Training LLMs with MXFP4

Albert Tseng[†]
Cornell University
albert@cs.cornell.edu

Tao Yu
AWS AI
taou@amazon.com

Youngsuk Park
AWS AI
pyoungsu@amazon.com

Abstract

Low precision (LP) datatypes such as MXFP4 can accelerate matrix multiplications (GEMMs) and reduce training costs. However, directly using MXFP4 instead of BF16 during training significantly degrades model quality. In this work, we present the first near-lossless training recipe that uses MXFP4 GEMMs, which are $2\times$ faster than FP8 on supported hardware. Our key insight is to compute unbiased gradient estimates with stochastic rounding (SR), resulting in more accurate model updates. However, directly applying SR to MXFP4 can result in high variance from block-level outliers, harming convergence. To overcome this, we use the random Hadamard tranform to theoretically bound the variance of SR. We train GPT models up to 6.7B parameters and find that our method induces minimal degradation over mixed-precision BF16 training. Our recipe computes $> 1/2$ the training FLOPs in MXFP4, enabling an estimated speedup of $> 1.3\times$ over FP8 and $> 1.7\times$ over BF16 during backpropagation.

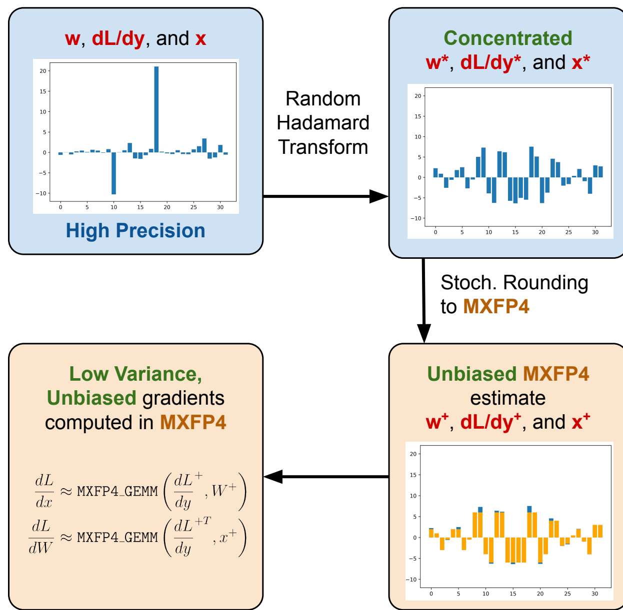

Figure 1: Our method uses stochastic rounding (SR) to compute unbiased gradients and the random Hadamard transform to bound the variance of SR. This enables us to perform more accurate model updates with MXFP4 in the backward pass, enabling a speedup of $> 1.3\times$ over FP8 and $> 1.7\times$ over BF16.

1 Introduction

The latest large language models (LLMs) have billions of parameters that are trained on trillions of tokens, making them incredibly expensive to train. For example, training Llama 3.1 405B required 3×10^{24} floating point operations (FLOPs), or over 10000 GPUs for multiple months (Dubey et al., 2024). Recent hardware accelerators have started supporting low precision floating point ($\leq$ 16 bit) matrix mulitiplications (GEMMs). Compared to 32-bit GEMMs, hardware-accelerated low precision (LP) GEMMs run at significantly higher throughputs. For example, FP8 GEMMs can be $4\times$ faster than FP32 GEMMs and also more energy efficient (NVIDIA, 2024a).

Since LLM training is compute bound in matrix multiplications, LP GEMMs can accelerate training. Almost all modern LLMs are trained with 16 bit GEMMs (Touvron et al., 2023; Dubey et al., 2024), and some even use FP8 GEMMs (Peng et al., 2023). Using 16 bit GEMMs halves the cost of matrix multiplications and improves end-to-end throughput by almost $2\times$ (Micikevicius et al., 2018). However, there is no free lunch

[†]Work done during internship at AWS. Correspondence to Tao Yu. Our code is available here.

Proceedings of the 28th International Conference on Artificial Intelligence and Statistics (AISTATS) 2025, Mai Khao, Thailand. PMLR: Volume 258. Copyright 2025 by the author(s).

with low precision training. Reducing the GEMM precision increases quantization distortion and can cause numerical instability.

To counteract these issues, the recently introduced Microscaling (MX) family of datatypes uses a shared blockwise scale across multiple floating point numbers (Project, 2023). For example, MXFP4 uses an INT8 scale s for every contiguous block v of 32 FP4 numbers to represent $2^{s-1}v$, where 1 is the exponent bias for FP4. This scale enables a significantly wider range of representable numbers at the cost of an extra $8/32 = 0.25$ bits per entry. However, MX alone is not enough to enable lossless low precision training with FP4. As we show in Section 4, directly using MXFP4 in even only the backward pass of decoder linear layers significantly degrades model quality.

In this work, we introduce two techniques that enable near-lossless distributed training with MXFP4. Our method hinges on computing low-variance, unbiased gradient estimates that enable more accurate model updates. First, we use stochastic rounding to compute unbiased GEMMs. Then, we use a memory-bound construction of random Hadamard transform to reduce the effect of outliers and theoretically bound the variance of SR, aiding convergence. We apply our method to decoder linear layers and show that it incurs minimal degradation over BF16 mixed precision training when pretraining GPT models up to 6.7B parameters. Our recipe computes over half the training FLOPs in MXFP4 and can significantly accelerate pretraining. In summary, we:

- Introduce a MXFP4 training recipe that uses stochastic rounding and the random Hadamard transform compute unbiased, low-variance gradient estimates during backpropagation.
- Pretrain GPT models up to 6.7B and show that our training recipe closes the MXFP4-BF16 gap to < 0.1 validation perplexity.
- Show that our RHT and SR constructions add minimal overhead to MXFP4 GEMMs, giving a theoretical speedup of $> 1.3\times$ and $> 1.7\times$ over a FP8 and BF16 backward pass, respectively.

2 Background and Related Works

2.1 Low Precision Datatypes and IEEE 754 Floating Point Numbers

Traditionally, low precision (LP) datatypes refer to datatypes that use significantly fewer than 32 bits to represent a single number. For example, FP16 uses 16 bits to represent floating point numbers. While there are many different low precision datatypes, including stateful ones (Tseng et al., 2024b), a certain subset has been standardized under the IEEE754 floating point (FP) (IEEE, 2019). These datatypes often come with hardware acceleration for compute-bound workloads.

Table 1: Common HW supported FP datatypes.

Name	Bits			
	Total	Sign	Exponent	Mantissa
FP64	64	1	11	52
FP32	32	1	8	23
FP16	16	1	5	10
BF16	16	1	8	7
FP8 E4M3	8	1	4	3
FP8 E5M2	8	1	5	2
FP4	4	1	2	1

IEEE floats (Table 1) are defined with 1 sign bit, e exponent bits, and m mantissa bits. In shorthand, a $1+m+e$ bit datatype is written as EeMm. The actual "normal" value represented by an IEEE float with sign bit S, mantissa M, and exponent E is

$$(-1)^S(1+M)2^{E-\texttt{bias}},$$

where `bias` is a datatype-dependent integer exponent bias offset specified by IEEE (2019). This exponent-mantissa construction means FP datatypes are scale-invariant with respect to quantization signal-to-noise ratio (SNR) bar over/underflow (Blake et al., 2023).

2.2 LLM Training

The most common way to train a LLM involves computing a loss function, computing the gradient of the loss function with respect to the model parameters, and then updating the parameters with gradient information. For example, when pretraining a decoder-only LLM, one might use an autoregressive cross-entropy-based loss and the AdamW optimizer (Touvron et al., 2023; Dubey et al., 2024). While the exact training setup may differ, the core bottlenecks of training are the compute-bound forward and backward passes that calculate the loss and gradients, respectively. Within these two components, the majority of the FLOPs are in the linear layers – at 30B parameters, over 90% of the FLOPs are in the linear layers (Casson, 2023).

The forward pass for a linear layer with input dimension n and output dimension m computes $y = xW^T + b$, where $W \in \mathbb{R}^{m\times n}$ is a parameter matrix and $b \in \mathbb{R}^m$ is an optional bias term. To backpropagate through a linear layer, we need to calculate the gradient of y with respect to x, W, and b. These are given by $\frac{dL}{dx} = \frac{dL}{dy}W$, $\frac{dL}{dW} = \frac{dL}{dY}^T x$, and $\frac{dL}{db} = \mathbb{1}\frac{dL}{dy}$, where $\mathbb{1}$ is the all-ones

vector and $\frac{dL}{dy}$ is the backprop output from the previous (going backwards) operation in the chain rule (Johnson, 2017). Each linear layer requires 3 computationally intensive matrix multiplications (xW^T, $\frac{dL}{dx}$, and $\frac{dL}{dW}$), 2 of which are in the backward pass.

2.3 Mixed Precision Training

One way to accelerate training is with "mixed precision" (MP) training. In MP, parameters are kept in high precision and GEMM operands are converted to a LP datatype for a LP GEMM. MP is a simple way to achieve the throughput benefits of LP datatypes since quantization usually has minimal overhead. End to end, BF16 MP is often $> 70\%$ faster than FP32 training (Shoeybi et al., 2020). However, quantization introduces distortion in the GEMM operands and thus outputs. Since the forward and backward passes all happen in low precision, both the loss and the model updates can deviate from their "true" values. At low bitrates $\ll 16$, distortion can degrade model quality and even cause divergence, necessitating advanced training recipes. For example, FP8 MP recipes typically use E4M3 (more precision) in the forward pass and E5M2 (more range) in the backward pass due to the different properties of gradients, weights, and activations (Peng et al., 2023; Engine).

At 4 bits, quantization distortion becomes even more difficult to manage. Xi et al. (2023) train smaller non-GPT transformers with INT4 GEMMs by using the non-randomized Hadamard transform in the forward pass and leverage score sampling (LSS) in the backward pass. Since LSS introduces additional overhead, they were only able to achieve an end-to-end speedup of 30% over FP16, which is on par with FP8 mixed precision training (Peng et al., 2023). We are also aware of a concurrent work by Wang et al. (2025) that trains LLMs with FP4. There, the authors train billion parameter GPT models with FP4 in both the forward and backward pass by using a differentiable gradient estimator and keeping outliers in high precision, resulting in a perplexity gap of > 0.5. Since their work was released after our paper went through the review process, we reserve a full comparison for future work.

2.4 Stochastic Rounding

Mixed precision requires quantizing from a higher precision tensor to a LP tensor at every step – this opens up flexibility in how the actual quantization happens. The canonical "nearest rounding" (NR) method rounds each high precision number to its closest representable value in the LP datatype (IEEE, 2019). However, NR is not *unbiased*, which we later show to be detrimental to low precision training. One way to achieve unbiased rounding is with "stochastic rounding" (SR), which randomly rounds a number to a representable value in the LP datatype so that, in expectation, the rounded number equals the original number (Croci et al., 2022).

SR can be implemented efficiently through *dithering*, which adds random uniform noise to the input number and then performs NR (Croci et al., 2022). For example, Amazon's Trainium line of chips can perform SR with dithering while adding less than 2% overhead to a BF16 GEMM. Equation 1 describes SR with dithering for a uniform integer quantizer; the non-uniform case requires modifying the noise scale but is otherwise essentially the same.

$$\delta \sim \mathcal{U}(-0.5, 0.5) \tag{1}$$

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

SR can also be used anywhere where numbers are quantized. For example, near the end of training, the model update norm is much smaller than the parameter norm and information in low precision updates can be"lost" (Yu et al., 2024). Here, stochastic rounding can be used to preserve the update *in expectation*, which uses less memory than keeping a high precision copy of the parameters.

2.5 Microscaling (MX) FP Formats

The recently introduced microscaling floating point family of datatypes builds upon IEEE floats by adding a groupwise scale to a base IEEE float (Project, 2023). This scale allows a MXFP tensor to take on a wider range of values without significantly increasing the total bitrate, with the caveat that entries in a group should be roughly the same magnitude for the scale to be useful. In practice, MX scaling is more important as the base datatype bitrate decreases. Whereas FP8 E4M3 has a dynamic range of $\frac{448}{2^{-9}} = 2.3 \times 10^6$, FP4 has a dynamic range of $\frac{6}{0.5} = 12$. MX scaling enables MXFP4 to represent a much wider range of values *across blocks*.

The core hardware-supported MXFP formats generally follow similar patterns. Scales are shared across contiguous entries in memory (usually 32), and quantizing a scalar tensor to a MX tensor depends on the largest element in each group (Project, 2023; NVIDIA, a). Algorithm 1 describes the "reference" algorithm for quantizing a scalar tensor to MX, which can be implemented efficiently on modern AI accelerators (Thakkar et al., 2023). Algorithm 1 scales each group based on its maximum magnitude element and then performs nearest rounding to obtain a MX tensor.

Algorithm 1 Convert vector of scalar floats $V \in$ `HP_DTYPE`^{k} to an MX block $\{X, P \in$ `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** $X, \{P_i\}_{i=1}^k$

Algorithm 2 Unbiased quantization of $V \in$ `HP_DTYPE`^{k} to an MXFP4 block $\{X, P \in$ `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** $X, \{P_i\}_{i=1}^k$

3 Training with MXFP4

The rest of this paper describes our approach that enables near-lossless *training* with MXFP4-accelerated GEMMs. Although our paper focuses on MXFP4, our analysis also applies to other low precision datatypes such as MXINT4. We chose MXFP4 due to its relevance and hardware support on the latest accelerators. To the best of our knowledge, MXFP4 has only been successfully used for near-lossless inference (Rouhani et al., 2023; NVIDIA, 2024b). Although certain works have achieved near-lossless training with MXFP4 weights, these require the activations and gradients to kept in higher precision. These recipes run at the throughput of the higher precision operand, making them slower than pure-FP4 recipes.

Our method hinges on obtaining unbiased, low-variance gradient estimates with pure-MXFP4 GEMMs in the backward pass, enabling more accurate model updates. Since the backward pass consists of $> 1/2$ training FLOPs, our recipe can significantly accelerate training without reducing the representational power of the model from LP forward passes (Kumar et al., 2025). To do this, we first modify the OCP MX quantization algorithm to perform unbiased quantization with scaling and stochastic rounding. Then, we show that by first transforming the GEMM operands with a memory-bound construction of the random Hadamard transform (RHT) *before quantization*, we can bound the variance of the GEMM output. Our method adds minimal overhead while significantly improving the quality of trained models, making MXFP4 practical for training.

3.1 Unbiased Quantization to MXFP4

Algorithm 1 describes the "reference" MX quantization algorithm to convert a scalar matrix to an MX matrix. Algorithm 1 finds, for each group of 32 entries, value with the largest magnitude $m = \max_i(|V_i|)$. Then, it calculates a shared exponent as a function of m and $\texttt{emax}_{\texttt{elem}}$, the largest exponent of a normal number in the base data format. For example, $\texttt{emax}_{\texttt{elem}} = 2$ for FP4 since its maximum normal value is $6 = 2^2 * 1.5$.

Finally, group elements are normalized by the shared exponent and rounded to the base datatype.

For MXFP4, line 1 of Algorithm 1 returns $\texttt{shared_exp} \leftarrow \lfloor \log_2(m) \rfloor - 2$. Observe that after dividing the entire group by $2^{\texttt{shared_exp}}$, m becomes

$$m \leftarrow \frac{m}{2^{\texttt{shared_exp}}} < \frac{m}{2^{\log_2(m)-3}} = 8 \tag{3}$$

Since the maximum representable normal value in FP4 is 6, values scaled to between 6 and 8 will get clipped, making Algorithm 1 inherently biased. Although the proportion clipped depends on the input matrix, we can empirically check that for a wide distribution of matrices, roughly 3% of the entries will get clipped.

We can make Algorithm 1 unbiased with two simple modifications, both of which can be efficiently implemented in hardware. First, we scale V_i/X by 3/4 to prevent clipping. Then, we use stochastic rounding to quantize Q' to FP4, which gives an unbiased estimate of Q'. Algorithm 2 summarizes these modifications. The resulting MX matrix is an unbiased estimate of 3/4 the original matrix. Since SR is implemented with uniform independent dithering in hardware, the resulting GEMM output is an unbiased estimator of $(3/4)^2 = 9/16$ of the correct output. To get an unbiased output, we can simply scale the high precision accumulator output by 16/9.

Lemma 3.1. *Assume stochastic rounding is implemented with dithering with independent noise. Then, Algorithm 2 produces a MXFP4 matrix that is an unbiased estimate of* 3/4 *its input. Furthermore, Algorithm 3 with Algorithm 2 as a subroutine produces an unbiased estimate of* $\frac{dL}{dx}$ *and* $\frac{dL}{dW}$.

3.2 Bounding the Variance of SR with the Random Hadamard Transform

The backward pass for a linear layer ($y = xW^T$) requires computing $\frac{dL}{dx} = \frac{dL}{dy} W$ and $\frac{dL}{dW} = \frac{dL}{dy}^T x$. LLMs

have been known to have activation (x) and weight (W) "outliers" as well as sparse gradients ($\frac{dL}{dy}$) (Xi et al., 2023; Tseng et al., 2024a). Recall that MXFP4 quantization relies on groupwise statistics such as the largest magnitude element, so blocks with outliers will suffer from high quantization distortion and stochastic rounding variance.

Although Lemma 3.1 tells us that Algorithm 2 produces an unbiased estimate of the true GEMM, high variance estimates can still degrade model quality by effectively adding noise to the gradient estimate. To remedy this, we use the randomized Hadamard transform to concentrate gradients, activations, and weights before quantization, which asymptotically reduces the variance of the GEMM output.

The random Hadamard transform performs $x \leftarrow HSx$, where $x \in \mathbb{R}^{j\times k}, S \in \{\pm 1\}^k$ (a random sign vector), and H is the k-dimensional Hadamard matrix (Halko et al., 2011). Hadamard matrices are recursively defined orthogonal matrices that satisfy the following:

$$H_n = \frac{1}{2^{n/2}} \begin{bmatrix} H_{n-1} & H_{n-1} \\ H_{n-1} & -H_{n-1} \end{bmatrix}, \quad (4)$$

where $H_1 = \begin{bmatrix} 1 \end{bmatrix}$. Since both H and $diag(S)$ are orthogonal, the RHT is fully invertible. This means that we can apply the RHT to GEMM operands without inverting the RHT – that is, $(HSA)^T(HSB) = A^TB$.

Theorem 3.2. *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 $\mathcal{O}(b\Delta^4\|A\|_\infty\|B\|_\infty)$ and the variance of $\mathcal{Q}(HSA)^T\mathcal{Q}(HSB)$ is, with probability $\geq (1-\epsilon)^2$, $\mathcal{O}(\Delta^4\|A\|\|B\|\log(2b/\epsilon))$, where the largest gap between two consecutive representable points in $\mathcal{Q}$'s quantizer is Δ.*

Theorem 3.2 tells us that the variance of a MX matrix multiplication with respect to stochastic rounding is linear in the product of the largest magnitude elements in the operands. Applying the RHT to a vector effectively concentrates it to have a sub-Gaussian tail distribution. From Tseng et al. (2024a), we know that

$$\mathbb{P}\left(|e_i HSx| \geq a\right) \leq 2\exp\left(\frac{-a^2k}{2\|x\|^2}\right), \quad (5)$$

letting us bound the the variance of the SR GEMM in Theorem 3.2. Specifically, applying the RHT reduces the variance from a linear dependence on blocksize to a log-dependence on blocksize, albeit with the L_2 norm of the input instead of the L_∞ norm.

We can verify this empirically by measuring the variance of a SR GEMM with and without the RHT. Figure 2 shows the mean variance of $\mathcal{Q}(A)^T\mathcal{Q}(B)$ vs. $\mathcal{Q}(HSA)^T\mathcal{Q}(HSB)$ over 4K samples of $A, B \in \mathbb{R}^b \sim \mathcal{N}(0, I)$ with proportion p outliers from $\mathcal{N}(0, 5I)$, where $\mathcal{Q}$ performs Algorithm 2. That is, $A, B \sim \mathcal{N}(0, I) + \text{Bernoulli}(p) * \mathcal{N}(0, 5I)$. As expected from Theorem 3.2, the variance grows much slower as a function of b with the RHT vs. without.

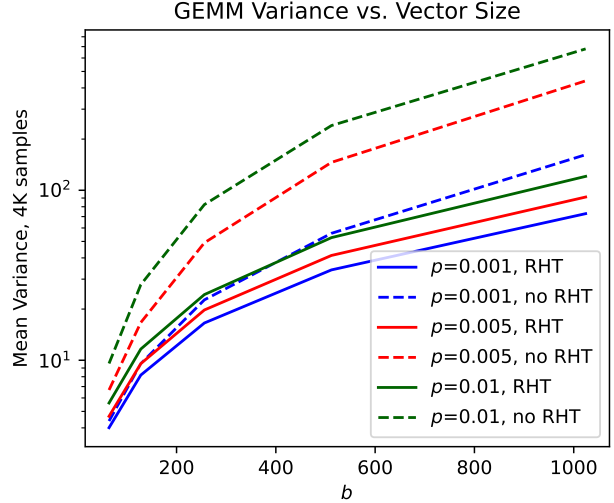

Figure 2: Mean variance of $\mathcal{Q}(A)^T\mathcal{Q}(B)$ vs. $\mathcal{Q}(HSA)^T\mathcal{Q}(HSB)$ over 4K samples of $A, B \in \mathbb{R}^b \sim \mathcal{N}(0, I)$ with proportion p outliers from $\mathcal{N}(0, 5I)$. $\mathcal{Q}$ performs Algorithm 2. Variance with the RHT grows much slower than without.

However, the RHT is not free. First, observe that when computing $\frac{dL}{dW} \approx \mathcal{Q}(HS\frac{dL}{dy})^T\mathcal{Q}(HSx)$, the RHT "mixes" along the batch dimension. In data-parallel settings (e.g. FSDP (Zhao et al., 2023) or ZeRO-3 (Rajbhandari et al., 2020)) where activations are sharded across GPUs, the full RHT would require expensive cross-GPU communication. Even with fast interconnects, this would immediately bottleneck gradient computation. Second, although Equation 4 admits an $O(n \log n)$ time matrix-vector product algorithm, the RHT step occurs in high precision. Reducing this overhead is critical – if the RHT is slower than a FP4 matmul, one should just use FP8 instead.

To solve these problems, we apply the RHT as a dense matrix multiplication over a small number of MX blocks, which makes it *memory bound* in the GEMM operands (see Table 5). Specifically, let the RHT block size be $g, 32|g$. Applying this block-wise RHT as a dense matmul gives a runtime of $O((b+m)ng)$ and IO cost of $O(bn+nm+bm)$. Since modern AI accelerators have high compute to memory ratios, this "blockwise" RHT is memory bound when $g \lessapprox 256$. Algorithm 3 summarizes how we use the RHT in the backward pass of a linear layer. Since g is smaller than

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: $G' \leftarrow \left(\left(\frac{dL}{dy}\right).\mathtt{view}\left(\frac{bm}{g}, g\right)\right)\mathtt{diag}(S)H$
4: $W' \leftarrow H^T\mathtt{diag}(S)\left(W.\mathtt{view}\left(g, \frac{nm}{g}\right)\right)$
5: $GT' \leftarrow \left(\left(\frac{dL}{dy}^T\right).\mathtt{view}\left(\frac{bm}{g}, g\right)\right)\mathtt{diag}(S)H$
6: $X' \leftarrow H^T\mathtt{diag}(S)\left(x.\mathtt{view}\left(\frac{bn}{g}, g\right)\right)$
7: $\frac{dL}{dx} \leftarrow \mathtt{MXFP4_GEMM}(G', W')$
8: $\frac{dL}{dW} \leftarrow \mathtt{MXFP4_GEMM}(GT', X')$
{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: $\frac{dL}{dx} \leftarrow \frac{16}{9}\frac{dL}{dx}$
11: $\frac{dL}{dW} \leftarrow \frac{16}{9}\frac{dL}{dW}$
12: **end if**
13: **return** $\frac{dL}{dx}, \frac{dL}{dW}$

the sequence length of any reasonably large model, Algorithm 3 works as a drop-in replacement for a linear layer even in data-parallel settings. Furthermore, although lines 3-6 are written out for clarity, an efficient implementation could fuse them into lines 7 and 8, reducing costly memory accesses.

The tradeoff to doing this blockwise RHT is that equation 5 depends on g (k in the equation) – the higher g is, the tighter the concentration will be. However, in practice, we observe $g = 64$ is sufficient to get a tight distribution and MX can handle scale differences across blocks. Finally, note that this construction also lets us use *any* random orthogonal transformation. We chose the RHT since it is fast to randomize (by sampling a single g-dim sign vector) and has good concentration, but other matrices could work as well.

4 Experiments

Our main experiments focus on pretraining GPT 345M, 1.3B, and 6.7B (Brown et al., 2020). We follow prior low precision training works and train for at least 20 billion tokens, which is sufficient to determine overall training performance on a longer full-scale run (Peng et al., 2023). We use the Megatron-LM codebase to train our models (Shoeybi et al., 2020), the publicly available GPT2 Wikipedia dataset (Neuron), and the bit-accurate Microsoft `microxcaling` library for MX

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

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 Tulu V2	25.6	50.6	62.7	59.6	51.6
MXFP4⋆ Tulu V2	25.9	49.9	62.9	60.5	51.8

Table 3: GPT 6.7B model trained on 20B tokens before and after Tulu V2 fine-tuning. Both BF16 and our MXFP4+RHT+SR (MXFP4⋆) model exhibit similar performance before and after fine-tuning.

emulation (Microsoft, 2024). Since pretraining is expensive, we stopped certain experiments short when it was clear they did not match BF16. Our analysis below uses validation perplexity from a holdout set, but we observe the same behavior with training perplexity. Training perplexity plots and additional experiments can be found in the Appendix.

4.1 GPT Pretraining Results

Table 2 and Figure 6 show our main results with using BF16 in the forward pass and various MXFP4 constructions in the backward pass. We ablate on using the RHT only, which produces a biased but reduced-distortion GEMM, and RHT and SR, which produces an unbiased, lower varianace GEMM. For GPT 1.3B, we also measure the performance of MXFP4+SR only, which gives an unbiased but higher variance GEMM. All experiments use Megatron-LM's mixed precision

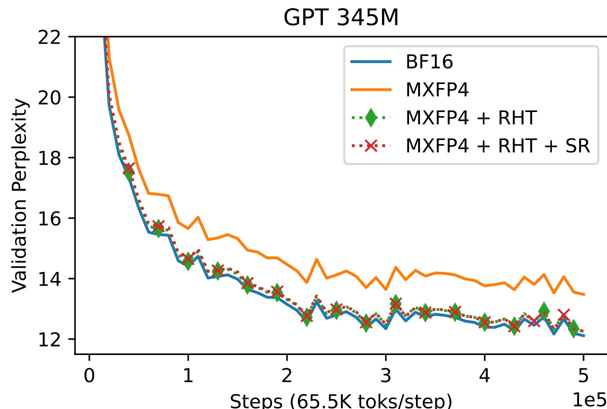

Figure 3: GPT 345M validation perplexity curves with BF16 forward pass. With RHT and SR, MXFP4 can match the performance of BF16 in the backward pass.

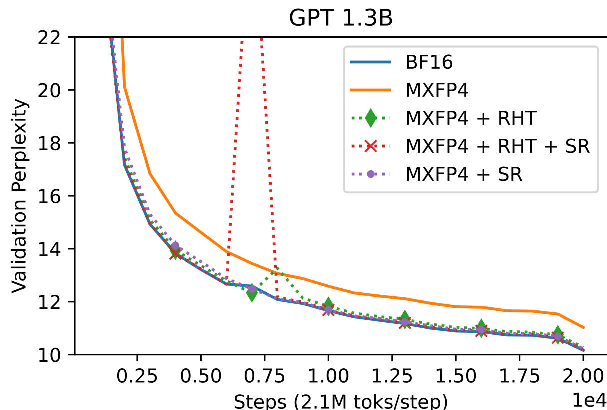

Figure 4: GPT 1.3B validation perplexity curves with BF16 forward pass. With RHT and SR, MXFP4 can match the performance of BF16 in the backward pass.

implementation with separate FP32 master weights and BF16 parameter copies. For the backward pass for decoder linear layers, the BF16 and gradients are quantized to MXFP4. Experiments with the RHT use $g = 64$, which mixes across 2 MX blocks.

Table 2 shows that for shorter runs (20-40 billion tokens), using either the RHT or SR with MXFP4 is sufficient to achieve near-lossless training all tested model sizes. However, Figure 6 shows that for longer runs (210 billion tokens), having an unbiased gradient estimator is necessary to maintain performance. Whereas using the RHT only results in an ≈ 0.1 perplexity gap, using stochastic rounding (with or without the RHT) results in *no validation perplexity gap*.

Figures 3, 4 and 5 show the validation perplexity curves for the experiments in Table 2. At all scales, the MXFP4+RHT+SR curve closely tracks

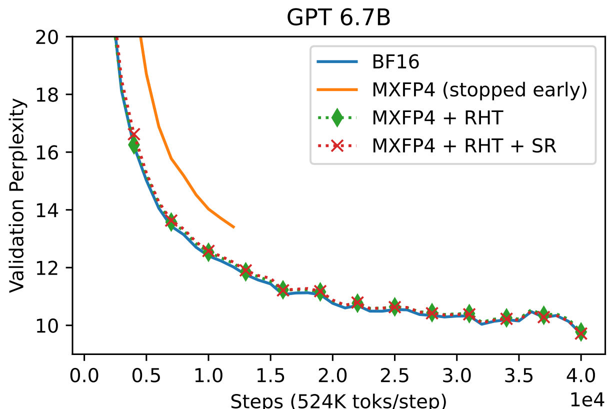

Figure 5: GPT 6.7B validation perplexity curves with BF16 forward pass. With RHT and SR, MXFP4 can match the performance of BF16 in the backward pass. The MXFP4-only run was stopped early to save resources.

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.

BF16. In contrast, although the final performance of MXFP4+SR matches MXFP4+SR+RHT, MXFP4+SR exhibits slower initial convergence than BF16 and MXFP4+RHT+SR. We suspect that this is due to loss of gradient information without the RHT. Although using only SR will give an unbiased gradient estimator, small values will still get stochastically flushed to 0 (Equation 1), resulting in loss of gradient information. In contrast, the RHT transforms the gradient to a different space. This reduces variance and also significantly reduces the probability that a single gradient entry in the original space will be set to 0. To verify this, Table 4 shows an ablation on the RHT block size – increasing the block size improves quality.

These figures also include curves for using pure MXFP4 (no RHT and no SR) MP in the backward pass. Using only MXFP4 (the orange curve) results in significant degradation and a large perplexity gap at all sizes. Even further, if we consider that FP8 is near-lossless (Peng et al., 2023) vs. BF16 and is only an estimated 30-40% slower end-to-end than pure MXFP4, then pure MXFP4 isn't even "worth it." For a fixed amount of wall clock time, simply training with FP8 for fewer steps would give a better model than using pure MXFP4. In contrast, our techniques close the

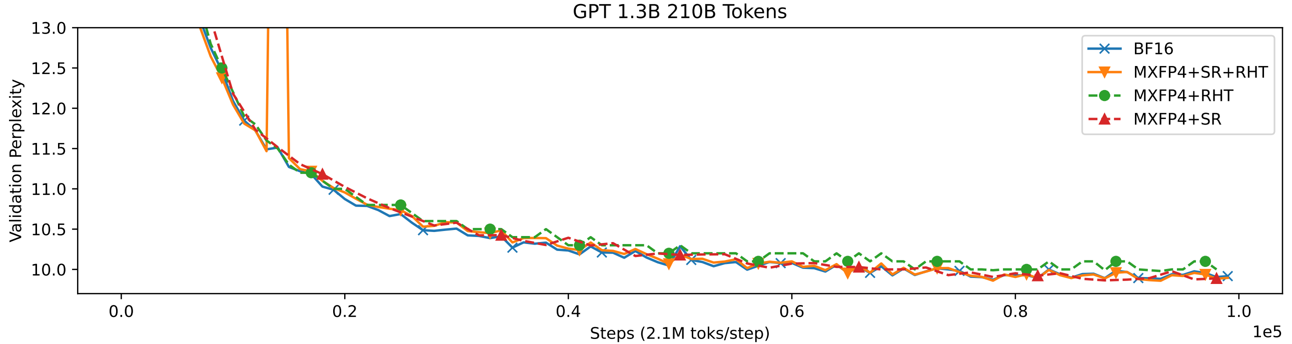

Figure 6: Validation perplexity for training GPT 1.3B for 210 billion tokens, or 5× longer than in Table 2. All experiments used BF16 in the forward pass and the specified backward precision.

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.

gap to BF16 and FP8, making MXFP4 practical for training. Our techniques are also compatible with FP8 forward passes (Figure 7, more details in Appendix), further pushing the speed-quality tradeoff curve.

To further evaluate our MXFP4 models, we ran zeroshot evaluation for downstream tasks on our 20B token GPT 6.7B models. Both the BF16 and MXFP4+RHT+SR models perform around the same. To test how well these models can be fine-tuned, we fine-tuned them using the publicly-available Tulu V2 dataset (657M tokens) and codebase (Ivison et al., 2023). We used the hyperparameters in the Tulu V2 codebase and trained for 5 epochs with BF16/FP32 mixed precision. The BF16 model reached a final training perplexity of 1.96, and the MXFP4+RHT+SR model 1.98. Like before finetuning, both models achieve similar zeroshot performance, indicating that they are of similar quality. Table 3 summarizes these results.

4.2 Overhead Calculations

The goal of MXFP4 training is to achieve a wall-clock time speedup over FP8 training. Unfortunately, we do not have access to FP4 hardware yet so we cannot measure empirical wall-clock speedups over FP8. However, we can estimate the overhead of the RHT and stochastic rounding with proxy benchmarks.

Our RHT construction operates on a small "tile" in the operand and is memory bound, so we can conceivably fuse it with the MXFP4 GEMM and avoid writing its output to memory. We can estimate the performance of this setup in two ways. First, we measured the overhead of RHT-GEMM kernels for *FP8*. Specifically, we timed A`.to(E4M3)`B`.to(E4M3)`^{T}, $A \in \texttt{BF16}^{n\times k}, B \in \texttt{BF16}^{m\times k}$ with and without the RHT along the k dimension. We generated Triton (Tillet et al., 2019) kernels with `torch.compile` (Team), an RHT size of $g = 64$, and benchmarked 7B and 70B-sized matrices: $(m, n, k) = (32768, 8192, 8192)$ and $(16384, 28672, 28672)$. On a NVIDIA H100 GPU, the RHT adds 9.7% overhead for the 7B-sized setup and 1.6% for the 70B-sized setup. Assuming MXFP4 has twice the throughput of FP8, these numbers would double to 19.4% and 3.2%, respectively, which is still faster than a FP8 GEMM.

Second, we measured the overhead of the RHT on the HuggingFace implementation (Wolf et al., 2020) of single Llama 2 70B decoder layer decoder layer on a NVIDIA A100 GPU. Specifically, we report the end-to-end tokens per second for computing the forward pass in FP16 and backward pass in either FP16 or INT4, which has the same hardware speedup (4×) on the NVIDIA A100 vs. FP16 as MXFP4 has on modern hardware. We also include INT8 as a proxy for the expected speedup of a FP8 backward pass. We use a batch size of 4 sequences with 4K tokens each (16K tokens/batch), Flash Attention 2, `torch.compile`, and

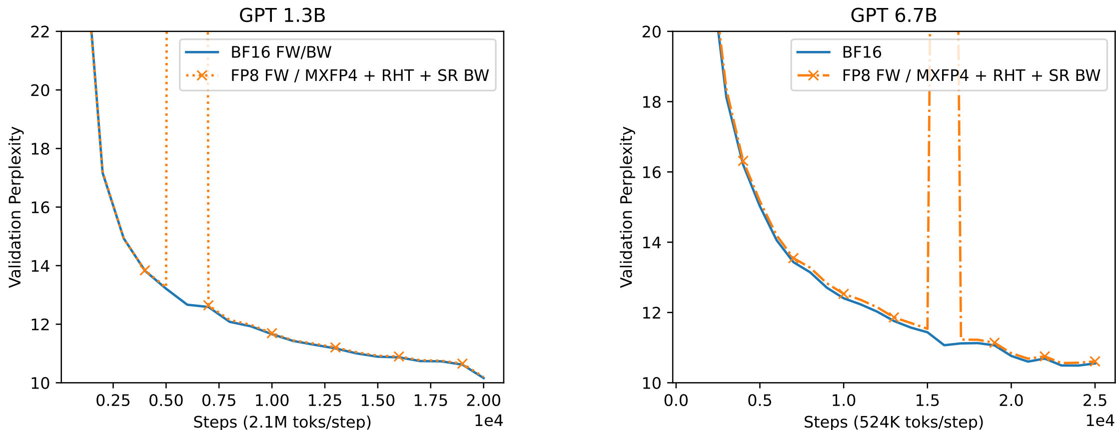

Figure 7: GPT 1.3B and 6.7B perplexity curves with a FP8 forward pass, our MXFP4 backward pass, and the same settings as Figures 4 and 5. Our method is compatible with FP8 forward passes for additional speedups. See Appendix for details.

the CUTLASS INT4 and INT8 GEMM kernels. We were unable to use CUDA graphs since the HuggingFace implementation is not compatible with CUDA graphs; we expect CUDA graphs to improve speedup ratios by masking kernel launch overhead.

Table 5 summarizes these results. End to end with a FP16 FW pass, an INT4+RHT backward pass is over 40% faster than a FP16 backward pass and over 20% faster than an INT8 backward pass. If we only consider the backward pass, INT4+RHT is $\approx 70\%$ faster than a FP16 backward pass and $\approx 30\%$ faster than a INT8 backward pass. The HuggingFace Llama implementation is not known to be fast, so a more efficient implementation would achieve better INT4 and INT8 speedups over FP16. Table 5 also shows that the RHT adds less than 5% E2E overhead and is memory bound in the operands until $g \approx 256$. Interestingly, the recently released $\mathcal{O}(n \log n)$ HadaCore kernel (Agarwal et al., 2024) recovers most of the dense GEMM penalty at $g = 1024$, but is still slower than smaller g.

To measure the overhead of stochastic rounding, we used an Amazon Trainium 1 chip (EC2 *Trn1* instance), which is one of the few widely available chips that has dedicated stochastic rounding hardware (Amazon). Our experiments show that for most matrix sizes, using SR to quantize GEMM operands from FP32 to BF16 adds less than 2% overhead over the BF16 GEMM itself. Assuming a $4\times$ increase in GEMM throughput when going from BF16 to FP4, this would mean SR adds less than 10% overhead.

5 Conclusion

While hardware support for low precision datatypes continues to advance, it is becoming increasingly difficult to train with these datatypes without suffering from significant model degradation. In this work, we demonstrate the first MXFP4 training recipe that achieves near-lossless model quality vs. FP32/BF16 mixed precision training. Our method hinges on computing low variance, unbiased gradient estimates for decoder linear layer, which enables us to make more accurate model updates. To do this, we propose using stochastic rounding (SR) and the random Hadamard transform (RHT). Stochastic rounding produces unbiased gradient estimates, and the RHT reduces the variance of SR and the chance of losing gradient information from underflow. Our experiments pretraining GPT models up to 6.7B show that both the RHT and SR are crucial for near-lossless MXFP4 training. Finally, our benchmarks show that our method can be implemented with minimal overhead, giving an estimated 30% speedup over FP8 and 70% speedup over BF16 in the backward pass.

Acknowledgements

We thank Chris De Sa for valuable feedback. We also thank Yida Wang and George Karypis for their support within AWS AI Research.

References

Krish Agarwal, Rishi Astra, Adnan Hoque, Mudhakar Srivatsa, Raghu Ganti, Less Wright, and Sijia Chen. Hadacore: Tensor core accelerated hadamard transform kernel, 2024. URL https://arxiv.org/abs/2412.08832.

Amazon. Trainium architecture. URL https://awsdocs-neuron.readthedocs-hosted.com/en/latest/general/arch/neuron-hardware/trainium.html.

Charlie Blake, Douglas Orr, and Carlo Luschi. Unit scaling: out-of-the-box low-precision training. In *Proceedings of the 40th International Conference on Machine Learning*, ICML'23. JMLR.org, 2023.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020. URL https://arxiv.org/abs/2005.14165.

Adam Casson. Transformer flops, 2023. URL https://www.adamcasson.com/posts/transformer-flops.

Matteo Croci, Massimiliano Fasi, Nicholas J. Higham, Theo Mary, and Mantas Mikaitis. Stochastic rounding: implementation, error analysis and applications. *Royal Society Open Science*, 9(3), March 2022. ISSN 2054-5703. doi: 10.1098/rsos.211631. URL http://dx.doi.org/10.1098/rsos.211631.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

NVIDIA Transformer Engine. Transformer engine 1.11.0. URL https://docs.nvidia.com/deeplearning/transformer-engine/user-guide/index.html.

N. Halko, P. G. Martinsson, and J. A. Tropp. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. *SIAM Review*, 53(2):217–288, 2011. doi: 10.1137/090771806. URL https://doi.org/10.1137/090771806.

IEEE. Ieee standard for floating-point arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pages 1–84, 2019. doi: 10.1109/IEEESTD.2019.8766229.

Hamish Ivison, Yizhong Wang, Valentina Pyatkin, Nathan Lambert, Matthew Peters, Pradeep Dasigi, Joel Jang, David Wadden, Noah A. Smith, Iz Beltagy, and Hannaneh Hajishirzi. Camels in a changing climate: Enhancing lm adaptation with tulu 2, 2023. URL https://arxiv.org/abs/2311.10702.

Justin Johnson, Apr 2017. URL https://cs231n.stanford.edu/handouts/linear-backprop.pdf.

Tanishq Kumar, Zachary Ankner, Benjamin Frederick Spector, Blake Bordelon, Niklas Muennighoff, Mansheej Paul, Cengiz Pehlevan, Christopher Re, and Aditi Raghunathan. Scaling laws for precision. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=wg1PCg3CUP.

Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. Mixed precision training. In *International Conference on Learning Representations*, 2018. URL https://openreview.net/forum?id=r1gs9JgRZ.

Microsoft. Mx pytorch emulation library, 2024. URL https://github.com/microsoft/microxcaling.

AWS Neuron. Gpt wikiedia dataset. URL https://github.com/aws-neuron/aws-neuron-parallelcluster-samples/blob/master/examples/jobs/neuronx-nemo-megatron-gpt-job.md.

NVIDIA. Ptx isa 8.7, a. URL https://docs.nvidia.com/cuda/pdf/ptx_isa_8.7.pdf.

NVIDIA. Transformer engine, b. URL https://github.com/NVIDIA/TransformerEngine.

NVIDIA. Nvidia blackwell architecture technical brief, 2024a. URL https://resources.nvidia.com/en-us-blackwell-architecture.

NVIDIA. Nvidia blackwell platform sets new llm inference records in mlperf inference v4.1, Sep 2024b. URL https://developer.nvidia.com/blog/nvidia-blackwell-platform-sets-new-llm-inference-records-in-mlperf-inference-v4-1/.

Houwen Peng, Kan Wu, Yixuan Wei, Guoshuai Zhao, Yuxiang Yang, Ze Liu, Yifan Xiong, Ziyue Yang, Bolin Ni, Jingcheng Hu, Ruihang Li, Miaosen Zhang, Chen Li, Jia Ning, Ruizhe Wang, Zheng Zhang,

Shuguang Liu, Joe Chau, Han Hu, and Peng Cheng. Fp8-lm: Training fp8 large language models, 2023. URL `https://arxiv.org/abs/2310.18313`.

Open Compute Project, 2023. URL `https://www.opencompute.org/documents/ocp-microscaling-formats-mx-v1-0-spec-final-pdf`.

Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models, 2020. URL `https://arxiv.org/abs/1910.02054`.

Bita Darvish Rouhani, Ritchie Zhao, Ankit More, Mathew Hall, Alireza Khodamoradi, Summer Deng, Dhruv Choudhary, Marius Cornea, Eric Dellinger, Kristof Denolf, Stosic Dusan, Venmugil Elango, Maximilian Golub, Alexander Heinecke, Phil James-Roxby, Dharmesh Jani, Gaurav Kolhe, Martin Langhammer, Ada Li, Levi Melnick, Maral Mesmakhosroshahi, Andres Rodriguez, Michael Schulte, Rasoul Shafipour, Lei Shao, Michael Siu, Pradeep Dubey, Paulius Micikevicius, Maxim Naumov, Colin Verrilli, Ralph Wittig, Doug Burger, and Eric Chung. Microscaling data formats for deep learning, 2023. URL `https://arxiv.org/abs/2310.10537`.

Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism, 2020. URL `https://arxiv.org/abs/1909.08053`.

PyTorch Team. Pytorch. URL `https://pytorch.org`.

Vijay Thakkar, Pradeep Ramani, Cris Cecka, Aniket Shivam, Honghao Lu, Ethan Yan, Jack Kosaian, Mark Hoemmen, Haicheng Wu, Andrew Kerr, Matt Nicely, Duane Merrill, Dustyn Blasig, Fengqi Qiao, Piotr Majcher, Paul Springer, Markus Hohnerbach, Jin Wang, and Manish Gupta. CUTLASS, January 2023. URL `https://github.com/NVIDIA/cutlass`.

Philippe Tillet, H. T. Kung, and David Cox. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2019, page 10–19, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367196. doi: 10.1145/3315508.3329973. URL `https://doi.org/10.1145/3315508.3329973`.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

Albert Tseng, Jerry Chee, Qingyao Sun, Volodymyr Kuleshov, and Christopher De Sa. QuIP#: Even better LLM quantization with hadamard incoherence and lattice codebooks. In Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp, editors, *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 48630–48656. PMLR, 21–27 Jul 2024a. URL `https://proceedings.mlr.press/v235/tseng24a.html`.

Albert Tseng, Qingyao Sun, David Hou, and Christopher De Sa. Qtip: Quantization with trellises and incoherence processing. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024b.

Ruizhe Wang, Yeyun Gong, Xiao Liu, Guoshuai Zhao, Ziyue Yang, Baining Guo, Zhengjun Zha, and Peng Cheng. Optimizing large language model training using fp4 quantization, 2025. URL `https://arxiv.org/abs/2501.17116`.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Huggingface's transformers: State-of-the-art natural language processing, 2020. URL `https://arxiv.org/abs/1910.03771`.

Haocheng Xi, ChangHao Li, Jianfei Chen, and Jun Zhu. Training transformers with 4-bit integers. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL `https://openreview.net/forum?id=H9hWlfMT6O`.

Tao Yu, Gaurav Gupta, Karthick Gopalswamy, Amith Mamidala, Hao Zhou, Jeffrey Huynh, Youngsuk Park, Ron Diamant, Anoop Deoras, and Luke Huan. Collage: Light-weight low-precision strategy for llm training. In *Proceedings of the 41st International Conference on Machine Learning (ICML 2024)*. PMLR, 2024.

Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, Alban Desmaison, Can Balioglu, Pritam Damania, Bernard

Nguyen, Geeta Chauhan, Yuchen Hao, Ajit Mathews, and Shen Li. Pytorch fsdp: Experiences on scaling fully sharded data parallel, 2023. URL `https://arxiv.org/abs/2304.11277`.

Checklist

1. For all models and algorithms presented, check if you include:
   (a) A clear description of the mathematical setting, assumptions, algorithm, and/or model. [**Yes**]
   (b) An analysis of the properties and complexity (time, space, sample size) of any algorithm. [**Yes**]
   (c) (Optional) Anonymized source code, with specification of all dependencies, including external libraries. [**No**]

2. For any theoretical claim, check if you include:
   (a) Statements of the full set of assumptions of all theoretical results. [**Yes**]
   (b) Complete proofs of all theoretical results. [**Yes**]
   (c) Clear explanations of any assumptions. [**Yes**]

3. For all figures and tables that present empirical results, check if you include:
   (a) The code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL). [**Yes**]
   (b) All the training details (e.g., data splits, hyperparameters, how they were chosen). [**Yes**]
   (c) A clear definition of the specific measure or statistics and error bars (e.g., with respect to the random seed after running experiments multiple times). [**Not Applicable**]
   (d) A description of the computing infrastructure used. (e.g., type of GPUs, internal cluster, or cloud provider). [**Yes**]

4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets, check if you include:
   (a) Citations of the creator If your work uses existing assets. [**Yes**]
   (b) The license information of the assets, if applicable. [**Not Applicable**]
   (c) New assets either in the supplemental material or as a URL, if applicable. [**Not Applicable**]
   (d) Information about consent from data providers/curators. [**Not Applicable**]
   (e) Discussion of sensible content if applicable, e.g., personally identifiable information or offensive content. [**Not Applicable**]

5. If you used crowdsourcing or conducted research with human subjects, check if you include:
   (a) The full text of instructions given to participants and screenshots. [**Not Applicable**]
   (b) Descriptions of potential participant risks, with links to Institutional Review Board (IRB) approvals if applicable. [**Not Applicable**]
   (c) The estimated hourly wage paid to participants and the total amount spent on participant compensation. [**Not Applicable**]

Training LLMs with MXFP4: Supplementary Materials

6 Additional Results

6.1 FP8 Forward Pass Results

This section contains experiments using mixed-precision FP8 in the forward pass and MXFP4 in the backward pass. Prior works have shown that mixed-precision FP8 forward and backward passes can be close to lossless over mixed-precision BF16 training (Peng et al., 2023; NVIDIA, b). To test if MXFP4 backward passes are still practical with FP8 forward passes, we trained GPT 1.3B and 6.7B models with NVIDIA's TransformerEngine (TE) FP8 (E4M3) implementation (NVIDIA, b) in the forward pass and our MXFP4 formulation in the backward pass. We did not test GPT 345M since TE FP8 already had a > 0.1 validation perplexity gap vs. BF16 in our experiments. For GPT 6.7B, since we did not have access to FP8-capable hardware with fast interconnects necessary for tensor-parallel training, we emulated FP8 matrix multiplications by dequantizing FP8 GEMM operands into BF16 and performing a BF16 GEMM. While this is not bit-accurate vs. a FP8 GEMM, the relative error of the output is $\approx 0.3\%$ for random Gaussian inputs. Furthermore, this is essentially how PyTorch emulates FP8 GEMMs (Team). At both model scales, we find that FP8 forward passes and MXFP4 backward passes are sufficient to essentially match BF16 training.

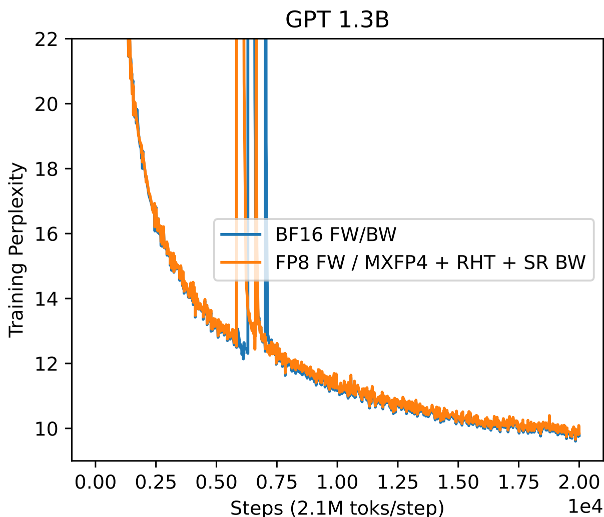

Figure 8: Training perplexity curves for GPT 1.3B for 33 billion tokens. Using FP8 in the forward pass and MXFP4 in the backward pass does not result in noticeable degradation.

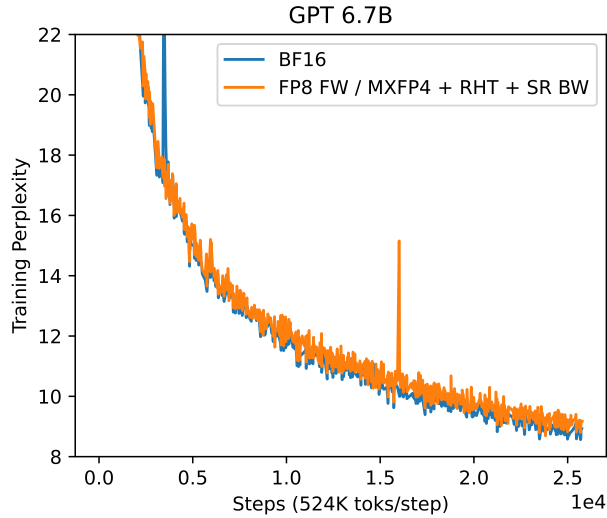

Figure 9: Training perplexity curves for GPT 6.7B for the first 13 billion tokens of a 20 billion token run. Due to time constraints and the cost of training a 6.7B parameter model, we were unable to include the full run. Like 1.3B, using FP8 in the forward pass and MXFP4 in the backward pass does not result in noticeable degradation.

6.2 GPT 345M Validation Curves with Stochastic Rounding Only

This plot is the same as those from the main body except that it includes an experiment with stochastic rounding only (no RHT). Like 1.3B, SR starts off "worse" than the RHT variants but is able to match their performance at the end of the training run.

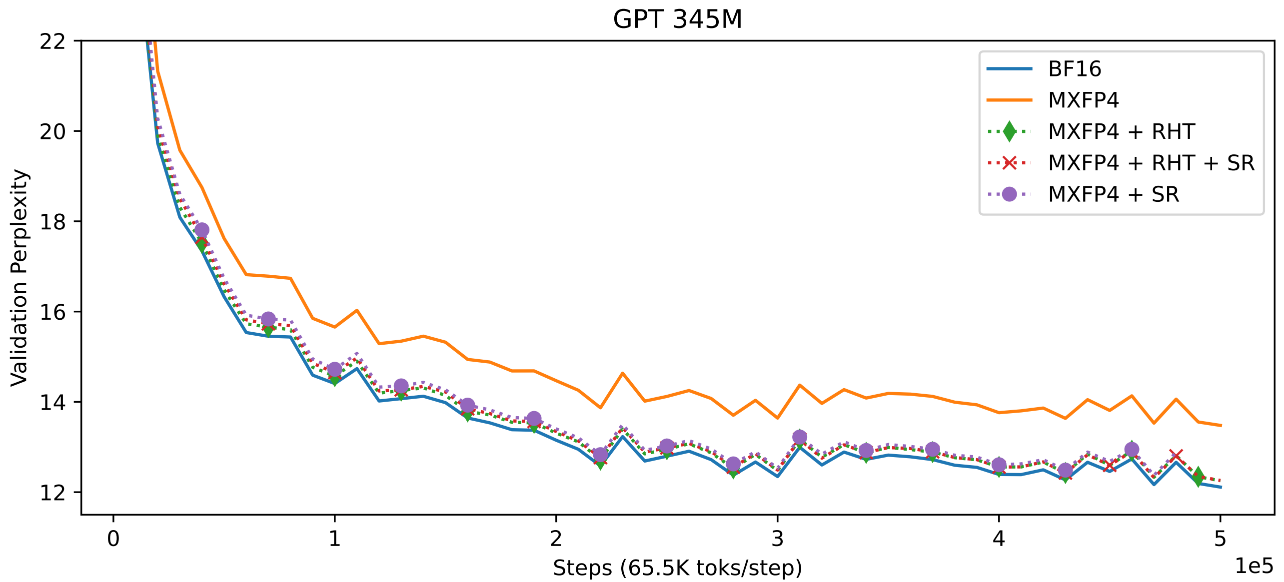

Figure 10: Validation perplexity for training GPT 345M for 33 billion tokens. All experiments used BF16 in the forward pass and the specified backward precision. This plot is the same as Figure 3 in the main body except that it adds an experiment with MXFP4+SR only.

6.3 Training Curve for GPT 1.3B

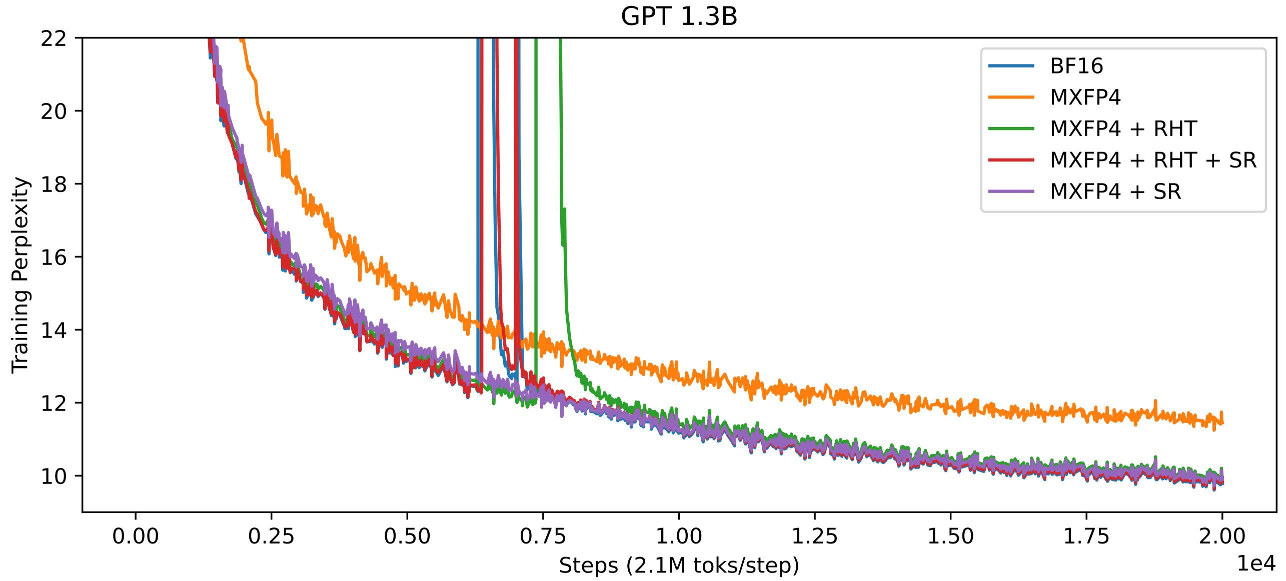

Figure 11: Training perplexity for training GPT 1.3B for 40 billion tokens. All curves used BF16 in the forward pass and the specified backward precision.

6.4 Training Perplexity for GPT 1.3B on 200 Billion Tokens

This section contains the full 210B token GPT 1.3B run referenced in Section 4 of the main body. There is an approximately 0.1 validation perplexity gap between MXFP4+RHT only (10.02 ppl) and BF16 (9.92 ppl), whereas MXFP4+RHT+SR matches BF16 (9.90 ppl). This suggests that stochastic rounding is important for near-lossless full-scale FP4 training.

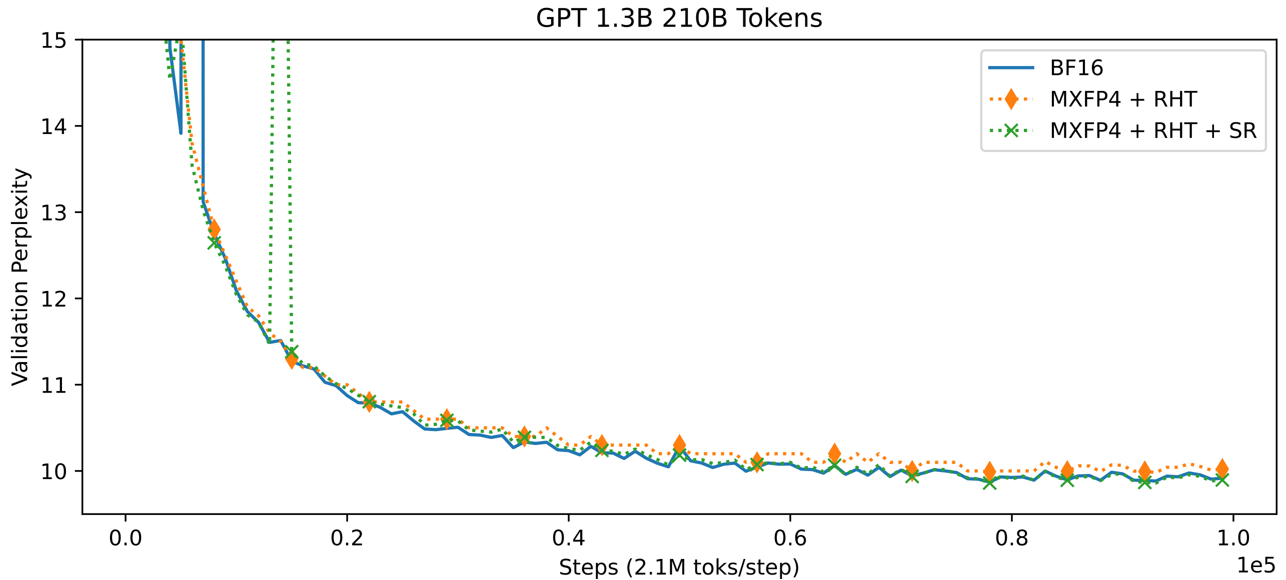

Figure 12: Validation perplexity for training GPT 1.3B for 210 billion tokens. All experiments used BF16 in the forward pass and the specified backward precision.

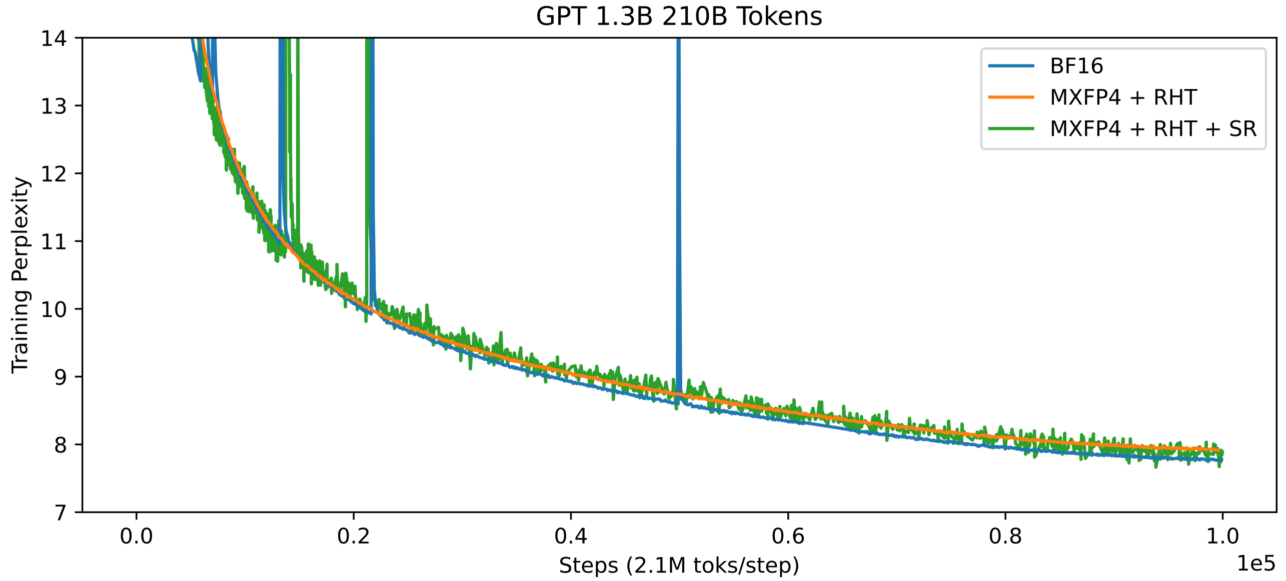

Figure 13: Training perplexity for training GPT 1.3B for 210 billion tokens. All experiments used BF16 in the forward pass and the specified backward precision. The BF16 and MXFP4+RHT curves have lower variance due to a configuration difference with the logger.

6.5 Training Curve for GPT 6.7B

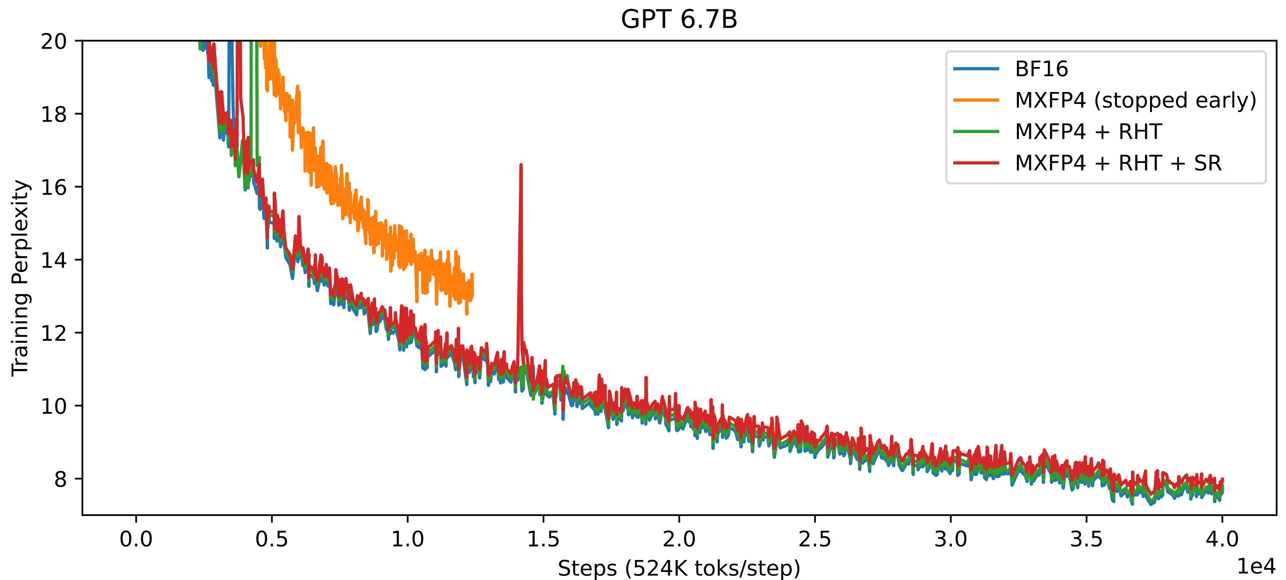

Figure 14: Training perplexity for training GPT 6.7B for 20 billion tokens. All experiments used BF16 in the forward pass and the specified backward precision.

7 Experimental Setup Details

All experiments were run on AWS P4 and G6e EC2 instances. Our code was based off of the Megatron-LM codebase at Github commit `a4ad305d4b117217141730b9b18af52dda069450` and the Microsoft microxcaling codebase at Github commit `7bc41952de394f5cc5e782baf132e7c7542eb4e4`. We used the NVIDIA Pytorch +

Ubuntu 24.04 docker image, which contains a version of Transformer Engine 1.5 for the FP8 experiments. All models were trained with the AdamW optimizer, FlashAttention, and the following hyperparameters:

Hyperparameter	GPT 345M	GPT 1.3B	GPT 6.7B
Decoder Layers	24	24	32
Hidden Size	1024	2048	4096
Attention Heads	16	16	32
Context Length	1024	2048	2048
Max. Positional Embeddings	1024	2048	2048
Batch Size	64	1024	256
Learning Rate (LR)	0.00015	0.0002	0.00012
Training Iterations	500000	20000	40000
LR Scheduler	Cosine	Cosine	Cosine
LR Decay Iterations	320000	20000	40000
Minimum LR	1e-5	2e-5	1.2e-5
Weight Decay	1e-2	0.1	0.1
LR Warmup Fraction	0.01	0.01	0.01
Gradient Clipping	1.0	1.0	1.0

8 Proof of Lemma 3.1

Lemma 3.1. *Assume stochastic rounding is implemented with dithering with independent noise. Then, Algorithm 2 produces a MXFP4 matrix that is an unbiased estimate of* $3/4$ *its input. Furthermore, Algorithm 3 with Algorithm 2 as a subroutine produces an unbiased estimate of* $\frac{dL}{dx}$ *and* $\frac{dL}{dW}$.

Proof. First, we show that Algorithm 2 produces an unbiased MXFP4 estimate of $\frac{3}{4}$ the input vector v. Let $v \in \mathbb{R}^g$, where g is the MX group size. The input to `stochastic_round_to_FP4` is given by $w = \frac{3}{4}v/X$, where $X = 2^{\lfloor \log_2(\text{argmax}(|v|)) \rfloor - 2}$. Let $m = \text{argmax}(|v|)$. Observe that the largest magnitude element of w is

$$\frac{3}{4}\frac{m}{2^{\lfloor \log_2(m) \rfloor - 2}} < \frac{3}{4}\frac{m}{2^{\log_2(m)-3}} = \frac{3}{4} \times 8 = 6$$

By definition, `stochastic_round_to_FP4`(x) produces an unbiased estimate of x as long as x is "within range" – i.e. it does not overflow outside of the range of representable values in FP4. Since the maximum normal in FP4 is 6, `stochastic_round_to_FP4`(w) will give an unbiased FP4 estimate of $\frac{3}{4}v/X$. Finally, from linearity of expectation, $X *$ `stochastic_round_to_FP4`(w) gives an unbiased estimate of $\frac{3}{4}v$, as desired.

Now, we show that Algorithm 3 produces unbiased estimates of $\frac{dL}{dx}$ and $\frac{dL}{dW}$. Let $C =$ `MXFP4_GEMM`(A, B^T), where $A \in \mathbb{R}^{b \times n}$ and $B \in \mathbb{R}^{m \times n}$, and $g|n$. We have that

$$\mathbb{E}\left[C_{ij}\right] = \mathbb{E}\left[\sum_{k=0}^{n/g}\left(X_{A_{i,kg:(k+1)g}}X_{B_{j,kg:(k+1)g}}\sum_{l=0}^{g}\left(A^{FP4}_{i,kg:(k+1)g}\right)_l\left(B^{FP4}_{j,kg:(k+1)g}\right)_l\right)\right] \tag{6}$$

$$= \sum_{k=0}^{n/g}\left(X_{A_{i,kg:(k+1)g}}X_{B_{j,kg:(k+1)g}}\sum_{l=0}^{g}\mathbb{E}\left[\left(A^{FP4}_{i,kg:(k+1)g}\right)_l\left(B^{FP4}_{j,kg:(k+1)g}\right)_l\right]\right) \tag{7}$$

Where $A_{i,kg:(k+1)g}$ denotes the k-th size g vector of the i-th row of A, $X_{A_{i,kg:(k+1)g}}$ is the scale of applying Algorithm 2 to $A_{i,kg:(k+1)g}$, and $A^{FP4}_{i,kg:(k+1)g}$ is the FP4 component of applying Algorithm 2 to $A_{i,kg:(k+1)g}$. Since stochastic rounding is implemented with independent noise, $A^{FP4}_{i,kg:(k+1)g}$ and $B^{FP4}_{j,kg:(k+1)g}$ are independent random

variables. Thus,

$$\mathbb{E}\left[C_{ij}\right] = \sum_{k=0}^{n/g}\left(X_{A_{i,kg:(k+1)g}}X_{B_{j,kg:(k+1)g}}\sum_{l=0}^{g}\mathbb{E}\left[\left(A^{FP4}_{i,kg:(k+1)g}\right)_l\left(B^{FP4}_{j,kg:(k+1)g}\right)_l\right]\right) \tag{8}$$

$$= \sum_{k=0}^{n/g}\left(X_{A_{i,kg:(k+1)g}}X_{B_{j,kg:(k+1)g}}\sum_{l=0}^{g}\mathbb{E}\left[\left(A^{FP4}_{i,kg:(k+1)g}\right)_l\right]\mathbb{E}\left[\left(B^{FP4}_{j,kg:(k+1)g}\right)_l\right]\right) \tag{9}$$

$$= \sum_{k=0}^{n/g}\left(X_{A_{i,kg:(k+1)g}}X_{B_{j,kg:(k+1)g}}\sum_{l=0}^{g}\frac{3}{4}\frac{\left(A_{i,kg:(k+1)g}\right)_l}{X_{A_{i,kg:(k+1)g}}}\frac{3}{4}\frac{\left(B_{j,kg:(k+1)g}\right)_l}{X_{B_{j,kg:(k+1)g}}}\right) \tag{10}$$

$$= \frac{9}{16}\sum_{h=0}^{n}A_{ih}B_{jh} = \frac{9}{16}(AB^T)_{ij} \tag{11}$$

For $\frac{dL}{dx}$, $A = \frac{dL}{dy}\text{diag}(S)H$ and $B = W^T\text{diag}(S)H$, where H is the block-diagonal "small" Hadamard matrix constructed in Section 3.2.Here, $\mathbb{E}\left[\texttt{MXFP4_GEMM}(A, B^T)\right] = \frac{9}{16}\frac{dL}{dy}\text{diag}(S)HH^T\text{diag}(S)W = \frac{9}{16}\frac{dL}{dy}W$. For $\frac{dL}{dW}$, $A = \frac{dL}{dy}^T\text{diag}(S)H$ and $B = x^T\text{diag}(S)H$. Here, $\mathbb{E}\left[\texttt{MXFP4_GEMM}(A, B^T)\right] = \frac{9}{16}\frac{dL}{dy}^T\text{diag}(S)HH^T\text{diag}(S)x = \frac{9}{16}\frac{dL}{dy}^T x$. Finally, scaling both values by 16/9 in lines 10 and 11 gives the desired unbiased gradient estimators.

□

9 Bounding the variance of SR with the RHT

Theorem 3.2. *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 $\mathcal{O}(b\Delta^4\|A\|_\infty\|B\|_\infty)$ and the variance of $\mathcal{Q}(HSA)^T\mathcal{Q}(HSB)$ is, with probability $\geq (1-\epsilon)^2$, $\mathcal{O}(\Delta^4\|A\|\|B\|\log(2b/\epsilon))$, where the largest gap between two consecutive representable points in $\mathcal{Q}$'s quantizer is Δ.*

Proof. Consider two vectors of size b: $A, B \in \mathbb{R}^b$. Then, the output of Algorithm 2 on A is a scale X_A and vector Q_A such that $\mathbb{E}[Q_{A_i}] = A_i/X_A$ and the expectation is taken over runs of Algorithm 2. Likewise, the output of Algorithm 2 on B is a scale X_B and vector Q_B s.t. $\mathbb{E}[Q_{B_i}] = B_i/X_B$. Let $C = X_AX_B\sum_{i=1}^{b}Q_{A_i}Q_{B_i}$. Since stochastic rounding is implemented with dithering on A and B with independent random noise,

$$\text{Var}(C) = X_AX_B\sum_{i=1}^{b}\text{Var}(Q_{A_i}Q_{B_i}) \tag{12}$$

$$= X_AX_B\left(\sum_{i=1}^{b}\text{Var}(Q_{A_i})\text{Var}(Q_{B_i}) + \text{Var}(Q_{A_i})\text{E}(Q_{B_i})^2 + \text{Var}(Q_{B_i})\text{E}(Q_{A_i})^2\right) \tag{13}$$

$$= X_AX_B\left(\sum_{i=1}^{b}\text{Var}(Q_{A_i})\text{Var}(Q_{B_i}) + \text{Var}(Q_{A_i})\left(\frac{B_i}{X_B}\right)^2 + \text{Var}(Q_{B_i})\left(\frac{A_i}{X_A}\right)^2\right) \tag{14}$$

$$= \sum_{i=1}^{b}X_AX_B\text{Var}(Q_{A_i})\text{Var}(Q_{B_i}) + \text{Var}(Q_{A_i})\left(\frac{X_AB_i^2}{X_B}\right) + \text{Var}(Q_{B_i})\left(\frac{X_BA_i^2}{X_A}\right). \tag{15}$$

Let $\alpha = A_i/X_A$. Since Q_{A_i} is the output of stochastic rounding to FP4, Q_{A_i} takes on values $f(\alpha)$ with probabilty $\frac{c(\alpha)-\alpha}{c(\alpha)-f(\alpha)}$ and $c(\alpha)$ with probability $\frac{\alpha-f(\alpha)}{c(\alpha)-f(\alpha)}$, where $f(\alpha)$ denotes the largest representable FP4 value $\leq \alpha$ and $c(\alpha)$ denotes the smallest representable FP4 value $\geq \alpha$. Observe that $f(\alpha)$ and $c(\alpha)$ are both guaranteed to exist due to line 4 in Algorithm 2. Then,

$$\mathrm{Var}(Q_{Ai}) = \frac{f(\alpha)^2(c(\alpha)-\alpha)+c(\alpha)^2(\alpha-f(\alpha))}{c(\alpha)-f(\alpha)} - \alpha^2 \tag{16}$$
$$= \frac{(c(\alpha)^2-f(\alpha)^2)\alpha+(f(\alpha)-c(\alpha))f(\alpha)c(\alpha)}{c(\alpha)-f(\alpha)} - \alpha^2 \tag{17}$$
$$= (c(\alpha)+f(\alpha))\alpha - f(\alpha)c(\alpha) - \alpha^2. \tag{18}$$

Let $\delta^+ = c(\alpha)-\alpha$ and $\delta^- = f(\alpha)-\alpha$. Then,

$$\mathrm{Var}(Q_{Ai}) = (c(\alpha)+f(\alpha))\alpha - f(\alpha)c(\alpha) - \alpha^2 \tag{19}$$
$$= (2\alpha+\delta^-+\delta^+)\alpha - (\alpha+\delta^-)(\alpha+\delta^+) - \alpha^2 \tag{20}$$
$$= -\delta^-\delta^+ = (c(\alpha)-\alpha)(\alpha-f(\alpha)) \tag{21}$$
$$= \mathcal{O}((c(\alpha)-f(\alpha))^2). \tag{22}$$

Since $c(\alpha)-f(\alpha)$ is $O(\Delta)$,

$$\mathrm{Var}(C) = \mathcal{O}\left(b\Delta^4 X_A X_B + \Delta^2 \frac{X_A}{X_B}\sum_{i=1}^{b} B_i^2 + \Delta^2 \frac{X_B}{X_A}\sum_{i=1}^{b} A_i^2\right) \tag{23}$$
$$= \mathcal{O}\left(b\Delta^4 X_A X_B + \Delta^2 \frac{X_A}{X_B}\|B\|^2 + \Delta^2 \frac{X_B}{X_A}\|A\|^2\right). \tag{24}$$

Since $X_A = \Theta(\|A\|_\infty)$ and likewise for B, this reduces to

$$\mathrm{Var}(C) = \mathcal{O}\left(b\Delta^4\|A\|_\infty\|B\|_\infty + 2b\Delta^2\|A\|_\infty\|B\|_\infty\right) \tag{25}$$
$$= \mathcal{O}(b\Delta^4\|A\|_\infty\|B\|_\infty) \tag{26}$$

If A and B are transformed by the RHT in the way Algorithm X does (i.e. $\tilde{A} \leftarrow ASH^T$ and $\tilde{B} \leftarrow HSB$), then we can bound $\|\tilde{A}\|_\infty$ and $\|\tilde{B}\|_\infty$. From Tseng et al. (2024a), $\forall i, 1 \le i \le b$,

$$\mathbb{P}(|e_i\tilde{A}| \ge \epsilon) = \mathbb{P}(|e_i ASH^T| \ge \epsilon) \le 2\exp\left(\frac{-\epsilon^2 b}{2\|A\|^2}\right). \tag{27}$$

From the union bound,

$$\mathbb{P}\left(\max_i |e_i ASH^T| \ge \epsilon\right) \le 2b\exp\left(\frac{-\epsilon^2 b}{2\|A\|^2}\right) \tag{28}$$
$$\mathbb{P}\left(\max_i |e_i ASH^T| \ge \sqrt{\frac{2\|A\|^2}{b}\log\left(\frac{2b}{\epsilon}\right)}\right) \le \epsilon. \tag{29}$$

so with probability $\ge 1-\epsilon$,

$$\|A\|_\infty = \mathcal{O}\left(\sqrt{\frac{2\|A\|^2}{b}\log\left(\frac{2b}{\epsilon}\right)}\right) \tag{30}$$

and with probability $\ge (1-\epsilon)^2$,

$$\mathrm{Var}(C) = \mathcal{O}\left(\Delta^4\|A\|\|B\|\log\left(\frac{2b}{\epsilon}\right)\right). \tag{31}$$

□