,1.0] range with drand48() randomization. bxA B[d] means breaking each matrix up into A bfloat16 matrices, doing B products. Optionally, collect the final answer with "d" (double precision) or not.

GEMM contd.

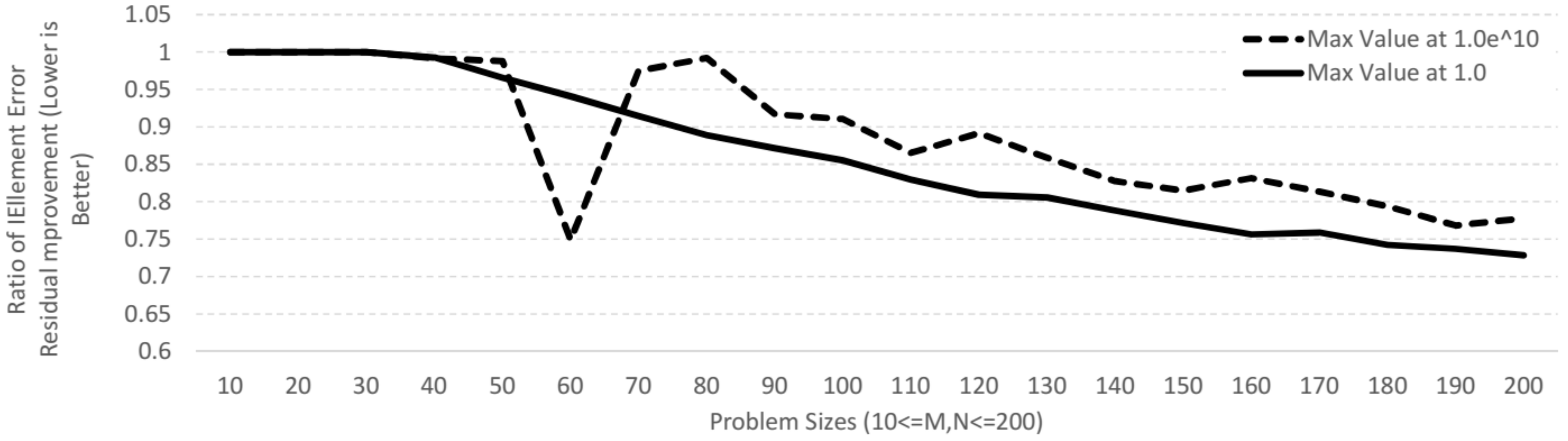


huge Range: Max. exponent distribution



medium Range: Gaussian exponent distribution

LU Factorization



SGETRF vs BFLOAT16x3 6 LU Decomposition: Element errors average improvement over a 100 runs for *N* x *N* square matrices with an extremely large range [-1^10, 1^10] and matrices with a small range [-1, 1]

Conclusions

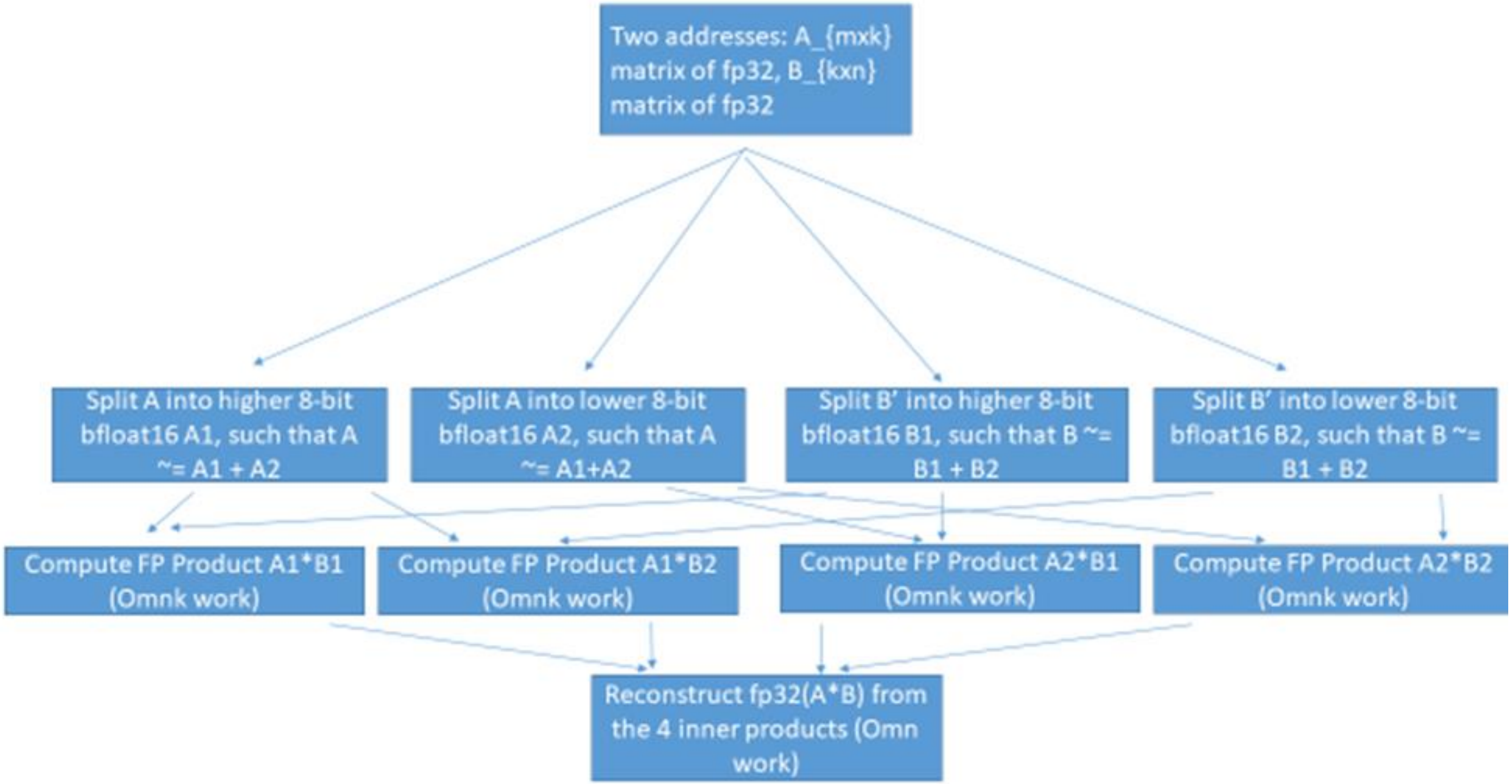
DL focused hardware solutions (which implement such mixed-precision FMA units) can be utilized for higher precision linear algebra

- We can match the accuracy of SGEMM and friends using BF16 mixed precision FMAs only
- BF16 mixed precision FMAs are potentially much energy and space friendly than pure FP32 units

The solutions present can be transferred to FP16 FMA with FP32 accumulate and also be combined with iterative refinement operations (see our paper)

Thank you for your time

Software


Two addresses: A_{mxk} matrix of fp32, B_{kxn} matrix of fp32
Split A into higher 8-bit bfloat16 A1, such that A ~= A1 + A2
Split A into lower 8-bit bfloat16 A2, such that A ~= A1+A2
Split B' into higher 8-bit bfloat16 B1, such that B ~= B1 + B2
Split B' into lower 8-bit bfloat16 B2, such that B ~= B1 + B2
Compute FP Product A1*B1 (Omnk work)
Compute FP Product A1*B2 (Omnk work)
Compute FP Product A2*B1 (Omnk work)
Compute FP Product A2*B2 (Omnk work)
Reconstruct fp32(A*B) from the 4 inner products (Omn work)

Worse case for the 3-bfloat split break-down?

- In general, accuracy is equivalent to FP32, but the worst case error is unfortunately worse.
- The “favorite bad” case is when due to being at the extreme negative exponent, none of the other bfloat16 numbers are non-zero.
  - Suppose x has an exponent of -126 and we wish: x = b1+b2+b3
  - b1 = bfloat(x), but suppose x has lots of interesting bits in position 0-15, these will be lost on b1.
  - b2 = b3 = 0 (Special bad case because it’s a conversion error.)
  - So any errors involving (b1,b2,b3) will have identical accuracy as simply 1 bfloat16 conversion- in other words, typically 2-3 decimal places only