COLLAGE: Light-Weight Low-Precision Strategy for LLM Training

Tao Yu [*1] Gaurav Gupta [†2] Karthick Gopalswamy [3] Amith Mamidala [3] Hao Zhou [4] Jeffrey Huynh [3]
Youngsuk Park [5] Ron Diamant [3] Anoop Deoras [2] Luke Huan [2]

Abstract

Large models training is plagued by the intense compute cost and limited hardware memory. A practical solution is low-precision representation but is troubled by loss in numerical accuracy and unstable training rendering the model less useful. We argue that low-precision floating points can perform well provided the error is properly compensated at the critical locations in the training process. We propose COLLAGE which utilizes multi-component float representation in low-precision to accurately perform operations with numerical errors accounted. To understand the impact of imprecision to training, we propose a simple and novel metric which tracks the lost information during training as well as differentiates various precision strategies. Our method works with commonly used low-precision such as half-precision (16-bit floating points) and can be naturally extended to work with even lower precision such as 8-bit. Experimental results show that pre-training using COLLAGE removes the requirement of using 32-bit floating-point copies of the model and attains similar/better training performance compared to $(16, 32)$-bit mixed-precision strategy, with up to $3.7\times$ speedup and $\sim 15\%$ to 23% less memory usage in practice. The code is available at `https://github.com/amazon-science/collage`.

1. Introduction

Recent success of large models using transformers backend has gathered the attention of community for generative language modeling (GPT-4 (OpenAI et al., 2023), LaMDA (Thoppilan et al., 2022), LLaMa (Touvron et al., 2023)), image generation (e.g., Dall-E (Betker et al., 2023)), speech generation (such as Meta voicebox, OpenAI jukebox (Le et al., 2023; Dhariwal et al., 2020)), and multimodality (e.g. gemini (Team et al., 2023)) motivating to further scale such models to larger size and context lengths. However, scaling models is prohibited by the hardware memory and also incur immense compute cost in the distributed training, such as $\sim$1M GPU-hrs for LLaMA-65B (Touvron et al., 2023), thus asking the question whether large model training could be made efficient while maintaining the accuracy?

Previous works have attempted to reduce the memory consumption and run models more efficiently by reducing precision of the parameter's representation, at training time (Zhang et al., 2022; Kuchaiev et al., 2018; 2019; Peng et al., 2023) and post-training inference time (Courbariaux et al., 2016; Rastegari et al., 2016; Choi et al., 2019). The former one is directly relevant to our work using *low-precision storages* at training time, but it suffers from issues such as numerical inaccuracies and narrow representation range. Researchers developed algorithms such as loss-scaling and mixed-precision (Micikevicius et al., 2018; Shoeybi et al., 2020) to overcome these issues. Existing algorithms still face challenges in terms of memory efficiency as they require the presence of high-precision clones and computations in optimizations. One critical limitation of all the aforementioned methods is that such methods keep the "standard format" for floating-points during computations and lose information with a reduced precision.

In this work, we elucidate that in the setting of low-precision (for example, 16-bit or lower) for floating point, using alternative representations such as multiple-component float (MCF) (Yu et al., 2022b) helps in making reduced precision accurate in computations. MCF was introduced as 'expansion' (Priest, 1991) in C++ (Hida et al., 2008) and hyperbolic spaces (Yu & De Sa, 2021) representation. Recently, MCF has been integrated with PyTorch in the MCTensor library (Yu et al., 2022b).

We propose COLLAGE[1], a new approach to deal with floating-point errors in low-precision to make LLM training

[*]Work conducted during an internship at Amazon [†]Major Contribution [1]Cornell University, Ithaca, NY [2]AWS AI Labs, Santa Clara, CA [3]AWS Annapurna Labs, Cupertino, CA [4]AWS Sagemaker, Santa Clara, CA [5]AWS AI Research and Education, Santa Clara, CA. Correspondence to: Gaurav Gupta <gauravaz@amazon.com>.

Proceedings of the 41st International Conference on Machine Learning, Vienna, Austria. PMLR 235, 2024. Copyright 2024 by the author(s).

[1]Inspired from the multi-component nature of the algorithm.

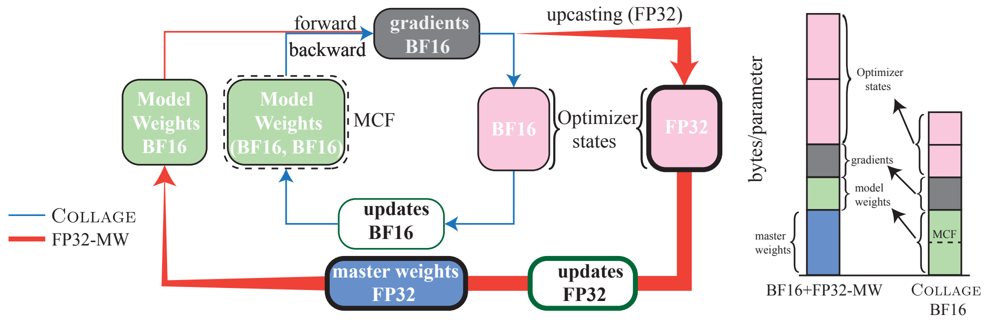

Figure 1. **Left:** COLLAGE uses a strict low-precision floating-point (such as BF16) optimization loop without ever needing to upcast to FP32 like in the mixed-precision with master weights (red thick loop). The model weights in COLLAGE are represented as multi-component float (MCF) instead of "standard float". **Right:** Total bytes/parameter savings for COLLAGE without taking the FP32 upcasting route. The memory savings and uncompromising use of low-precision results in speed-up as seen in Table 7.

accurate and efficient. Our primary objective is to develop a training loop with storage strict in low-precision without a need to maintain high-precision clones. We realize that when dealing with low-precision floats (such as Bfloat16), the "standard" representation is not sufficient to avoid rounding errors which *should not be ignored*. To solve these issues, we rather apply an existing technique of MCF to represent floats which (i) either encounters drastic rounding effects, (ii) the scale of the involved floats has a wide range such that arithmetic operations were lost. We implemented COLLAGE as a plugin to be easily integrated with the well-known optimizers such as AdamW (Loshchilov & Hutter, 2017) (extensions to SGD (Ruder, 2017) are straightforward) using low-precision storage & computations. By turning the optimizer to be more *precision-aware*, even with additional low-precision components in MCF, we obtain faster training (upto 3.7× better train throughput on 6.7B GPT model, Table 7) and also have less memory foot-print due to strict low-precision floats (see Figure 1 right), compared to the most advanced mixed precision baseline.

We have developed a novel metric called "effective descent quality" to trace the lost information in the optimizer model update step. Due to rounding and lost arithmetic (see definition in Section 3.1), the effective update applied to the model is different from the intended update from optimizer, thus distracting the model training trajectory. Tracing this metric during the training enables to compare different precision strategies at a fine-grained level (see Figure 3 right).

In this work, we answer the critical question of where (which computation) with low-precision during training is severely impacting the performance and why? The main contributions are outlined as follows.

- We provide COLLAGE as a **plugin** which could be easily integrated with existing optimizer such as AdamW for low-precision training and make it *precision-aware* by replacing critical floating-points with MCF. This avoids the path of high-precision master-weights and upcasting of variables, achieving memory efficiency (Figure 1 right).
- By proposing the metric effective descent quality, we measure loss in the information at model update step during the training process and provide **better understanding** of the impact of precisions and **interpretation** for comparing precision strategies.
- COLLAGE offers wall-clock time speedups by storing all variables in low-precision without upcasting. For GPT-6.7B and OpenLLaMA-7B, COLLAGE using bfloat16 has **up to 3.7**× speedup in the training throughput in particular settings, in comparison with mixed-precision strategy with FP32 master weights while following a similar training trajectory. The peak memory savings for GPTs (125M - 6.7B) is on average of $\mathbf{22.8\%/14.9\%}$ for COLLAGE formations (light/-plus), respectively.
- COLLAGE **trains accurate** models using only low-precision storage compared with FP32 master-weights counterpart. For RoBERTa-base, the average GLUE accuracy scores differ by $\mathbf{+0.85\%}$ among the best baseline in Table 4. Similarly, for GPT of sizes 125M, 1.3B, 2.7B, 6.7B, COLLAGE has **similar validation perplexity** as FP32 master weights in Table 5.

2. Background

We provide a survey on using different floating-points precision strategies for training LLM. We also introduce necessary background information on floating-point representations using a new structure, multi-component float.

2.1. Floats in LLM Training

In LLM training, weights, activation, gradients are usually stored in low precision floating-points such as 16-bit BF16

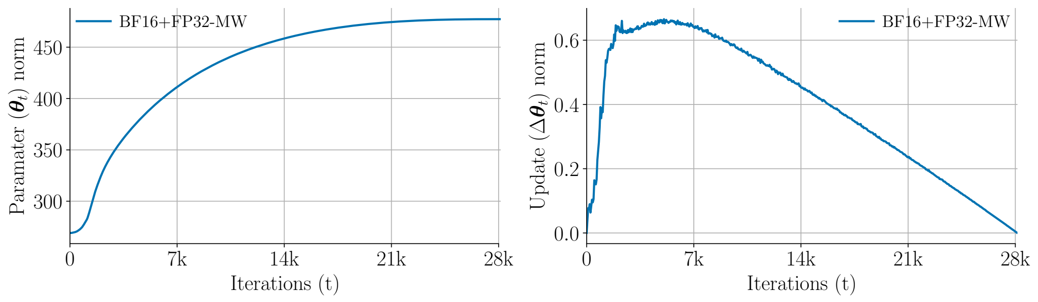

Figure 2. Bert-base-uncased phase-1 pretraining with settings as described in Section 5.1. **Left:** Model parameter L2 norm vs iterations for BF16 and FP32 master weights strategy. **Right:** update $\Delta\theta_t$ L2 norm across iterations. The model parameter norm and update norm are at different scales, for example, $\sim$ 450 vs $\sim$ 0.5 at 14k iterations, which is a factor of 900 and causes lost arithmetic.

(Micikevicius et al., 2017) for enhanced efficiency and optimized memory utilization. The low-bits floating point units (FPUs) are appealing because of its low memory foot-print and computational efficiency. Due to numerical inaccuracies, popular choices of training strategies using FPUs are as follows.

Mixed-precision refers to operations executed in low precision (16-bit) with minimal interactions with high precision (32-bits) floats, thus offering wall-clock speedups. For example, in GEMM (Generalized Matrix Multiplication), matrix multiplication is performed in 16-bit while accumulation in done in 32-bit through tensor cores in NVIDIA A100 (Jia & Sandt, 2021) and V100 (Jia et al., 2018).

Mixed-precision with Master Weights. Mixed-precision computations of the activations and gradients are not sufficient to ensure a stable training due to encountered numerical inaccuracies, especially, when gradients and model parameters are at different scale, which is the case with large models (see Figure 2). A standard workaround is to use the master weight (MW), which refers to maintaining an additional high-precision version (such as 32-bit float) copy of the model (Figure 1 left) and then performing model update (optimizer step) in high-precision to the master weight (Micikevicius et al., 2018). To our knowledge, this approach has the state-of-the-art performance among mixed-precision strategies.

Note that, we also use mixed-precision for GEMM (activations and gradients) in our work. In addition to "standard single float" representation which is used in the above strategies, an alternate form is discussed below in Section 2.2.

2.2. Multiple-Component Floating-point

Precise computations can be achieved with one of two approaches in numerical computing.

(i) *multiple-bit*, i.e., using "standard single float" with more bits in the mantissa/fraction, such as 32-, 64-bit floats, and even Bigfloat (Granlund, 2004);
(ii) *multiple-component* representation using unevaluated sum $x_1 + x_2 + \cdots + x_n$ of multiple floats usually in low-precision such as BF16, FP16, or even FP8.

Multiple-bit approach has an advantage of large representation range, while the multiple-component floating-point (MCF) has an advantage in speed, as it consists of only low-precision floating-point computations. Additionally, rounding is often required in p-bit "standard single float" arithmetic due to output requiring additional bits to express and store exactly, while in MCF, the rounding error could be circumvent and accounted via appending additional components. A basic structure in MCF is expansion:

Definition 2.1. (Priest, 1991). A length-n expansion $(x_1, \ldots, x_n)$ represents the unevaluated exact sum $x = x_1 + x_2 + \cdots + x_n$, where components x_i are non-overlapping p-bit floating-points in decreasing order, i.e., for $i < j$, the least significant non-zero bit of x_i is more significant than the most significant non-zero bit of x_j or vice versa.

Exact representations of real numbers such as 0.999 is usually muddled in low-precision, such as BF16, with rounding-to-the-nearest (RN); $0.999 \xrightarrow[\text{BF16}]{\text{RN}} 1.0$, but can be represented accurately as a length-2 expansion $(1.0, -0.001)$ in MCF with two BF16 components. The first component serves as an approximation to the value, while the second accounts for the roundoff error. This problem is further aggravated in weighted averaging (see Section 4.2), such that instead of the average, a monotonic increasing sum is produced causing reduced step size and poor learning. We aim to alleviate such problems by using expansions to represent numbers and parameters accurately (e.g., Table 1). Since speed and scalability is critical for LLM training, we are particularly interested in utilizing low-precision MCF (e.g., BF16 and FP16) as low-bit FPUs are faster than their high-bit counterparts such as FP32. For rest of the work, we consider only length-2 expansion for MCF as it suffices for our purpose.

3. Imprecision Issues

To motivate the work, in this section, we formalize the issue of imprecision in floating point units. Afterwards, we introduce a novel metric to monitor the information loss. Next, we show its impact via a case study on BERT-like models (Devlin et al., 2019; Liu et al., 2019). Unless specified otherwise, the low-precision FPU is referred to bfloat16, and the same analogy can be easily extended for other low-precision FPUs such as float16, float8.

3.1. Imprecision with Bfloat16

A commonly encountered problem of computations using low-precision arithmetic is *imprecision*, where an exact representation of a real-number x either requires more mantissa bits (see Appendix A for definitions) beyond the limit (for example, 7 bits in bfloat16), or is not possible (for example, $x = 0.1$, is rounded to 0.1001 in BF16). As a result, the given number x will be rounded to a representable floating-point value, causing numerical quantization errors. An important concept for FPU rounding is unit in the last place (ulp), which is the spacing between two consecutive representable floating-point numbers, i.e., the value the least significant (rightmost) bit represents if it is 1.

Definition 3.1 (ulp (Muller et al., 2018))**.** In radix 2 with precision P, if $2^e \leq |x| < 2^{e+1}$ for some integer e, then $\text{ulp}(x) = 2^{\max(e, e_{\min}) - P}$, where $e_{\min}$ is the zero offset in the IEEE 754 standard.

Broadly speaking, two numbers for a given FPU are separated by its ulp, hence the worst case rounding error for any given x is $\text{ulp}(x)/2$ (Goldberg, 1991) assumed rounding-to-the-nearest is used. Next, lets denote $\mathcal{F}^{\text{BF16}}(a \varkappa b)$ as bfloat16 floating-point operation between a, b, where $\varkappa$ could be $\oplus$ addition, $\odot$ multiplication, etc. Such operations can be computationally inaccurate and as a consequence, we identify below a problematic behavior with RN.

Definition 3.2 (Lost Arithmetic)**.** Given the input floating-point numbers a, b and precision P. A floating operation $\mathcal{F}^P(a \varkappa b)$ is lost if

$$|\mathcal{F}^P(a \varkappa b) - a| \leq \frac{\text{ulp}(a)}{2}, \text{ or } |\mathcal{F}^P(a \varkappa b) - b| \leq \frac{\text{ulp}(b)}{2}.$$

Consequently, $\mathcal{F}^P(a \varkappa b) = a,$ or b, respectively.

Remark: For any non-zero bfloat16 number, if $|b| \leq \text{ulp}(a)/2$, then $\mathcal{F}^{\text{BF16}}(a \oplus b) = a$. As an example, if $a = 200, b = 0.1$, then $\mathcal{F}^{\text{BF16}}(200 \oplus 0.1) = 200$, since $\text{ulp}(200) = 1$. Next, we discuss these concepts in the context of LLM training.

3.2. Loss of Information in LLM Training

The situation of 'adding two numbers at different scale' is very common in LLM training. See Figure 2 , where due to different scales of model parameter and updates, $\oplus$ in bfloat16 becomes an *lost arithmetic*. A pseudocode of model parameter ($\boldsymbol{\theta}$) update using bfloat16 at iteration t is written as

$$\boldsymbol{\theta}_t \leftarrow \mathcal{F}^{\text{BF16}}(\boldsymbol{\theta}_{t-1} \oplus \Delta\boldsymbol{\theta}_t), \tag{1}$$

where, $\Delta\boldsymbol{\theta}_t$ is the aggregated update from an optimizer (for example, including learning rate, momentum, etc.) at step t. With a possibility of lost arithmetic in Equation (1), the actual updated parameter could be different from expected. Hence, we define the effective update at step t as

$$\widehat{\Delta\boldsymbol{\theta}}_t = \mathcal{F}^{\text{BF16}}(\boldsymbol{\theta}_{t-1} \oplus \Delta\boldsymbol{\theta}_t) - \boldsymbol{\theta}_{t-1}. \tag{2}$$

Note that in the event of no lost arithmetic, $\widehat{\Delta\boldsymbol{\theta}}_t = \Delta\boldsymbol{\theta}_t$. While, when $\widehat{\Delta\boldsymbol{\theta}}_t \neq \Delta\boldsymbol{\theta}_t$ which is usually the case with low-precision FPUs, there is a loss in information as $\leq \text{ulp}/2$ values are simply ignored (see Figure 3(a)). To better capture this information loss, we introduce a novel metric.

Definition 3.3 (**Effective Descent Quality**)**.** Given the current parameter, aggregated update at step t as $\boldsymbol{\theta}_t$, $\Delta\boldsymbol{\theta}_t$, respectively. The effective descent quality for a given floating-pint precision is defined as

$$\text{EDQ}(\Delta\boldsymbol{\theta}_t, \widehat{\Delta\boldsymbol{\theta}}_t; \boldsymbol{\theta}_t, P) = \left\langle \frac{\Delta\boldsymbol{\theta}_t}{||\Delta\boldsymbol{\theta}_t||}, \widehat{\Delta\boldsymbol{\theta}}_t \right\rangle, \tag{3}$$

where, $\widehat{\Delta\boldsymbol{\theta}}_t$ is defined in eq. (2) for a given precision P.

In other words, EDQ in eq. (3) is projection of the effective update along the desired update. In the absence of any imprecision, EDQ will be simply the norm of original update. We show in Section 5.1 and Figure 3 how EDQ relates to the learning and helps understanding impacts of different precision strategies.

To remedy the imprecision and lost arithmetic in the model parameter update step (Equation (1)), works such as Kahan summation (Zamirai et al., 2020; Park et al., 2018) exist (see Appendix B), however, we see in Figure 3 (Middle) that although Kahan-based BF16 approach improves over 'BF16' training but it still could not match with the commonly used FP32 master weights approach.

4. COLLAGE: Low-Precision MCF Optimizer

In this section, we present COLLAGE, a low precision strategy & optimizer implementation to solve aforementioned imprecision and lost arithmetic issues in Section 3 without upcasting to a higher precision, using the multiple-component floating-point (MCF) structure.

4.1. Computing with MCF

Precise computing with exact numbers stored as MCF expansions is easy with some basic algorithms[2]. For example, `Fast2Sum` captures the roundoff error for the float addition $\oplus$ and outputs an expansion of length 2.

Theorem 4.1 (Fast2Sum (Dekker, 1971))**.** *Let two floating-point numbers a, b be $|a| \geq |b|$, `Fast2Sum` produces a MCF expansion (x, y) such that $a + b = x + y$, where $x \leftarrow \mathcal{F}^P(a \oplus b)$ is the floating-point sum with precision P, $y \leftarrow \mathcal{F}^P\left(b \ominus \mathcal{F}^P(x \ominus a)\right) = a + b - \mathcal{F}^P(a \oplus b)$ is the rounding error. Also, y is upper-bounded such that $|y| < \mathrm{ulp}(x)/2$.*

Note that, particularly for LLM training, we are able to add using `Fast2Sum` without any sorting since parameter weights $\boldsymbol{\theta}$ are usually larger than the gradients and updates $\boldsymbol{\Delta\theta}$ in absolute value at the parameter update step Equation (1) (See Figure 2). Similar basic algorithms exist for the multiplication of two floats, which produces in the same way a length-2 expansion. Using the basic algorithms, an exhaustive set of advanced algorithms are developed (Yu et al., 2022b). We refer the reader to Appendix C for more details. Particularly, for the optimizer update step (1), a useful algorithm to introduce is `Grow` (see Algorithm 1) which adds a float to a MCF expansion of length 2.

Algorithm 1 Grow

```
1: Input: an expansion $(x, y)$ and a float $a$ with $|x| \geq |a|$
2: $(u, v) \leftarrow$ Fast2Sum$(x, a)$
3: $(u, v) \leftarrow$ Fast2Sum$(u, y + v)$
4: Return: $(u, v)$
```

4.2. COLLAGE: Bfloat16 MCF AdamW

Using the basic components from Section 4.1 and Appendix C, we now provide plugins to modify a given optimizer such as AdamW (Loshchilov & Hutter, 2017) to be *precision-aware* and **store entirely with low-precision** floats, specifically bfloat16 in Algorithm 2. Note that, mixed-precision is still used in GEMM for obtaining gradients and activations but are stored in bfloat16 only. The required changes are highlighted in pink, and are discussed individually as follows.

Model Parameters We substitute the bfloat16 model parameter $\boldsymbol{\theta}_t$ with a length-2 MCF expansion $(\boldsymbol{\theta}_t, \boldsymbol{\delta\theta}_t)$ by appending an additional bfloat16 variable $\boldsymbol{\delta\theta}_t$ in line-3 which does not require any gradients. Next, to update the model parameter expansion, we use `Grow` in line-13 to add a float $\Delta\boldsymbol{\theta}_t$ to the expansion.

[2]The correctness of algorithms presented herein rely on the assumption that standard rounding-to-the-nearest is used.

Table 1. length-2 expansions for β_2 in Bfloat16.

β_2	BF16 MCF
0.999	$(1, -0.001)$
0.99	$(0.9893, 0.0017)$
0.95	$(0.9492, 0.0008)$

Optimizer States With Adam-like algorithms, unlike the first moment $\boldsymbol{m}_t$, the second moment $\boldsymbol{v}_t$ update suffers from severe imprecision and lost arithmetic due to smaller accumulation, $\boldsymbol{g}_t$ vs $\boldsymbol{g}_t^2$. To make the matter worse, default choice of β_2 such as 0.999 (Devlin et al., 2019) are simply rounded to 1.0 in bfloat16, thus resulting in a monotonic increase in second momentum. This in turn makes the update $\boldsymbol{\Delta\theta}_t$ smaller and hence slower learning as we see in Figure 3. To alleviate this issue, we propose switching β_2 from standard single float to a MCF expansion as $(\beta_2, \delta\beta_2)$, and also for second momentum as $(\boldsymbol{v}_t, \boldsymbol{\delta v}_t)$. Doing so, we have an exact representation of β_2 as shown in Table 1. We then perform a multiplication of two expansions using `Mul` (see Appendix C).

For the sake of simplicity in notations, we denote COLLAGE-light as using MCF expansions only for model parameters and COLLAGE-plus for both model parameters and optimizer states. It's worthy to note that imprecision and lost arithmetic are common and sometimes hard to notice. We only identify places when they hurt training accuracies. A rule of thumb is to do as many scalar computations in high precision as possible before casting them to low precision (e.g., PyTorch BFloat16 Tensor). Worthy to note, existing Kahan-based optimizers are special cases of COLLAGE-light under a magnitude assumption, we defer this discussion and other places of imprecision and lost arithmetic such as weight decay that exist in the algorithm to Appendix D.

In our experiments, we focus on Bfloat16 low precision due to limited FP8 hardware availability, yet the Algorithm-2 in its current format is straightforwardly extensible to FP8 without any training pipeline changes, with the detailed FP8 training recipe provided in the Appendix D.

5. Empirical Evaluation

We evaluate COLLAGE formations against the existing precision strategies on pretraining LLMs at different scales, including BERT (Devlin et al., 2019), RoBERTa (Liu et al., 2019), GPT (Andonian et al., 2023), and OpenLLaMA (Touvron et al., 2023). Specifically, we compared the following precision strategies in our experiments, which are ordered in an increasing number of byte/parameter (see Table 2).

- Option A: Bfloat16 parameters
- Option B: Bfloat16 + COLLAGE-light
- Option C: Bfloat16 + COLLAGE-plus
- Option D: Bfloat16 + FP32 Optimizer states + FP32 master weights

Since option D is the best-known baseline with state-of-the-art quality among mixed-precision strategies, we aim

Algorithm 2 COLLAGE: Bfloat16 MCF AdamW Optimization

```
1: Given $\alpha$ (learning rate), $\beta_1$, $\beta_2$, $\epsilon$, $\lambda \in \mathbb{R}$
2: Initialize time step: $t \leftarrow 0$, BF16 parameter vector $\boldsymbol{\theta}_{t=0} \in \mathbb{R}^n$, BF16 first moment vector: $\boldsymbol{m}_{t=0} \leftarrow \mathbf{0}$, BF16 second moment vector: $\boldsymbol{v}_{t=0} \leftarrow \mathbf{0}$
3: Initialize 2nd component $\boldsymbol{\delta\theta}_{t=0} \leftarrow \mathbf{0}$ in BF16 for parameter
4: (optional) Represent $\beta_2$ as expansion $(\hat{\beta}_2, \delta\beta_2)$, initialize 2nd component $\boldsymbol{\delta v}_{t=0} \leftarrow \mathbf{0}$ in BF16 for second moment
5: repeat
6:     $t \leftarrow t + 1$
7:     $\boldsymbol{g}_t \leftarrow \nabla f_t(\boldsymbol{\theta}_{t-1})$
8:     $\boldsymbol{m}_t \leftarrow \beta_1 \cdot \boldsymbol{m}_{t-1} + (1 - \beta_1) \cdot \boldsymbol{g}_t$
9:     $\boldsymbol{v}_t \leftarrow \beta_2 \cdot \boldsymbol{v}_{t-1} + (1 - \beta_2) \cdot \boldsymbol{g}_t^2 \implies (\boldsymbol{v}_t, \boldsymbol{\delta v}_t) \leftarrow \mathbf{Grow}(\mathbf{Mul}(\hat{\beta}_2, \delta\beta_2), (\boldsymbol{v}_{t-1}, \boldsymbol{\delta v}_{t-1})), (1 - \beta_2) \cdot \boldsymbol{g}_t^2)$
10:    $\hat{\boldsymbol{m}}_t \leftarrow \boldsymbol{m}_t / (1 - \beta_1^t)$
11:    $\hat{\boldsymbol{v}}_t \leftarrow \boldsymbol{v}_t / (1 - \beta_2^t)$
12:    $\boldsymbol{\Delta\theta}_t \leftarrow -\alpha(\hat{\boldsymbol{m}}_t / (\sqrt{\hat{\boldsymbol{v}}_t + \epsilon}) + \lambda \boldsymbol{\theta}_{t-1})$
13:    $\boldsymbol{\theta}_t \leftarrow \boldsymbol{\theta}_{t-1} + \boldsymbol{\Delta\theta}_t \implies (\boldsymbol{\theta}_t, \boldsymbol{\delta\theta}_t) \leftarrow \mathbf{Grow}((\boldsymbol{\theta}_{t-1}, \boldsymbol{\delta\theta}_{t-1}), \boldsymbol{\Delta\theta}_t)$
14: until stopping criterion is met
15: return: optimized parameters $\boldsymbol{\theta}_t$
```

Table 2. Precision breakdown of various training strategies applied to the given optimizer. The strategies are ranked from top to bottom in the order of byte/parameter occupancy.

Precision Option	Stages & Components			Memory (bytes/parameter)
	Parameter & Gradient	Optimizer States	MCF or Master Weight	
A (BF16)	BF16 $\times$ 2	BF16 $\times$ 2	NA	8
B (COLLAGE-light) (ours)	BF16 $\times$ 2	BF16 $\times$ 2	BF16 $\times$ 1	10
C (COLLAGE-plus) (ours)	BF16 $\times$ 2	BF16 $\times$ 2	BF16 $\times$ 2	12
D (BF16 + FP32Optim + FP32MW)	BF16 $\times$ 2	FP32 $\times$ 2	FP32 $\times$ 1	16

to outperform, or at least match the quality of option D with COLLAGE throughout our experiments. We show that COLLAGE matching the quality of option D, has **orders-magnitude higher** performance (speed, see Table 7). All strategies are evaluated using AdamW (Loshchilov & Hutter, 2017) optimizer with standard $\beta_1 = 0.9$ while varying β_2 as per different experiments. We use *aws.p4.24xlarge* compute instances for all of our experiments.

5.1. Pre-training BERT & RoBERTa

We demonstrate that BF16-COLLAGE can be used to obtain an accurate model, comparable to heavy-weighted FP32 master weights strategy.

Precision options. In addition to options A, B, C, D, we further augment our experiments with another baseline strategy D$^{-\text{MW}}$, where we disabled the FP32 master weights but only used FP32 optimizer states. This strategy saves 4 bytes/parameter in comparison to Option D and has the same bytes/parameter as option C (COLLAGE-plus).

Model and Dataset. We first pre-train the BERT-base-uncased, BERT-large-uncased, and RoBERTa-base model with HuggingFace (HF) (Wolf et al., 2019) configuration on the Wikipedia-en corpus (Attardi, 2015), preprocessed with BERT Wordpiece tokenizer. We execute the following pipeline to pretrain, i) BERT in two phases with phase-1 on 128 sequence length, and then phase-2 with 512 sequence length; and ii) RoBERTa with sequence length 512. We adopt $\beta_2 = 0.999$ for BERT and $\beta_2 = 0.98$ for RoBERTa following the configs from HF. We defer more training details to Appendix E.1.

Results. The final pretraining perplexity of various precision strategies are summarized in Table 3 and for BERT-base, the complete phase-1 training loss trajectory is shown in Figure 3 middle. Additionally, we did finetuning of the pre-trained models on the GLUE benchmark (Wang et al., 2019) for eight tasks in Table 4 with the same configurations specified in Appendix E.1. COLLAGE-plus although using only BF16 parameters, outperforms the vanilla BF16 option A and matches/exceeds option D for both pre-training and finetuning experiments. For BERT-base COLLAGE-plus

Table 3. Pre-training perplexity of BERT (both phases) and RoBERTa for all precision strategies as listed in Table 2. Lower values are better, with the best results in bold. D$^{-\mathrm{MW}}$ with FP32Optim with same bytes/parameter as COLLAGE could not match its performance.

Precision Option	$\beta_2 = 0.999$				$\beta_2 = 0.98$
	BERT-base		BERT-large		RoBERTa-base
	Phase-1	Phase-2	Phase-1	Phase-2	
A	8.67	7.61	6.05	5.47	3.82
B (COLLAGE-light)	5.99	5.26	4.39	3.90	3.49
C (COLLAGE-plus)	**5.26**	**4.66**	**3.94**	**3.53**	3.49
D$^{-\mathrm{MW}}$ (BF16 + FP32Optim)	6.23	5.64	4.66	4.22	3.82
D	**5.26**	4.71	4.06	3.63	**3.46**

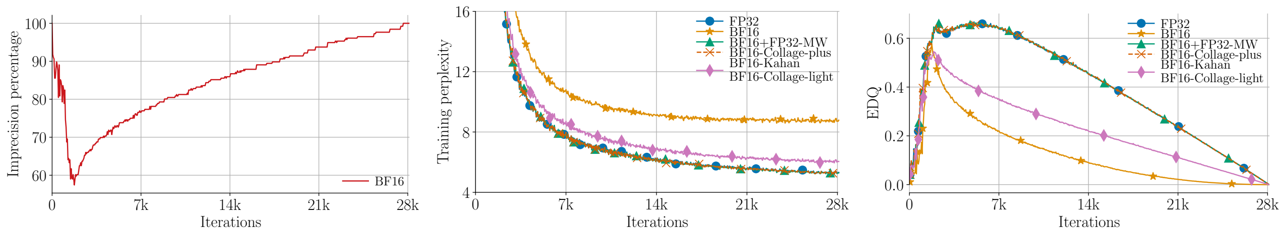

Figure 3. BERT phase-1 pre-training (see Appendix E.1 for details). **Left:** Imprecision percentage (%) measured as the percentage of lost arithmetic for all model parameters, i.e., not updated, vs iterations for BF16. **Middle:** Training perplexity vs iterations for various precision strategies (see Table 2). Additionally, we evaluate "FP32" as 32-bit counterpart of option A, and BF16-Kahan as Kahan-sum (Zamirai et al., 2020) with BF16 parameters. **Right:** Effective descent quality (EDQ) in (3) vs iterations to measure loss in information at the optimizer step for different precision strategies. BF16-COLLAGE-plus training perplexity and EDQ **overlaps** with the best "FP32", and "BF16 + FP32 MW" with less bytes/parameter.

exceeds on **5/8** tasks with $+\mathbf{0.1}\%$ lead in average, while for roberta-base its exceeds on **7/8** tasks with $+\mathbf{0.85}\%$ in average. Note that, although D$^{-\mathrm{MW}}$ has FP32 optimizer states and same/more byte/parameter complexity as COLLAGE-plus/light, respectively, it could not match the quality **showing the importance of MCF** in the AdamW through COLLAGE. This shows that simply having higher-precision is not enough to obtain better models but requires a *careful consideration of the floating errors*.

Interestingly, COLLAGE-light suffices to closely match the option D in the RoBERTa pretraining experiments with $\beta_2 = 0.98$, while lagging to match with the $\beta_2 = 0.999$ BERT pretraining experiments. Our proposed metric, the effective descent quality (EDQ) provides a nuanced understanding of this phenomenon in Figure 3 (Right). COLLAGE-light and Kahan-based approach improve EDQ upon BF16 option A at the parameter update step, yet cannot achieve the optimal EDQ due to lost arithmetic at the exponential moving averaging step. In contrast, COLLAGE-plus achieves better EDQ by taking it into considerations and thereby outperforms the best-known baseline, Option D.

5.2. Pretraining multi-size GPTs & OpenLLaMA 7B

Model and Dataset. We conduct following pretraining experiments; 1) GPT with different sizes ranging from 125M, 1.3B, 2.7B to 6.7B, and 2) OpenLLaMA-7B using NeMo Megatron (Kuchaiev et al., 2019) with the provided configs. The GPTs are trained on the Wikipedia corpus (Attardi, 2015) with GPT2 BPE tokenizer, and OpenLLaMA-7B on the LLaMA tokenizer, respectively. Additional training and hyerparameter details are described in Appendix E.2.

Results. Using the recommended $\beta_2 = 0.95$ (Andonian et al., 2023), Table 5 summarizes the train & validation perplexity after pre-training GPT models and OpenLLaMA-7B under various options. Our COLLAGE formations are able to **match** the quality of the **best-known** baseline, FP32 MW option D, most of the time *for all models* with the only exception on the smallest GPT-125M, while having the same validation perplexity.

Ablation: Impact of β_2. We conduct ablation experiments to illustrate the impact of β_2 on the quality of precision strategies by further pre-training the GPT-125M model using $\beta_2 = 0.99$ and 0.999, with a global batchsize 1024, 2048 and the same micro-batchsize 16, as summarized in Table 6. Similar to the BERT and RoBERTa pre-training experiments, COLLAGE-light is able to closely match Option D when $\beta_2 = 0.95$ or 0.99 and remain unaffected by changes in the global batchsize.

However, with $\beta_2 = 0.999$, COLLAGE-light underperforms option D while COLLAGE-plus is still able to closely match option D. As analyzed in Section 4.2, low precision (Bfloat16) arithmetic fails to represent and compute with

Table 4. GLUE benchmark for BERT-base-uncased and RoBERTa-base pre-trained using different precision strategies. See Appendix E.1 for experimental details. BF16-COLLAGE training strategy matches/exceeds the finetuning quality over several metrics.

Model	Precision	MRPC	QNLI	SST-2	CoLA	RTE	STS-B	QQP	MNLI	Avg
BERT-base	A	0.8210	0.8832	0.8890	0.3522	0.6462	0.8666/0.8618	0.8973	0.7993	0.7796
	B (ours)	0.8431	0.8974	0.9071	0.4149	0.6606	0.8837/0.8785	0.9031	0.8184	0.8007
	C (ours)	0.8602	**0.9090**	**0.9128**	**0.4314**	0.6698	0.8851/0.8821	**0.9069**	**0.8330**	**0.8100**
	D	**0.8651**	0.9071	0.9036	0.4212	**0.6714**	**0.8890/0.8849**	0.9064	**0.8330**	0.8090
RoBERTa-base	A	0.8504	0.8914	0.9000	0.3866	0.6281	0.8636/0.8625	0.8981	0.8155	0.7884
	B (ours)	0.8455	0.9000	0.9025	0.4460	0.6281	0.8636/0.8635	0.9002	0.8182	0.7964
	C (ours)	**0.8529**	**0.9040**	**0.9048**	**0.4588**	0.6137	**0.8658/0.8647**	**0.9005**	**0.8230**	**0.7986**
	D	0.8406	0.8993	0.9002	0.3870	**0.6389**	0.8622/0.8631	0.8999	0.8203	0.7901

Table 5. **Left:** Train | Validation perplexity of pre-trained GPT with $\beta_2 = 0.95$. **Right:** OpenLLaMA-7B with $\beta_2 = 0.95$ and 0.99.

Model	GPT				OpenLLaMA-7B	
Precision Option	125M	1.3B	2.7B	6.7B	$\beta_2 = 0.95$	$\beta_2 = 0.99$
A (BF16)	14.73 | 15.64	10.28 | 12.43	9.97 | 12.18	9.87 | 12.18	6.36 | 4.81	15.96 | 12.55
B (COLLAGE-light)	14.01 | 15.03	8.50 | 17.70	8.33 | 11.36	8.17 | 11.13	5.99 | 4.53	8.00 | 5.99
C (COLLAGE-plus)	14.01 | 15.03	8.50 | 17.70	8.33 | 11.36	8.17 | 11.13	5.99 | 4.57	6.11 | 4.62
D (BF16 + FP32Optim + FP32MW)	13.87 | 15.03	8.50 | 17.70	8.33 | 11.36	8.17 | 11.13	5.99 | 4.57	8.58 | 6.42

$\beta_2 = 0.999$ due to rounding errors. In fact, we observed the same phenomenon as pre-training BERT & RoBERTa in Section 5.1, including i) a high imprecision percentage of lost additions with low-precision BF16 arithmetic; ii) a reduced EDQ for COLLAGE-light and a better EDQ for COLLAGE-plus. These together rationalize the utility and significance of our proposed metric EDQ and the necessity of COLLAGE-plus for quality models. We defer figures of these metrics for GPTs to Appendix F.3.

We also pretrain OpenLLaMA-7B with $\beta_2 = 0.99$ in Table 5 (right), where both COLLAGE formations outperform option D. In fact, we observe that $\beta_2 = 0.99$ can easily lead to gradient explosion (see Figure 6 right in Appendix F.2), while COLLAGE-plus provides stable training. The training perplexity trajectories in Figure 5,6 (in Appendix F.2) show that COLLAGE-plus effectively solves the imprecision issue and produces quality models.

Remark 5.1. The optimal choice of β_2 differs case-by-case. To our best knowledge, there is no clear conclusion between β_2 and the converged performance of the pre-trained models. Showing COLLAGE works with different β_2's, enable LLM training to be not limited by such precision issues.

5.3. Performance and Memory

Throughput. We record the mean training throughput of precision strategies for pre-training GPTs and OpenLLaMA-7B in a simple setting for fair comparisons: one *aws.p4.24xlarge* node with sequence parallel (Korthikanti et al., 2023) turned off[3], and present relative speed-up in Table 7. Both COLLAGE formations are able to maintain the efficiency of option A. Moreover, the speed factor for COLLAGE increases with an increase in the model size, obtaining up to **3.74**× for GPT-6.7B model in the considered settings.

[3]Yet we observed similar throughputs for precision strategies when sequence parallel is turned on.

Memory. We probe the peak GPU memory of all training precision strategies during practical runs on 8×NVIDIA A100s (40GB) with the same hyper-parameters for a fair comparison: sequence length 2048, global batchsize 128 and micro (per-device) batchsize 1. Figure 4 visualizes the peak memory usage of GPTs vs model sizes. During real runs, on average, COLLAGE formations (light/plus) use **23.8%/15.6%** less peak memory compared to option D. The best savings are for the largest model OpenLLaMA-7B, with savings **27.8%/18.5%**, respectively.

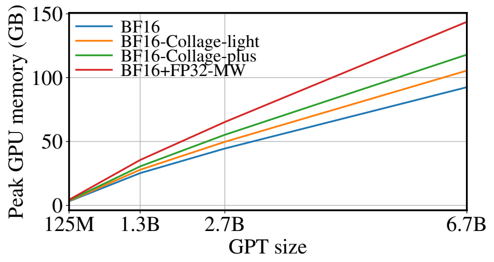

Figure 4. GPU peak memory in GB vs model size. GPT-125M is hosted on 1 NVIDIA A100 40GB, while all other models were hosted on 8× A100 40GB using tensor-parallelism 8.

Increased Sequence Length and Micro BatchSize. We study the benefits of COLLAGE's reduced memory foot-print (as shown in Figure 4), with a demonstration on pre-training a large GPT-30B model with tensor-parallelism=8, pipeline-parallelism=2 on two *aws.p4.24xlarge* (8×A100s 40GB) instances. Specifically, we identify the maximum sequence length and micro batchsize for all precision strategies to be able to run without OOM, as summarized in Table 8. COLLAGE enables training with an increased sequence length and micro batchsize compared to option D, thus providing a smooth trade-off between quality and performance.

Table 6. Train | Validation perplexity of GPT-125M pre-trained with $\beta_2 \in \{0.95, 0.99, 0.999\}$ and Global BatchSize $\in \{1024, 2048\}$.

Precision Option	Global BatchSize $= 1024$			Global BatchSize $= 2048$		
	$\beta_2 = 0.95$	$\beta_2 = 0.99$	$\beta_2 = 0.999$	$\beta_2 = 0.95$	$\beta_2 = 0.99$	$\beta_2 = 0.999$
A (BF16)	14.73 | 15.64	14.88 | 15.80	17.29 | 18.17	14.73 | 15.18	14.88 | 15.33	17.64 | 15.33
B (COLLAGE-light)	14.01 | 15.03	14.01 | 15.03	14.88 | 15.80	13.87 | 14.44	13.87 | 14.44	14.59 | 15.18
C (COLLAGE-plus)	14.01 | 15.03	14.01 | 15.03	14.15 | 15.18	13.87 | 14.44	13.87 | 14.44	14.01 | 14.59
D (BF16 + FP32Optim + FP32MW)	13.87 | 15.03	14.01 | 15.03	14.01 | 15.03	13.87 | 14.44	13.87 | 14.44	14.01 | 14.59

Table 7. Relative speed-up compared to the option D.

Precision	GPT			OpenLlama
Option	1.3B	2.7B	6.7B	7B
A	$1.78\times$	$2.59\times$	$3.82\times$	$3.15\times$
B (ours)	$1.74\times$	$2.57\times$	$3.74\times$	$3.14\times$
C (ours)	$1.67\times$	$2.48\times$	$3.57\times$	$3.05\times$
D	$1\times$	$1\times$	$1\times$	$1\times$

Table 8. Memory compatibility of pre-training GPT-NeoX-30B using precision options with different micro batchsize (UBS) and sequence length.

Precision	UBS= 1		UBS= 2	
option / SeqLen	$1,024$	$2,048$	$1,024$	$2,048$
A (BF16)	✓	✓	✓	✓
B (COLLAGE-light)	✓	✓	✓	OOM
C (COLLAGE-plus)	✓	✓	✓	OOM
D (BF16 + FP32Optim + FP32MW)	✓	OOM	OOM	OOM

Remark 5.2. Further improvements on throughput and memory can be achieved for COLLAGE with specialized fused kernels.

6. Conclusion and Future Work

We provide novel understandings on how low-precision memory and computations affect LLM training, with which, we propose a low-precision plugin COLLAGE using multiple-component floating-point. COLLAGE matches/exceeds the quality of (BF16, FP32) mixed-precision strategy with master weight while achieving an enhanced execution speed and optimized memory utilization.

COLLAGE can be used in a drop-in manner with any optimization algorithm and float-type. An interesting future work is the direct extension to even lower precision such as 8-bit FPUs, e.g., FP8, removing the usage of FP16 in (FP8, FP16) mixed-precision strategy (Peng et al., 2023). It's also intriguing to discern when COLLAGE with MCF expansions is more suitable than (BF16, FP32) mixed-precision strategy with FP32 MW, as elucidated in pretraining OpenLLaMA-7B.

Authors Contributions

TY conceived the ideas/algorithms, wrote the COLLAGE optimization code, conducted BERT/RoBERTa and GPT training experiments, and wrote the manuscript. GG conceived the ideas, conducted evaluation experiments, and wrote the manuscript. KG, AM participated in the research discussions and wrote the manuscript. HZ helped conduct OpenLLaMA-7B training experiments. JH, YP, RD, AP, LH contributed to writing the manuscript.

Acknowledgement

The authors would like to thank the reviewers for helping improving the readability of the manuscript. T.Y., G.G, and H.W. would like to acknowledge the GPU support of Amazon Sagemaker team for the Llama experiments.

Impact Statement

Our proposed COLLAGE speeds up LLM training with reduced memory usage, without hurting model performances. It can be easily integrated to the existing optimization frameworks. We believe that our method advances the field of LLM and enables efficient-training of even larger and more scalable language models with less carbon foot-print.

References

Andonian, A., Anthony, Q., Biderman, S., Black, S., Gali, P., Gao, L., Hallahan, E., Levy-Kramer, J., Leahy, C., Nestler, L., et al. GPT-NeoX: Large Scale Autoregressive Language Modeling in PyTorch, 9 2023. URL https://www.github.com/eleutherai/gpt-neox.

Attardi, G. Wikiextractor. https://github.com/attardi/wikiextractor, 2015.

Banner, R., Hubara, I., Hoffer, E., and Soudry, D. Scalable methods for 8-bit training of neural networks, 2018.

Betker, J., Goh, G., Jing, L., Brooks, T., Wang, J., Li, L., Ouyang, L., Zhuang, J., Lee, J., Guo, Y., et al. Improving image generation with better captions. *Computer Science. https://cdn. openai. com/papers/dall-e-3. pdf*, 2(3), 2023.

Chen, J., Gai, Y., Yao, Z., Mahoney, M. W., and Gonzalez, J. E. A statistical framework for low-bitwidth training of deep neural networks. *Advances in neural information processing systems*, 33:883–894, 2020.

Choi, J., Venkataramani, S., Srinivasan, V. V., Gopalakrishnan, K., Wang, Z., and Chuang, P. Accurate and efficient 2-bit quantized neural networks. In Talwalkar, A., Smith, V., and Zaharia, M. (eds.), *Proceedings of Machine Learning and Systems*, volume 1, pp. 348–359, 2019. URL https://proceedings.mlsys.org/paper_files/paper/2019/file/c443e9d9fc984cda1c5cc447fe2c724d-Paper.pdf.

Courbariaux, M., Hubara, I., Soudry, D., El-Yaniv, R., and Bengio, Y. Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or -1, 2016.

Croci, M., Fasi, M., Higham, N. J., Mary, T., and Mikaitis, M. Stochastic rounding: implementation, error analysis and applications. *Royal Society Open Science*, 9(3): 211631, 2022.

Dekker, T. J. A floating-point technique for extending the available precision. *Numer. Math.*, 18(3):224–242, jun 1971. ISSN 0029-599X. doi: 10.1007/BF01397083. URL https://doi.org/10.1007/BF01397083.

Dettmers, T. and Zettlemoyer, L. The case for 4-bit precision: k-bit inference scaling laws, 2023.

Dettmers, T., Lewis, M., Belkada, Y., and Zettlemoyer, L. Llm.int8(): 8-bit matrix multiplication for transformers at scale, 2022.

Dettmers, T., Pagnoni, A., Holtzman, A., and Zettlemoyer, L. Qlora: Efficient finetuning of quantized llms, 2023.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.

Dhariwal, P., Jun, H., Payne, C., Kim, J. W., Radford, A., and Sutskever, I. Jukebox: A generative model for music. *arXiv preprint arXiv:2005.00341*, 2020.

Frantar, E. and Alistarh, D. Sparsegpt: Massive language models can be accurately pruned in one-shot, 2023.

Frantar, E., Ashkboos, S., Hoefler, T., and Alistarh, D. Gptq: Accurate post-training quantization for generative pre-trained transformers. *arXiv preprint arXiv:2210.17323*, 2022.

Frantar, E., Ashkboos, S., Hoefler, T., and Alistarh, D. Gptq: Accurate post-training quantization for generative pre-trained transformers, 2023.

Goldberg, D. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1):5–48, mar 1991. ISSN 0360-0300. doi: 10.1145/103162.103163. URL https://doi.org/10.1145/103162.103163.

Granlund, T. Gnu mp: The gnu multiple precision arithmetic library. *http://gmplib. org/*, 2004.

Guo, H., Greengard, P., Xing, E. P., and Kim, Y. Lq-lora: Low-rank plus quantized matrix decomposition for efficient language model finetuning. *arXiv preprint arXiv:2311.12023*, 2023.

Gupta, S., Agrawal, A., Gopalakrishnan, K., and Narayanan, P. Deep learning with limited numerical precision, 2015.

Han, S., Mao, H., and Dally, W. J. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

Hida, Y., Li, S., and Bailey, D. Library for double-double and quad-double arithmetic. 01 2008.

Hinton, G., Vinyals, O., and Dean, J. Distilling the knowledge in a neural network, 2015.

Hoffmann, J., Borgeaud, S., Mensch, A., Buchatskaya, E., Cai, T., Rutherford, E., de Las Casas, D., Hendricks, L. A., Welbl, J., Clark, A., et al. Training compute-optimal large language models, 2022.

Hou, L., Zhang, R., and Kwok, J. T. Analysis of quantized models. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=ryM_IoAqYX.

Hsieh, C.-Y., Li, C.-L., Yeh, C.-K., Nakhost, H., Fujii, Y., Ratner, A., Krishna, R., Lee, C.-Y., and Pfister, T. Distilling step-by-step! outperforming larger language models with less training data and smaller model sizes, 2023.

Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., and Chen, W. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021.

Jacob, B., Kligys, S., Chen, B., Zhu, M., Tang, M., Howard, A., Adam, H., and Kalenichenko, D. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 2704–2713, 2018.

Jia, Z. and Sandt, P. V. Zhe jia and peter van sandt. dissecting the ampere gpu architecture via microbenchmarking. gpu technology conference, 2021. In *GTC*, 2021.

Jia, Z., Maggioni, M., Staiger, B., and Scarpazza, D. P. Dissecting the nvidia volta gpu architecture via microbenchmarking, 2018.

Kahan, W. How futile are mindless assessments of roundoff in floating-point computation. *Preprint, University of California, Berkeley*, 2006.

Kalamkar, D., Mudigere, D., Mellempudi, N., Das, D., Banerjee, K., Avancha, S., Vooturi, D. T., Jammalamadaka, N., Huang, J., Yuen, H., et al. A study of bfloat16 for deep learning training, 2019.

Korthikanti, V. A., Casper, J., Lym, S., McAfee, L., Andersch, M., Shoeybi, M., and Catanzaro, B. Reducing activation recomputation in large transformer models. *Proceedings of Machine Learning and Systems*, 5, 2023.

Kuchaiev, O., Ginsburg, B., Gitman, I., Lavrukhin, V., Li, J., Nguyen, H., Case, C., and Micikevicius, P. Mixed-precision training for nlp and speech recognition with openseq2seq, 2018.

Kuchaiev, O., Li, J., Nguyen, H., Hrinchuk, O., Leary, R., Ginsburg, B., Kriman, S., Beliaev, S., Lavrukhin, V., Cook, J., et al. Nemo: a toolkit for building ai applications using neural modules, 2019.

Kurtic, E., Campos, D., Nguyen, T., Frantar, E., Kurtz, M., Fineran, B., Goin, M., and Alistarh, D. The optimal bert surgeon: Scalable and accurate second-order pruning for large language models, 2022.

Kwon, S. J., Kim, J., Bae, J., Yoo, K. M., Kim, J.-H., Park, B., Kim, B., Ha, J.-W., Sung, N., and Lee, D. Alphatuning: Quantization-aware parameter-efficient adaptation of large-scale pre-trained language models. *arXiv preprint arXiv:2210.03858*, 2022.

Lagunas, F., Charlaix, E., Sanh, V., and Rush, A. M. Block pruning for faster transformers. *arXiv preprint arXiv:2109.04838*, 2021.

Le, M., Vyas, A., Shi, B., Karrer, B., Sari, L., Moritz, R., Williamson, M., Manohar, V., Adi, Y., Mahadeokar, J., and Hsu, W.-N. Voicebox: Text-guided multilingual universal speech generation at scale, 2023.

Li, H., De, S., Xu, Z., Studer, C., Samet, H., and Goldstein, T. Training quantized nets: A deeper understanding, 2017.

Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L., and Stoyanov, V. Roberta: A robustly optimized bert pretraining approach, 2019.

Loshchilov, I. and Hutter, F. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.

Micikevicius, P., Narang, S., Alben, J., Diamos, G., Elsen, E., Garcia, D., Ginsburg, B., Houston, M., Kuchaiev, O., Venkatesh, G., et al. Mixed precision training. *arXiv preprint arXiv:1710.03740*, 2017.

Micikevicius, P., Narang, S., Alben, J., Diamos, G., Elsen, E., Garcia, D., Ginsburg, B., Houston, M., Kuchaiev, O., Venkatesh, G., and Wu, H. Mixed precision training, 2018.

Micikevicius, P., Stosic, D., Burgess, N., Cornea, M., Dubey, P., Grisenthwaite, R., Ha, S., Heinecke, A., Judd, P., Kamalu, J., et al. Fp8 formats for deep learning, 2022.

Muller, J.-M., Brisebarre, N., De Dinechin, F., Jeannerod, C.-P., Lefevre, V., Melquiond, G., Revol, N., Stehlé, D., Torres, S., et al. *Handbook of floating-point arithmetic*. Springer, 2018.

OpenAI, :, Achiam, J., Adler, S., Agarwal, S., Ahmad, L., Akkaya, I., Aleman, F. L., Almeida, D., Altenschmidt, J., et al. Gpt-4 technical report, 2023.

Park, H., Lee, J. H., Oh, Y., Ha, S., and Lee, S. Training deep neural network in limited precision, 2018.

Peng, H., Wu, K., Wei, Y., Zhao, G., Yang, Y., Liu, Z., Xiong, Y., Yang, Z., Ni, B., Hu, J., et al. Fp8-lm: Training fp8 large language models, 2023.

Perez, S. P., Zhang, Y., Briggs, J., Blake, C., Levy-Kramer, J., Balanca, P., Luschi, C., Barlow, S., and Fitzgibbon, A. W. Training and inference of large language models using 8-bit floating point, 2023.

Priest, D. Algorithms for arbitrary precision floating point arithmetic. In *[1991] Proceedings 10th IEEE Symposium on Computer Arithmetic*, pp. 132–143, 1991. doi: 10.1109/ARITH.1991.145549.

Priest, D. M. *On properties of floating point arithmetics: numerical stability and the cost of accurate computations*. PhD thesis, University of California at Berkeley, USA, 1992. UMI Order No. GAX93-30692.

Rae, J. W., Borgeaud, S., Cai, T., Millican, K., Hoffmann, J., Song, F., Aslanides, J., Henderson, S., Ring, R., Young, S., et al. Scaling language models: Methods, analysis & insights from training gopher, 2022.

Rastegari, M., Ordonez, V., Redmon, J., and Farhadi, A. Xnor-net: Imagenet classification using binary convolutional neural networks, 2016.

Ruder, S. An overview of gradient descent optimization algorithms, 2017.

Sa, C. D., Leszczynski, M., Zhang, J., Marzoev, A., Aberger, C. R., Olukotun, K., and Ré, C. High-accuracy low-precision training, 2018.

Sakr, C. and Shanbhag, N. Per-tensor fixed-point quantization of the back-propagation algorithm, 2018.

Sanh, V., Debut, L., Chaumond, J., and Wolf, T. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter, 2020.

Shoeybi, M., Patwary, M., Puri, R., LeGresley, P., Casper, J., and Catanzaro, B. Megatron-lm: Training multi-billion parameter language models using model parallelism, 2020.

Su, J., Ahmed, M., Lu, Y., Pan, S., Bo, W., and Liu, Y. Roformer: Enhanced transformer with rotary position embedding. *Neurocomputing*, 568:127063, 2024.

Sun, X., Choi, J., Chen, C.-Y., Wang, N., Venkataramani, S., Srinivasan, V. V., Cui, X., Zhang, W., and Gopalakrishnan, K. Hybrid 8-bit floating point (hfp8) training and inference for deep neural networks. *Advances in neural information processing systems*, 32, 2019a.

Sun, X., Choi, J., Chen, C.-Y., Wang, N., Venkataramani, S., Srinivasan, V. V., Cui, X., Zhang, W., and Gopalakrishnan, K. Hybrid 8-bit floating point (hfp8) training and inference for deep neural networks. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019b. URL https://proceedings.neurips.cc/paper_files/paper/2019/file/65fc9fb4897a89789352e211ca2d398f-Paper.pdf.

Team, G., Anil, R., Borgeaud, S., Wu, Y., Alayrac, J.-B., Yu, J., Soricut, R., Schalkwyk, J., Dai, A. M., Hauth, A., et al. Gemini: A family of highly capable multimodal models, 2023.

Thoppilan, R., De Freitas, D., Hall, J., Shazeer, N., Kulshreshtha, A., Cheng, H.-T., Jin, A., Bos, T., Baker, L., Du, Y., et al. Lamda: Language models for dialog applications. *arXiv preprint arXiv:2201.08239*, 2022.

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., et al. Llama: Open and efficient foundation language models, 2023.

Wang, A., Singh, A., Michael, J., Hill, F., Levy, O., and Bowman, S. R. Glue: A multi-task benchmark and analysis platform for natural language understanding, 2019.

Wang, N., Choi, J., Brand, D., Chen, C.-Y., and Gopalakrishnan, K. Training deep neural networks with 8-bit floating point numbers. In *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018a. URL https://proceedings.neurips.cc/paper_files/paper/2018/file/335d3d1cd7ef05ec77714a215134914c-Paper.pdf.

Wang, N., Choi, J., Brand, D., Chen, C.-Y., and Gopalakrishnan, K. Training deep neural networks with 8-bit floating point numbers. *Advances in neural information processing systems*, 31, 2018b.

Wei, X., Gonugondla, S., Ahmad, W., Wang, S., Ray, B., Qian, H., Li, X., Kumar, V., Wang, Z., Tian, Y., et al. Greener yet powerful: Taming large code generation models with quantization, 2023.

Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Louf, R., Funtowicz, M., et al. Huggingface's transformers: State-of-the-art natural language processing. *arXiv preprint arXiv:1910.03771*, 2019.

Wortsman, M., Dettmers, T., Zettlemoyer, L., Morcos, A., Farhadi, A., and Schmidt, L. Stable and low-precision training for large-scale vision-language models, 2023.

Wu, S., Li, G., Chen, F., and Shi, L. Training and inference with integers in deep neural networks, 2018.

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

Xia, H., Zheng, Z., Li, Y., Zhuang, D., Zhou, Z., Qiu, X., Li, Y., Lin, W., and Song, S. L. Flash-llm: Enabling cost-effective and highly-efficient large generative model inference with unstructured sparsity. *arXiv preprint arXiv:2309.10285*, 2023.

Xia, M., Zhong, Z., and Chen, D. Structured pruning learns compact and accurate models. *arXiv preprint arXiv:2204.00408*, 2022.

Xiao, G., Lin, J., Seznec, M., Wu, H., Demouth, J., and Han, S. Smoothquant: Accurate and efficient post-training quantization for large language models, 2023.

Xu, Y., Xie, L., Gu, X., Chen, X., Chang, H., Zhang, H., Chen, Z., Zhang, X., and Tian, Q. Qa-lora: Quantization-aware low-rank adaptation of large language models. *arXiv preprint arXiv:2309.14717*, 2023.

Yao, Z., Wu, X., Li, C., Youn, S., and He, Y. Zeroquant-v2: Exploring post-training quantization in llms from comprehensive study to low rank compensation. *arXiv preprint arXiv:2303.08302*, 2023.

Yu, F., Huang, K., Wang, M., Cheng, Y., Chu, W., and Cui, L. Width & depth pruning for vision transformers. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pp. 3143–3151, 2022a.

Yu, T. and De Sa, C. M. Representing hyperbolic space accurately using multi-component floats. In Ranzato, M., Beygelzimer, A., Dauphin, Y., Liang, P., and Vaughan, J. W. (eds.), *Advances in Neural Information Processing Systems*, volume 34, pp. 15570–15581. Curran Associates, Inc., 2021. URL https://proceedings.neurips.cc/paper_files/paper/2021/file/832353270aacb6e3322f493a66aaf5b9-Paper.pdf.

Yu, T., Guo, W., Li, J. C., Yuan, T., and Sa, C. D. Mctensor: A high-precision deep learning library with multi-component floating-point, 2022b.

Zamirai, P., Zhang, J., Aberger, C. R., and De Sa, C. Revisiting bfloat16 training. *arXiv preprint arXiv:2010.06192*, 2020.

Zhang, S., Roller, S., Goyal, N., Artetxe, M., Chen, M., Chen, S., Dewan, C., Diab, M., Li, X., Lin, X. V., et al. Opt: Open pre-trained transformer language models, 2022.

Zhou, A., Ma, Y., Zhu, J., Liu, J., Zhang, Z., Yuan, K., Sun, W., and Li, H. Learning n:m fine-grained structured sparse neural networks from scratch, 2021.

A. Floating point units

Floating-point representation uses a sign bit to indicate positive or negative numbers, an exponent to determine scale, and a mantissa for significant digits, enabling efficient handling of a wide range of numbers with potential for precision errors. Different floating-point formats offer varying benefits and trade-offs. Single Precision (FP32) provides wide range and reasonable precision, while consuming more resources. Half Precision (FP16) reduces memory usage and improves efficiency, but sacrifices precision and range. Brain floating point (BF16) as another 16-bit format has a much bigger dynamic range (same as FP32), while having a worse precision than FP16. FP8 (two versions) could further reduce resources, suitable for constrained environments, but with even more limited precision and range.

We present different formats referenced in the paper along with their exponent and mantissa bits.

Table 9. Floating-Point Precisions and ULPs

Precision	#Exponent bits	#Mantissa (significand) bits	$\text{ulp}(1)$
Single (FP32)	8	23	2^{-23}
Half (FP16)	5	10	2^{-10}
BF16	8	7	2^{-7}
FP8 E4M3	4	3	2^{-3}
FP8 E5M2	5	2	2^{-2}

B. Related Work

Low Precision and Quantization-aware Training. Fully quantized training attempts to downscale numerical precisions but not to compromise accuracy, mainly for large-scale training, using FP16 (Micikevicius et al., 2017), BF16 (Kalamkar et al., 2019), FP8 (Wang et al., 2018b; Sun et al., 2019a), and other combination of integer types (Banner et al., 2018; Chen et al., 2020). Micikevicius et al. (2018) developed a mixed precision strategy that maintains master weight in FP32 only whereas others are in lower precision of FP16. Xi et al. (2023) recently proposed a training method using INT4 but without customized data types, compatitable with contemporary hardwares. In parallel, Peng et al. (2023) proposed a new mixed-precision strategy, gradually incorporating 8-bit gradients, optimizer states in an incremental manner, under distributed settings. When it comes to fine-tuning setings, LoRA (Hu et al., 2021) leverage structure of matrix to update in low-rank. Dettmers et al. (2023); Xu et al. (2023); Guo et al. (2023) proposed various variants of LoRA more in memory and computationally efficient manners. Overall, these works develop training strategies based on numerical structures like low-rank over attention matrices and/or sparsity over parameters in each layer, numerical scale of each variables used for gradient updates. However, they lack of thorough diagnosis on imprecision errors, which has been depriving potential algorithmic developments in numeric precision levels. Zamirai et al. (2020) proposed to adopt Kahan summation (Kahan, 2006) and stochastic rounding (SR) (Croci et al., 2022) to alleviate the influence of imprecision and lost arithmetic at the model parameter update step.

Pruning and Distillation. Pruning (Han et al., 2015; Kurtic et al., 2022) removes redundant parameters from the network. The goal is to maintain prediction quality of the model while shrinking its size, and therewith increasing its efficiency. distillation (Hsieh et al., 2023; Hinton et al., 2015; Hsieh et al., 2023) transfers knowledge from a large model to a smaller one. Pruning can be combined with distillation approach to further reduce model parameters (Sanh et al., 2020; Lagunas et al., 2021; Xia et al., 2022). Structured pruning removes whole components of the network such as neurons, heads, and layers (Yu et al., 2022a; Lagunas et al., 2021; Zhou et al., 2021). Unstructured pruning removes individual weights of the network with smaller magnitudes (Frantar & Alistarh, 2023; Xia et al., 2023). Albeit these are useful in reducing computational overhead, distillation and pruning requires either the model already trained as post-hoc method, architecture change than original models or iterative procedures that potentially take longer in an end-to-end manner.

Post-training Quantization. Quantization compresses the representation of the parameters into low-precision data types, reducing the storage when loading the model in devices. Post-training quantization methods quantize the parameters of the pre-trained model (Yao et al., 2023; Xiao et al., 2023; Frantar et al., 2022) often with fine-tuning steps (Kwon et al., 2022). (Jacob et al., 2018) emulates inference-time quantization, creating a model that can be quantized later post-training . However, these works mostly focus on faster inference, rather than reducing end-to-end training time.

Algorithm 2 TwoSum

```
1: Input: $P$-bit floats $a$ and $b$
2: $x \leftarrow \mathcal{F}^P(a \oplus b)$
3: $b_{\text{virtual}} \leftarrow \mathcal{F}^P(x \ominus a)$
4: $a_{\text{virtual}} \leftarrow \mathcal{F}^P(x \ominus b_{\text{virtual}})$
5: $b_{\text{roundoff}} \leftarrow \mathcal{F}^P(b \ominus b_{\text{virtual}})$
6: $a_{\text{roundoff}} \leftarrow \mathcal{F}^P(a \ominus a_{\text{virtual}})$
7: $y \leftarrow \mathcal{F}^P(a_{\text{roundoff}} \oplus b_{\text{roundoff}})$
8: Return: $(x, y)$
```

Algorithm 3 Split

```
1: Input: $P$-bit float $a$ (with $p$-bit mantissa)
2: $c \leftarrow \lfloor \frac{p}{2} \rfloor$
3: $t \leftarrow \mathcal{F}^P(2^c \oplus 1) \cdot a$
4: $a_{hi} \leftarrow \mathcal{F}^P(t \ominus \mathcal{F}^P(t \ominus a))$
5: $a_{lo} \leftarrow \mathcal{F}^P(a \ominus a_{hi})$
6: Return: $(a_{hi}, a_{lo})$
```

Algorithm 4 TwoProd

```
1: Input: $P$-bit floats $a$ and $b$
2: $x \leftarrow \mathcal{F}^P(a \odot b)$
3: $(a_{hi}, a_{lo}) \leftarrow$ Split$(a)$
4: $(b_{hi}, b_{lo}) \leftarrow$ Split$(b)$
5: $err_1 \leftarrow \mathcal{F}^P(p \ominus \mathcal{F}^P(a_{hi} \odot b_{hi}))$
6: $err_2 \leftarrow \mathcal{F}^P(err_1 \ominus \mathcal{F}^P(a_{lo} \odot b_{hi}))$
7: $err_3 \leftarrow \mathcal{F}^P(err_2 \ominus \mathcal{F}^P(a_{hi} \odot b_{lo}))$
8: $e \leftarrow \mathcal{F}^P(\mathcal{F}^P(a_{lo} \odot b_{lo}) \ominus err_3)$
9: Return: $(x, e)$
```

Algorithm 5 TwoProdFMA

```
1: Input: $P$-bit floats $a$ and $b$
2: Requires: Machine supports FMA
3: $x \leftarrow \mathcal{F}^P(a \odot b)$
4: $e \leftarrow \mathcal{F}^P(a \odot b \ominus x)$ in FMA
5: Return: $(x, e)$
```

Algorithm 6 Scaling

```
1: Input: a float $v$ and a length-2 expansion $(a_1, a_2)$
2: $(x, e) \leftarrow$ TwoProdFMA$(a_1, v)$
3: $e \leftarrow \mathcal{F}^P(a_2 \odot v \oplus e)$
4: $(x, e) \leftarrow$ Fast2Sum$(x, e)$
5: Return: $(x, e)$
```

Algorithm 7 Mul

```
1: Input: length-2 expansions $(a_1, a_2)$ and $(b_1, b_2)$
2: $(x, e) \leftarrow$ TwoProdFMA$(a_1, b_1)$
3: $e \leftarrow \mathcal{F}^P(e \oplus ((a_1 \odot b_2) \oplus (a_2 \odot b_1)))$
4: $(x, e) \leftarrow$ Fast2Sum$(x, e)$
5: Return: $(x, e)$
```

Kahan Summation. The Kahan summation is a standard algorithm in numerical analysis for accurate summation of floating-point numbers, just like the case of adding updates to the parameter. When incorporated with optimization algorithms such as SGD and AdamW, the Kahan algorithm introduces an auxiliary Kahan variable $\boldsymbol{c}$ (in the same precision) to track numerical errors at the parameter update step (i.e., $\boldsymbol{\theta}_t \leftarrow \mathcal{F}^P(\boldsymbol{\theta}_{t-1} \oplus \boldsymbol{\Delta\theta}_t)$) with $\boldsymbol{c} \leftarrow \mathcal{F}^P\left(\boldsymbol{\Delta\theta}_t \ominus \mathcal{F}^P(\boldsymbol{\theta}_t \ominus \boldsymbol{\theta}_{t-1})\right)$, and to compensate the addition results by adding $\boldsymbol{c}$ to the next iteration update: $\boldsymbol{\Delta\theta}_{t+1} \leftarrow \mathcal{F}^P(\boldsymbol{\Delta\theta}_{t+1} \oplus \boldsymbol{c})$. The Kahan variable $\boldsymbol{c}$ accumulates lost small updates until it grows large enough to be added with the model parameters. As pointed in (Zamirai et al., 2020), "16-bit-FPU training with Kahan summation for model weight updates have advantages in terms of throughput and memory consumption compared to 32-bit and mixed precision training", despite of the additional auxiliary value.

Stochastic Rounding. Different from the deterministic rounding-to-the-nearest behavior, stochastic rounding rounds the number up and down in a probabilistic way. For any $x \in \mathbb{R}$, assume $a_u, a_l \in \mathbb{R}$ be the closest upper and lower neighboring floating-point values of x, i.e., $a_l \leq x < a_u = a_l + \text{ulp}(a_l)$, then $\text{SR}(x) = a_l$ with probability $(a_u - x)/(a_u - a_l) = 1 - (x - a_l)/\,\text{ulp}(a_l)$ and otherwise rounds up to a_u. Stochastic rounding provides an unbiased estimate of the precise value: $\mathbb{E}\left[\text{SR}(x)\right] = x$ and alleviates the influence of imprecision by making the addition valid in expectation. Stochastic rounding for model weight updates adds minimal overhead for training and is supported in modern hardwares, such as AWS Trainium instances.

C. MCF algorithms

As noted in Theorem 4.1, `Fast2Sum` requires $|a| > |b|$ so as to perform the arithmetic correctly. One can also derive the same length-2 expansion using `Two-Sum` in Algorithm 2 for any floats a, b without sorting.

Another category of basic MCF algorithms is the multiplication, between i) a float and a float (with `TwoProd` Algorithm 4); ii) a float and a length-2 expansion (with `Scaling` Algorithm 6); iii) an expansion and an expansion (with `Mul` Algorithm 7), to produce length-2 expansions.

Case i). `TwoProd` computes the expansion using another basic algorithm `Split` (Algorithm 3), which takes a single P-bit floating point value and splits it into its high and low parts, both with $\frac{P}{2}$ bits. On a machine which supports the fused-multiply-add (FMA) instruction set, a much more efficient version `TwoProdFMA` Algorithm 5 can be adopted to give the same results. We utilized this efficient `TwoProdFMA` in our implementations as (Bfloat16) FMA is supported on CUDA, e.g., using `torch.addcmul`$(-x, a, b)$.

Case ii) and iii). Algorithm 6 `Scaling` describes the multiplication of a single float with a length-2 expansion and Algorithm 7 `Mul` the multiplication between 2 length-2 expansions. With FMA enabled, both algorithms run efficiently.

We refer the readers to (Yu et al., 2022b) for a full list of MCF algorithms.

D. Further Discussions on Algorithms

FP8 Backends and Implementations. Existing FP8 backends such as transformer engine and MS-AMP adopt FP8 mainly in the forward (parameter θ_t) & backward (gradient g_t) computations. For example, with the FP8-hybrid recipe, FP8_E4M3 (4 bits for exponent and 3 for mantissa) is adopted in the forward and FP8_E5M2 in backward pass. It is important to note that, both backends require higher-precision (FP16 with scale factors or FP32) master weights to be used as the optimizer-update parameters in a given optimizer, (for example, AdamW). In particular, MS-AMP uses FP8 datatype for the storage of first momentum m_t and FP16 for the second momentum v_t, as they have observed that FP8 second momentum leads to degraded performances (Peng et al., 2023).

During the optimization computation, all low-precision variables including FP8 gradients g_t, FP8 1st-momentum m_t, FP16 master weights and FP16 2nd momentum v_t are first upcasted to FP32 variables, which are then used to compute FP32 updates $\Delta\theta_t$ following the optimizer algorithm. The FP32 variables (momentums and parameter updates) are then downcasted at the end to the given low-precision FP8 to update their low-precision counterparts. The total bytes per parameter of MS-AMP's FP16-FP8 approach is 7 bytes with the optimizer computations upcasting to FP32 at the run-time.

Since the above discussed strategy of MS-AMP is an Option-D (BF16+FP32) counterpart of FP8+FP16, we can easily adopt our presented collage in Algorithm-2 without any algorithmic changes as follows.

- Similarly to BF16-Collage, FP8-Collage first removes the FP16 master weight, but augments the FP8 parameter θ_t with another FP8 $\delta\theta_t$ component.
- Collage-light adopts FP8 datatype for both momentums.
- Collage-plus further augments the 2nd-momentum v_t with another FP8 δv_t component.

The rest of the algorithm stays the same as Algorithm 2 in our paper, using MCF algorithms Grow (Alg. 1) and Mul (Alg. 7). The total bytes per parameter of Collage-(light/plus) is 5/6 bytes, respectively.

Equivalence. The equivalence of using 'Kahan-sum in the optimizer' at the model-update step and COLLAGE-light is straightforward, realizing i) the Kahan variable $\boldsymbol{c}$ calculation is essentially `Fast2Sum` given $|\boldsymbol{\theta}_t| \geq |\Delta\boldsymbol{\theta}_{t-1}|$; and ii) next iteration update $\boldsymbol{\Delta\theta}_{t+1}$ has similar magnitude as $\boldsymbol{c}$ so that lost arithmetic doesn't happen. In contrast, COLLAGE-light doesn't have such concerns using `Grow`.

Weight Decay. (Loshchilov & Hutter, 2017) propose AdamW with the decoupled weight decay placed at line 12 in Algorithm 2 for a summed update $\boldsymbol{\Delta\theta}_t$, standard libraries including PyTorch and HuggingFace however implement the decoupled weight decay directly to the parameter:

$$\boldsymbol{\theta}_t \leftarrow \boldsymbol{\theta}_{t-1} - \alpha\lambda\boldsymbol{\theta}_{t-1}, \quad \text{or} \quad \boldsymbol{\theta}_t \leftarrow (1-\alpha\lambda)\boldsymbol{\theta}_{t-1} \tag{4}$$

which works as expected using Float32, but is usually ineffective in Bfloat16 arithmetic due to imprecision and lost arithmetic. For example, a standard choice of the learning rate and weight decay hyper-parameter in GPT-6.7B pretraining is $\alpha = 1.2e-4$ and $\lambda = 0.1$, yielding $\alpha\lambda = 1.2e-5$ and causing lost arithmetic in Equation 4 when Bfloat16 is used. In fact, the least $\alpha\lambda$ value to avoid invalid arithmetic is half ulp(1.0), i.e., $2^{-7}/2 \approx 0.0039$. Either decaying the parameter (expansion) with `Grow` or placing the decoupled weight decay term at line 13 following the original AdamW algorithm statement solves the issue, where we chose the latter option in our experiments.

Scalar and Bias Correction. A rule of thumb to avoid imprecision and lost arithmetic during low precision training is to do as many scalar computations in high precision as possible before casting them to low precision (e.g., PyTorch BFloat16 Tensor). For example, in BFloat16 AdamW, it's recommended to compute the bias correction scalar terms $1-\beta_1^t$ and $1-\beta_2^t$ in high precision before dividing the low precision momentums.

Table 10. Pre-training hyperparameters used for BERT and RoBERTa.

Model	Phase	hyperparameters	Values
BERT-base	Phase-1	iterations	$28,125$
		warmup steps	$2,000$
		sequence length	128
		global batch size	$16,384$
		learning rate	$4e-4$
		(β_1, β_2)	$(0.9, 0.999)$
	Phase-2	iterations	$28,125$
		warmup steps	$2,000$
		sequence length	512
		global batch size	$32,768$
		learning rate	$2.8e-4$
		(β_1, β_2)	$(0.9, 0.999)$
RoBERTa-base	Phase-1	iterations	$28,125$
		warmup steps	$2,000$
		sequence length	512
		global batch size	$8,192$
		learning rate	$lr = 6e-4$
		(β_1, β_2)	$(0.9, 0.98)$

E. Experiments details

E.1. BERT and RoBERTa

We pretrain the BERT-base-uncased, BERT-large-uncased and RoBERTa-base model from HuggingFace (Wolf et al., 2019) on the Wikipedia-en corpus (Attardi, 2015), preprocessed with BERT tokenizer. We follow the standard pipeline to pretrain BERT and RoBERTa with the same configs and hyper-parameters for all precision strategies. Note that we took these configs and hyper-parameters from open-sourced models in HuggingFace. We finetuned the pretrained BERT and RoBERTa models following (Wang et al., 2019) with BF16 mixed precision through HuggingFace and evaluated the final model on GLUE benchmarks. Particularly, we used $2e-5$ learning rate and a batch size of 32 evaluated on single Nvidia A100. All tasks were finetuned for 3 epochs, apart from MRPC which we ran for 5 epochs.

E.2. Multi-size GPTs & OpenLLaMA-7B

We conduct pretraining experiments of 1) GPT at different sizes ranging from 125M, 1.3B, 2.7B to 6.7B, and 2) OpenLLaMA-7B using NeMo Megatron (Kuchaiev et al., 2019) with provided standard configs, both on the Wikipedia corpus with HuggingFace GPT2 and LLaMA tokenizer, respectively. We split the dataset into train/val/test with the split ratio $980 : 10 : 10$. We trained all models consistently with disabled sequence parallelism, enabled flash attention, rotary positional embedding (of percentage 1.0) (Su et al., 2024), disabled transformer engine, untied embeddings & output weights, sequence length of $2,048$, weight decay 0.1 and pipeline parallelism 1, for all GPT models and OpenLLaMA-7B in our experiments unless otherwise specified. All models were trained with the CosineAnnealing learning rate scheduler with 200 warmup iterations. We trained all GPT models for 20k iterations and OpenLLaMA for 9k iterations due to timing constraints. The dafault value of βs are $\beta_1 = 0.9$ and $\beta_2 = 0.95$ unless otherwise specified, e.g., in ablation experiments. Note that we took these configs from EleutherAI/gpt-neox (Andonian et al., 2023).

Table 11. Some configs and hyper-parameters of GPT models and OpenLLaMA-7B.

Model	#Layers	HiddenSize	#AttentionHeads	Global BatchSize	TensorParallelism	lr
GPT-125M	12	768	12	$1,024$	1	$6e-4$
GPT-1.3B	24	2048	16	$1,024$	8	$2e-4$
GPT-2.7B	32	2560	32	512	8	$1.6e-4$
GPT-6.7B	32	4096	32	256	8	$1.2e-4$
OpenLLaMA-7B	32	4096	32	256	8	$3e-4$

GPT-30B. For the GPT-30B model used in Section 5.3, it has 56 layers, hidden size 7168 and 56 attention heads. We trained it with a global batchsize 256, tensor parallelism 8 and pipeline parallelism 2, then varied the micro batchsize and sequence length to explore their maxium values without causing OOM on a NVIDIA A100 cluster with 2 nodes, 8 GPUs each.

F. Additional Results

F.1. Memory Statistics

Table 12 summarizes the peak (total) memory of all training precision strategies during practical runs with the same hyper-parameters for a fair comparison: Sequence Length 2048, Global BatchSize 128 and Micro (per-device) BatchSize 1. We trained GPTs and OpenLLaMA with TensorParallelism 8 over 8×A100 40GB, except from GPT-125M with TensorParallelism 1 on 1×A100 40GB. In Table 12, we report the saved memory compared to the mixed-precision option D with the percentage calculated. During real runs, on average, COLLAGE formations (light/plus) use **23.8%/15.6%** less peak memory compared to option D. The best savings are for largest model OpenLLaMA-7B, with savings **27.8%/18.5%**, respectively. The memory savings match the theoretical calculation in Table 2.

Table 12. Peak (saved) pretraining memory (GB) of precision strategies compared to option D on GPTs and OpenLLaMA-7B.

Precision	GPT				OpenLLaMA
Option	125M	1.3B	2.7B	6.7B	7B
A	$-1.1(-26.6\%)$	$-10.3(-28.9\%)$	$-20.8(-31.2\%)$	$-51.2(-35.6\%)$	$-65.7(-37.2\%)$
B (ours)	$-0.8(-19.3\%)$	$-7.6(-21.5\%)$	$-15.6(-23.8\%)$	$-38.2(-26.6\%)$	$-49.2(-27.8\%)$
C (ours)	$-0.5(-12.1\%)$	$-5.0(-14.1\%)$	$-10.1(-15.4\%)$	$-25.7(-17.9\%)$	$-32.7(-18.5\%)$
D	4.4	35.5	65.3	143.7	176.7

F.2. OpenLLama 7B pretraining

We provide the pretraining iterations progress for OpenLLama-7B (described in Section 5.2) in the Figure 5, 6 for $\beta_2 = 0.95, 0.99$, respectively. We observe a stable training using COLLAGE-plus when using $\beta_2 = 0.99$, where other precision strategies show slow convergence. The gradient norm in Figure 6 left show that COLLAGE-plus has stability while other precision strategies encounter transient errors causing blow-ups in gradients.

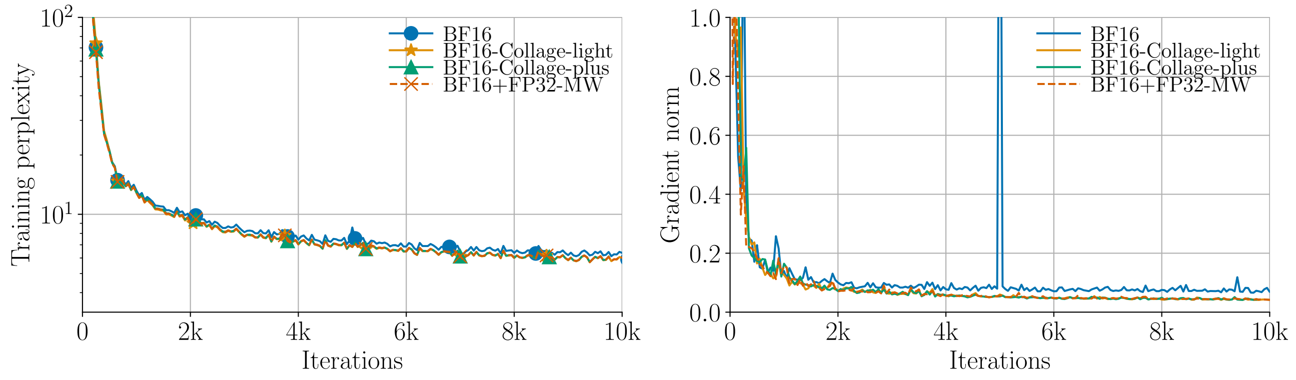

Figure 5. Openllama 7B pretraining (see settings in Section 5.2) with $\beta_2 = 0.95$. **Left:** Training perplexity for different precision strategies listed in Table 2. **Right:** Model gradient L2 norm across iterations for different strategies. The COLLAGE formations overlap with heavy-weighted FP32 master weights strategy.

F.3. GPT pretraining

The pretraining progress of GPT 125M for various settings of β_2 and global batch sizes is provided in Figure 7 8 9 10 11 12. For pretraining of GPT 1.3B, see Figure 13. For pretraining of 2.7B, see Figure 14. For pretraining of 6.7B, see Figure 15.

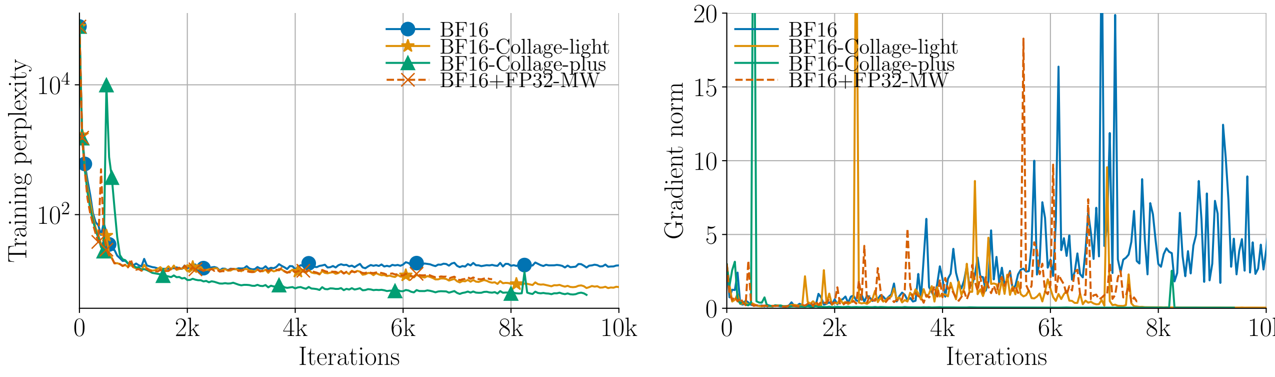

Figure 6. Openllama 7B pretraining (see settings in Section 5.2) with $\beta_2 = 0.99$. **Left:** Training perplexity for different precision strategies listed in Table 2. **Right:** Model gradient L2 norm across iterations for different strategies. The COLLAGE-plus results in the best train perplexity over iterations while other approaches struggle. The gradient norm blows-up frequently but stays stable for COLLAGE-plus which suggest importance of using multi-components at critical locations.

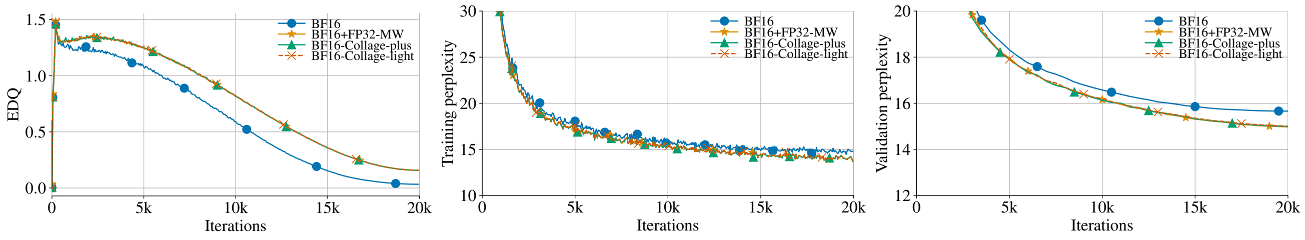

Figure 7. Pretrainnig progress for GPT 125M with settings described in Section 5.2 and global batch-size=1024, $\beta_2 = 0.95$. **Top-left:** EDQ metric vs iterations, **top-right:** training perplexity vs iterations, and **bottom:** validation perplexity vs iterations for different precision strategy listed in Table 2. The proposed COLLAGE formations consistently match the FP32 master weights with much less memory footprint and faster training.

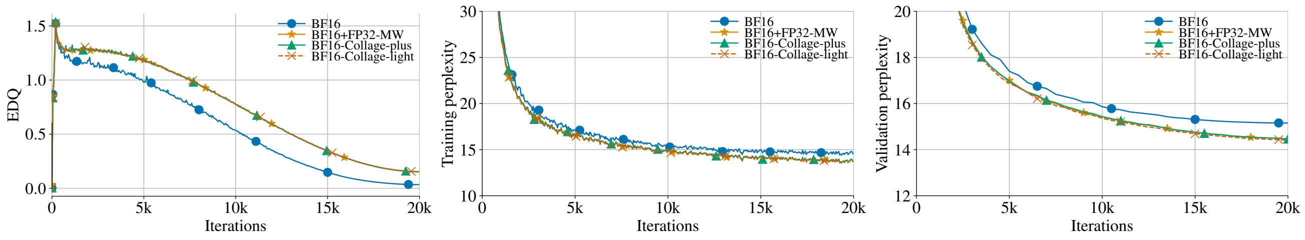

Figure 8. Pretrainnig progress for GPT 125M with settings described in Section 5.2 and global batch-size=2048, $\beta_2 = 0.95$. **Top-left:** EDQ metric vs iterations, **top-right:** training perplexity vs iterations, and **bottom:** validation perplexity vs iterations for different precision strategy listed in Table 2. The proposed COLLAGE formations consistently match the FP32 master weights with much less memory footprint and faster training.

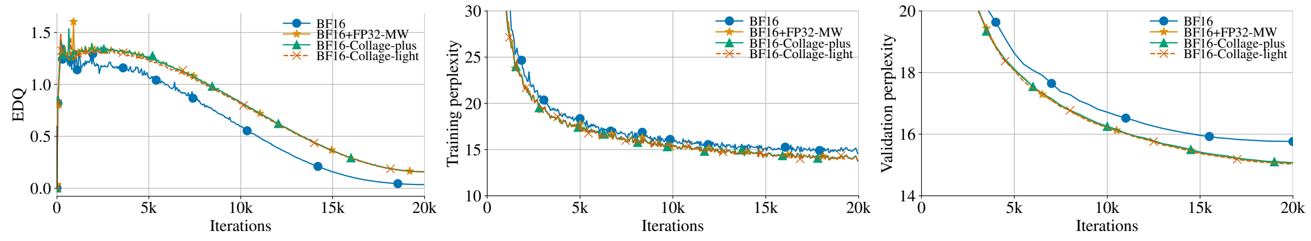

Figure 9. Pretrainnig progress for GPT 125M with settings described in Section 5.2 and global batch-size=1024, $\beta_2 = 0.99$. **Top-left:** EDQ metric vs iterations, **top-right:** training perplexity vs iterations, and **bottom:** validation perplexity vs iterations for different precision strategy listed in Table 2. The proposed COLLAGE formations consistently match the FP32 master weights with much less memory footprint and faster training.

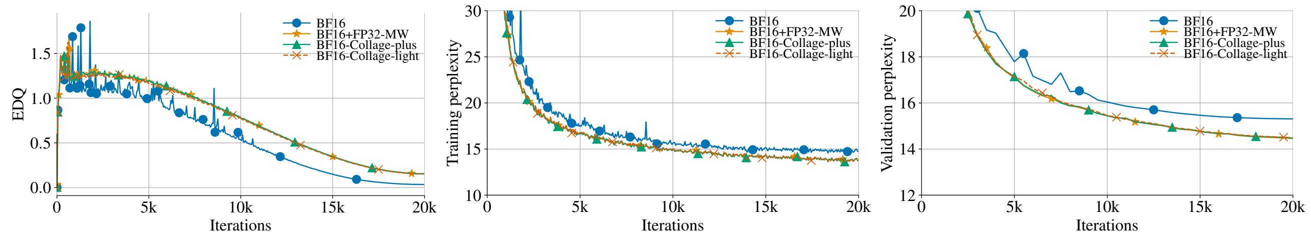

Figure 10. Pretrainnig progress for GPT 125M with settings described in Section 5.2 and global batch-size=2048, $\beta_2 = 0.99$. **Top-left:** EDQ metric vs iterations, **top-right:** training perplexity vs iterations, and **bottom:** validation perplexity vs iterations for different precision strategy listed in Table 2. The proposed COLLAGE formations consistently match the FP32 master weights with much less memory footprint and faster training.

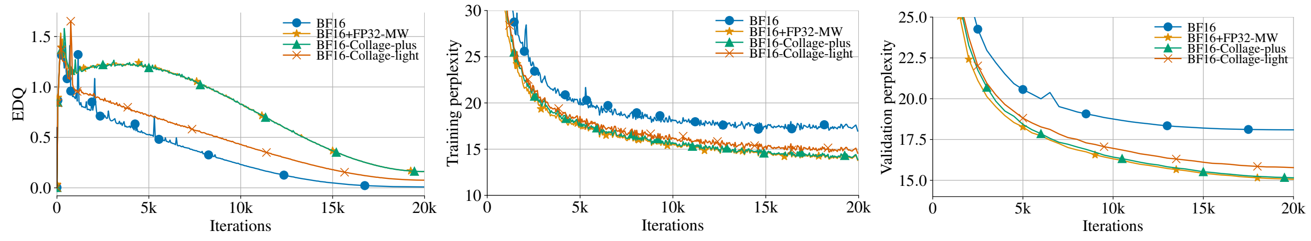

Figure 11. Pretrainnig progress for GPT 125M with settings described in Section 5.2 and global batch-size=1024, $\beta_2 = 0.999$. **Top-left:** EDQ metric vs iterations, **top-right:** training perplexity vs iterations, and **bottom:** validation perplexity vs iterations for different precision strategy listed in Table 2. The proposed COLLAGE formations consistently match the FP32 master weights with much less memory footprint and faster training.

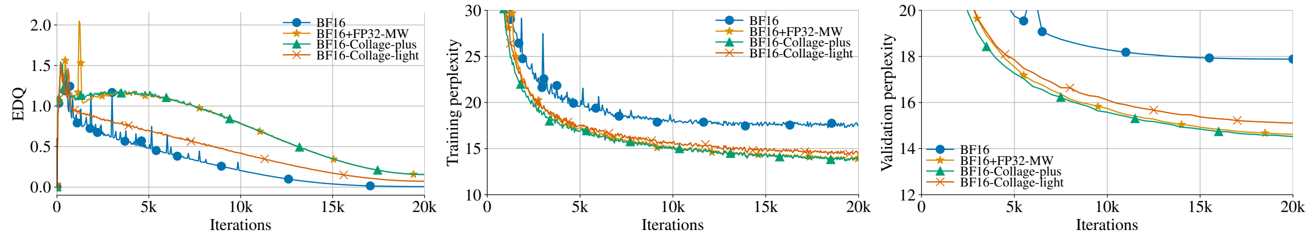

Figure 12. Pretrainnig progress for GPT 125M with settings described in Section 5.2 and global batch-size=2048, $\beta_2 = 0.999$. **Top-left:** EDQ metric vs iterations, **top-right:** training perplexity vs iterations, and **bottom:** validation perplexity vs iterations for different precision strategy listed in Table 2. The proposed COLLAGE formations consistently match the FP32 master weights with much less memory footprint and faster training.

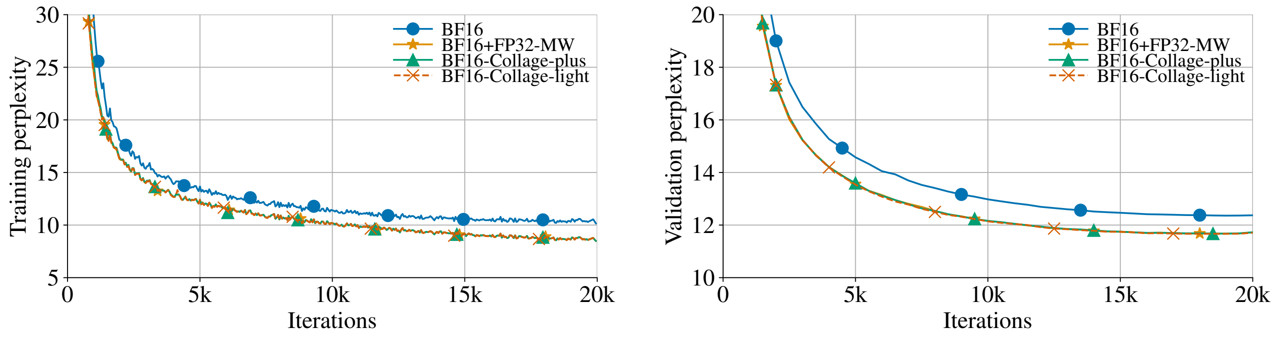

Figure 13. Pretrainnig progress for GPT 1.3B with settings described in Section 5.2 and global batch-size=512, $\beta_2 = 0.95$. **Left:** training perplexity vs iterations, and **right:** validation perplexity vs iterations for different precision strategy listed in Table 2. The proposed COLLAGE formations consistently match the FP32 master weights with much less memory footprint and faster training.

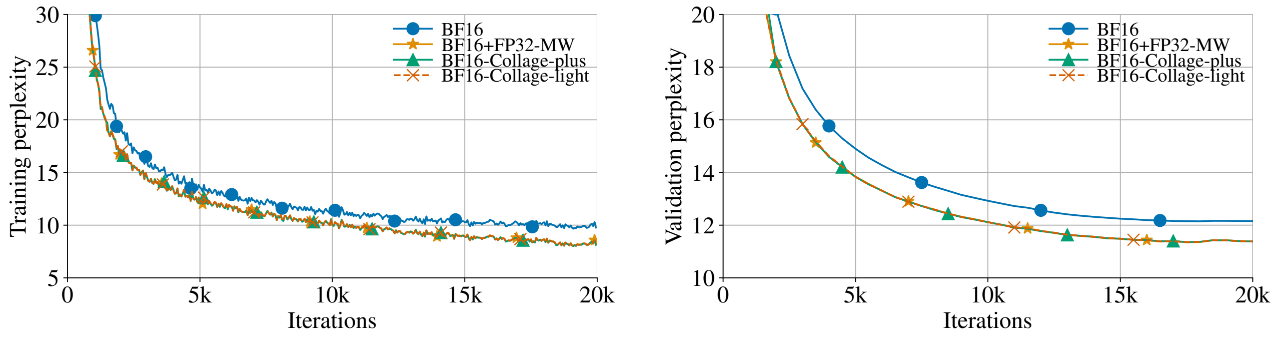

Figure 14. Pretrainnig progress for GPT 2.7B with settings described in Section 5.2 and global batch-size=512, $\beta_2 = 0.95$. **Left:** training perplexity vs iterations, and **right:** validation perplexity vs iterations for different precision strategy listed in Table 2. The proposed COLLAGE formations consistently match the FP32 master weights with much less memory footprint and faster training.

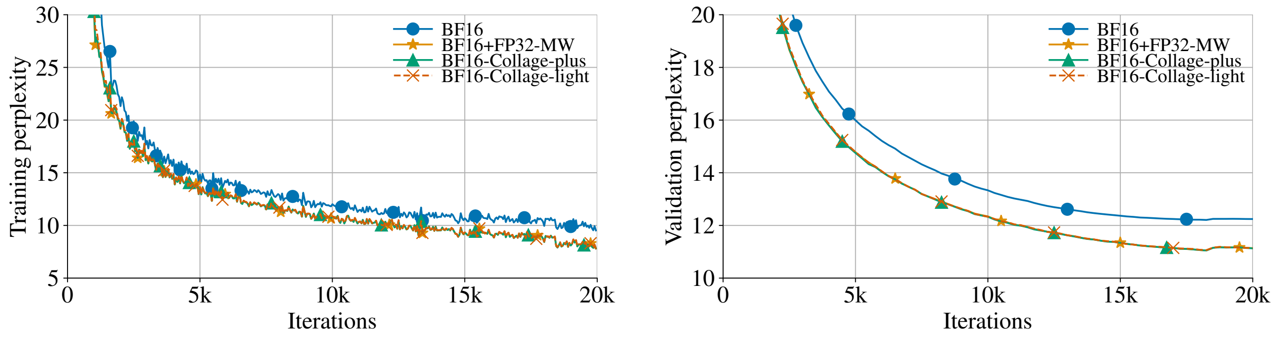

Figure 15. Pretrainnig progress for GPT 6.7B with settings described in Section 5.2 and global batch-size=256, $\beta_2 = 0.95$. **Left:** training perplexity vs iterations, and **right:** validation perplexity vs iterations for different precision strategy listed in Table 2. The proposed COLLAGE formations consistently match the FP32 master weights with much less memory footprint and faster training.