Training LLMs with MXFP4

Albert Tseng[1*] Tao Yu[2] Youngsuk Park[2]
[1]Cornell University [2]Amazon Web Services
*Work done during internship at Amazon

amazon | science

Low Precision Training

Modern AI accelerators come with hardware acceleration for low precision (LP) matrix multiplications (GEMMs). Reducing the precision linearly increase GEMM throughput, making LP GEMMs attractive for compute-bound model training. However, reducing precision increases distortion, which can harm model quality.

Datatype	Scale Bits	Block Size	Exponent Bits	Mantissa Bits	Speedup vs. FP32
MXFP4	8	32	2	1	8X
FP8 E4M3	-	-	4	3	4X
FP8 E5M2	-	-	5	2	4X
FP16	-	-	5	10	2X
BF16	-	-	8	7	2X

One way to use LP GEMMs is with **mixed precision (MP) training**, which quantizes high precision parameters for LP GEMMs but performs updates in high precision. MP works for BF/FP16 and FP8, but is **insufficient for MXFP4**.

MXFP4

The MXFP4 datatype shares an INT8 scale across 32 contiguous FP4 entries in memory. The value represented by a scale s and block b is $2^s b$

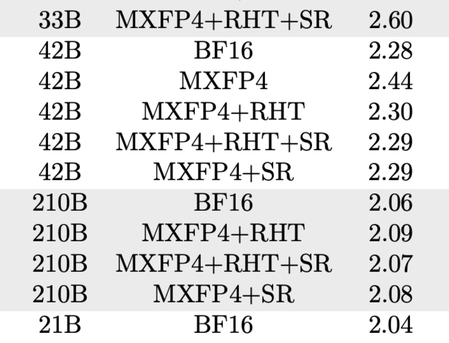
Shared INT8 Scale s	32 Contiguous FP4 Numbers

We can quantize a vector of reals to a MXFP4 vector in a hardware-efficient way with the OCP quantization algorithm (Algorithm 1). However, Algorithm 1 is **biased** since it scales elements to be < 8 ($\text{emax}_{\text{FP4}} = 2$) before quantizing to FP4. Since the largest FP4 value is 6, numbers between 6 and 8 get clipped to 6.

Algorithm 1 Convert vector of scalar floats $V \in \texttt{HP\_DTYPE}^k$ to an MX block $\{X, P \in \texttt{LP\_DTYPE}^k\}$ (from (Project, 2023))

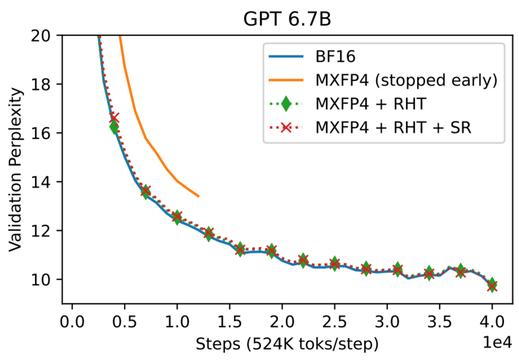
Require: $\texttt{emax}_{\texttt{elem}}$ = exponent of largest normal in LP_DTYPE, $k = 32$ for hardware support.

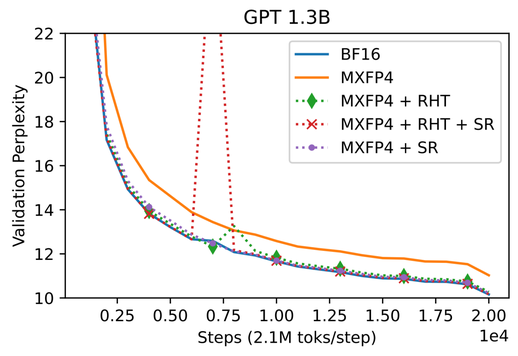
1: $\texttt{shared\_exp} \leftarrow \lfloor \log_2(\max_i(|V_i|)) \rfloor - \texttt{emax}_{\texttt{elem}}$
2: $X \leftarrow 2^{\texttt{shared\_exp}}$
3: **for** $i = 1$ to k **do**
4: $\quad P_i = \texttt{quantize\_to\_LP}(V_i / X)$
5: **end for**
6: **return** X, $\{P_i\}_{i=1}^k$



Pretraining & Finetuning Experiments

To test our recipe, we pretrained GPT 345M, 1.3B, and 6.7B models on up to 210B tokens of the Wikipedia corpus. Our recipe achieves a **< 0.1 perplexity gap** vs. a BF16 backward pass and works with both BF16 and FP8 forward passes.





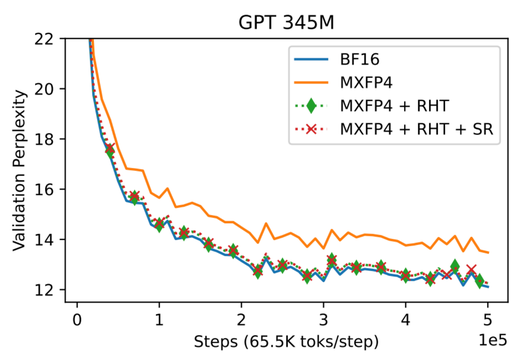
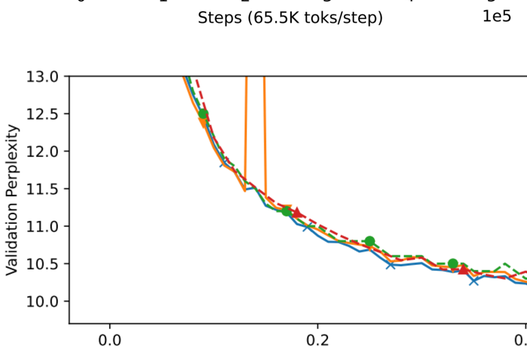


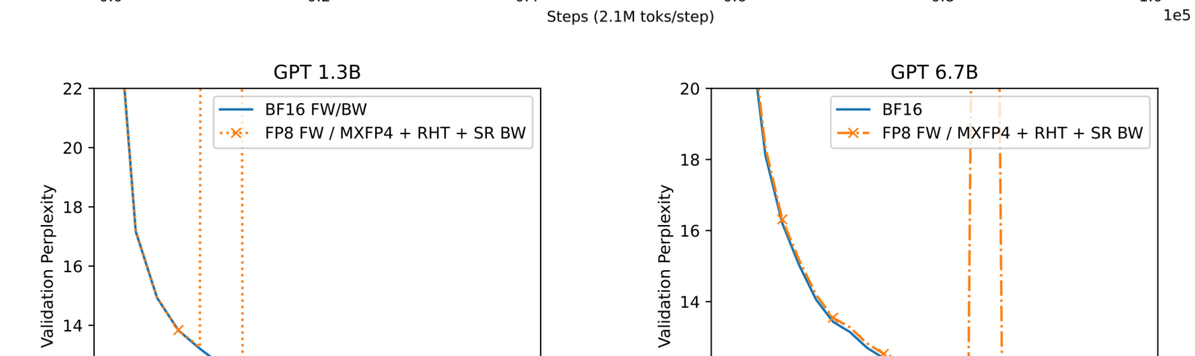
Table 2: Final losses for GPT models trained on the GPT2 Wikipedia corpus. All models were trained with BF16 mixed precision for the forward pass.

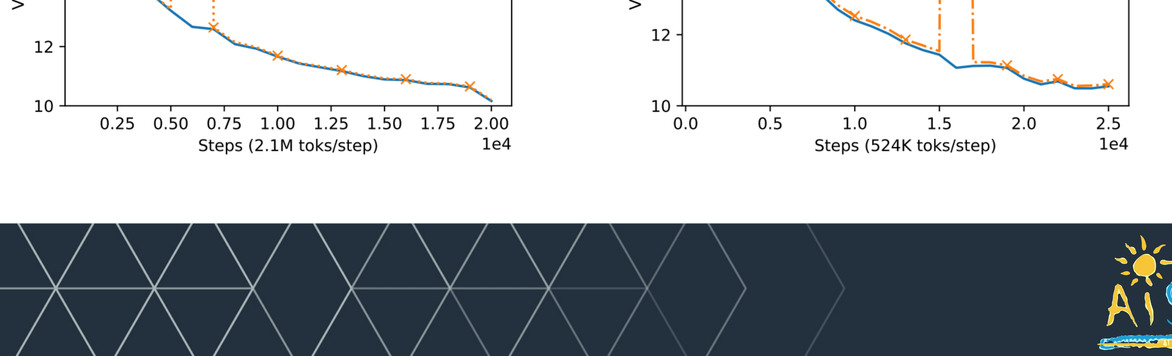
Params.	Toks.	Bwd. Prec.	Train. Loss	Val. Loss
345M	33B	BF16	2.58	2.49
345M	33B	MXFP4	2.73	2.60
345M	33B	MXFP4+RHT	2.60	2.51
345M	33B	MXFP4+RHT+SR	2.60	2.51
1.3B	42B	BF16	2.28	2.32
1.3B	42B	MXFP4	2.44	2.40
1.3B	42B	MXFP4+RHT	2.30	2.33
1.3B	42B	MXFP4+RHT+SR	2.29	2.32
1.3B	42B	MXFP4+SR	2.29	2.32
1.3B	210B	BF16	2.06	2.29
1.3B	210B	MXFP4+RHT	2.09	2.31
1.3B	210B	MXFP4+RHT+SR	2.07	2.29
1.3B	210B	MXFP4+SR	2.08	2.29
6.7B	21B	BF16	2.04	2.27
6.7B	21B	MXFP4+RHT	2.05	2.28
6.7B	21B	MXFP4+RHT+SR	2.08	2.27

Downstream Finetuning Performance

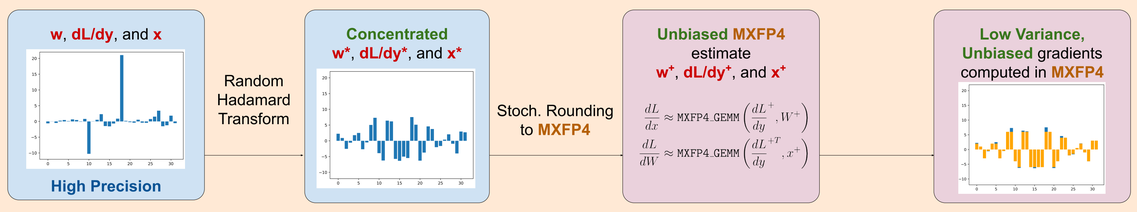
Model	ArcC	ArcE	PiQA	BoolQ	Wino
BF16	**23.1**	**49.2**	60.5	53.3	**52.0**
MXFP4	22.2	47.8	**61.3**	**59.6**	49.6
BF16 Tulu2	25.6	**50.6**	62.7	59.6	51.6
MXFP4 Tulu2	**25.9**	49.9	**62.9**	**60.5**	**51.8**







Our Approach



We present the **first near-lossless MXFP4 training recipe** using fully hardware-accelerated operations. We use MXFP4 to calculate **unbiased gradient estimates for linear layers**, which consists of >1/2 training FLOPs. Specifically,

- We use **stochastic rounding (SR)** to make MX quantization unbiased.
- We use a memory-bound construction of the **random Hadamard transform (RHT)** to theoretically bound the variance of SR in gradient estimates.

Low Variance Unbiased Quantization

To achieve unbiased MXFP4 quantization, we use stochastic rounding (SR). The latest AI accelerators come with hardware SR support, making SR practical. In SR, inputs are randomly quantized such that the quantized number equals the original number *in expectation*. SR is usually implemented with *dithering*, which adds uniform random noise to an input before performing nearest rounding.

Algorithm 2 Unbiased quantization of $V \in \texttt{HP\_DTYPE}^k$ to an MXFP4 block $\{X, P \in \texttt{LP\_DTYPE}^k\}$

Require: $\texttt{emax}_{\texttt{elem}}$ = exponent of the largest normal number in LP_DTYPE

1: $\texttt{shared\_exp} \leftarrow \lfloor \log_2(\max_i(|V_i|)) \rfloor - \texttt{emax}_{\texttt{elem}}$
2: $X \leftarrow 2^{\texttt{shared\_exp}}$
3: **for** $i = 1$ to k **do**
4: $\quad V_i \leftarrow \frac{3}{4} V_i$
5: $\quad P_i = \texttt{stochastic\_round\_to\_FP4}(V_i / X)$
6: **end for**
7: **return** X, $\{P_i\}_{i=1}^k$

$$\delta \sim \mathcal{U}(-0.5, 0.5)$$

$$\text{SR}_{\text{dither}}(x) = \begin{cases} \lfloor x \rfloor & x + \delta < \lfloor x \rfloor + \frac{1}{2} \\ \lceil x \rceil & x + \delta \geq \lfloor x \rfloor + \frac{1}{2} \end{cases}$$

With SR, we can perform unbiased quantization of the GEMM operands in the backward pass and ultimately compute unbiased gradient estimates with pure-MXFP4 GEMMs. However, since SR is stochastic, it adds variance to the GEMM output. First, we characterize the variance SR adds to a MXFP4 GEMM:

Theorem 1 Let A and B be two size-b vectors $\in \mathbb{R}^b$, and let $\mathcal{Q}$ perform Algorithm 2. Then, the variance of $\mathcal{Q}(A)^T\mathcal{Q}(B)$ is $\mathcal{O}(b\|A\|_\infty\|B\|_\infty)$

Since the variance of the output is bounded by the largest magnitude element, we can reduce the variance by applying a random orthogonal transform to the operands. In our experiments, we use the **random Hadamard transform (RHT)**, which is *fast to randomize* and runs in *O(n log n)* time. The RHT performs

$$x \leftarrow HSx \qquad H_n = \frac{1}{2^{n/2}} \begin{bmatrix} H_{n-1} & H_{n-1} \\ H_{n-1} & -H_{n-1} \end{bmatrix}$$

and lets us bound the variance of the MXFP4 SR GEMM output:

Theorem 2 The variance of $\mathcal{Q}(HSA)^T\mathcal{Q}(HSB)$ is, with probability $\geq (1-\epsilon)^2$, $\mathcal{O}(\log(2b/\epsilon)\|A\|_2\|B\|_2)$

Hardware Considerations

Directly applying the RHT is inefficient due to how LLMs are trained. When computing $dL/dW = (dl/dy)^T x$, the RHT happens along the batch dimension. In data-parallel setups where data is sharded across devices, this requires expensive communication. Furthermore, while the RHT admits a O(n log n) matrix-vector product, doing so still adds FLOPs to already *compute bound* training.

We solve these problems by applying a blockwise RHT along a small number (~2-4) of MX blocks (Algorithm 3). This makes the RHT *memory bound* on modern accelerators while still achieving good quality due to MX's scaling.

Algorithm 3 MXFP4 linear layer (no bias) backward pass with the random Hadamard transform.

Require: Gradient of output $\frac{dL}{dy} \in \mathbb{R}^{b \times m}$, activations $x \in \mathbb{R}^{b \times n}$, weights $W \in \mathbb{R}^{m \times n}$, block size $g \leq 256, 32|g, g|m, g|n$.

1: $H \leftarrow$ Hadamard matrix $H_b \in \mathbb{R}^{m \times m}$.
2: Sample random sign vector $S \in \{\pm 1\}^b$.
3: $G' \leftarrow \left(\left(\frac{dL}{dy}\right).\texttt{view}\left(\frac{bm}{g}, g\right)\right)\texttt{diag}(S)H$
4: $W' \leftarrow H^T\texttt{diag}(S)\left(W.\texttt{view}\left(g, \frac{nm}{g}\right)\right)$
5: $GT' \leftarrow \left(\left(\frac{dL}{dy}^T\right).\texttt{view}\left(\frac{bm}{g}, g\right)\right)\texttt{diag}(S)H$
6: $X' \leftarrow H^T\texttt{diag}(S)\left(x.\texttt{view}\left(\frac{bn}{g}, g\right)\right)$
7: $\frac{dL}{dx} \leftarrow \texttt{MXFP4\_GEMM}(G', W')$
8: $\frac{dL}{dW} \leftarrow \texttt{MXFP4\_GEMM}(GT', X')$
{Where MXFP4_GEMM forms MX groups along the reduction dimension and uses either Algorithm 1 or 2 to quantize to MXFP4.}
9: **if** Using Algorithm 2 **then**
10: $\quad \frac{dL}{dx} \leftarrow \frac{16}{9}\frac{dL}{dx}$
11: $\quad \frac{dL}{dW} \leftarrow \frac{16}{9}\frac{dL}{dW}$
12: **end if**
13: **return** $\frac{dL}{dx}, \frac{dL}{dW}$

Effect of RHT Blocksize

As expected (Theorem 2), increasing the RHT blocksize improves quality. However, increasing the blocksize past a certain point (~256) makes the RHT compute bound, slowing down training.

BW Pass	BF16	g=32	g=64	g=128	g=256
Val. PPL	11.89	12.02	12.01	11.98	11.98

Table 4: Validation perplexity for training GPT 345M on 33B tokens with various RHT blocksizes. Increasing the RHT block size improves performance by reducing the variance of stochastic rounding.

BW Pass	FP16	INT8 no RHT	INT4 no RHT	+ RHT g=64	+ RHT g=128	+ RHT g=256	+ RHT g=1024 dense	+ RHT g=1024 $\mathcal{O}(n \log n)$
E2E tok/s	46983	55469	67306	64335	64171	63979	61186	62640
BW tok/s	72563	94688	133952	123056	122734	121823	112299	120495

Table 5: Throughput for a FP16 forward pass and specified backward pass of a Llama 2 70B decoder layer. Measured on a NVIDIA A100; see Section 4.2 for more details. Since the A100 can perform INT4 GEMMs 4× faster than FP16 GEMMs, these numbers represent the expected speedup of MXFP4 on supported hardware.

albert@cs.cornell.edu
taou@amazon.com
pyoungsu@amazon.com