# Jetfire: Efficient and Accurate Transformer Pretraining with INT8 Data Flow and Per-Block Quantization

**Haocheng Xi** [1 2 *] **Yuxiang Chen** [1] **Kang Zhao** [1] **KAI JUN TEH** [1] **Jianfei Chen** [1] **Jun Zhu** [1]

## Abstract

Pretraining transformers are generally time-consuming. Fully quantized training (FQT) is a promising approach to speed up pretraining. However, most FQT methods adopt a quantize-compute-dequantize procedure, which often leads to suboptimal speedup and significant performance degradation when used in transformers due to the high memory access overheads and low-precision computations. In this work, we propose Jetfire, an efficient and accurate INT8 training method specific to transformers. Our method features an INT8 data flow to optimize memory access and a per-block quantization method to maintain the accuracy of pretrained transformers. Extensive experiments demonstrate that our INT8 FQT method achieves comparable accuracy to the FP16 training baseline and outperforms the existing INT8 training works for transformers. Moreover, for a standard transformer block, our method offers an end-to-end training speedup of 1.42x and a 1.49x memory reduction compared to the FP16 baseline.

## 1. Introduction

Recently, large-scale pre-trained transformer-based models such as GPT-4 (OpenAI, 2023), LLAMA (Touvron et al., 2023), and PaLM (Anil et al., 2023) have attained significant breakthroughs in multiple fields, including natural language processing and computer vision. However, pre-training transformers from scratch are extremely resource-intensive since they require numerous computations and high-bandwidth memory for updating weights and accessing huge amounts of training tokens, respectively.

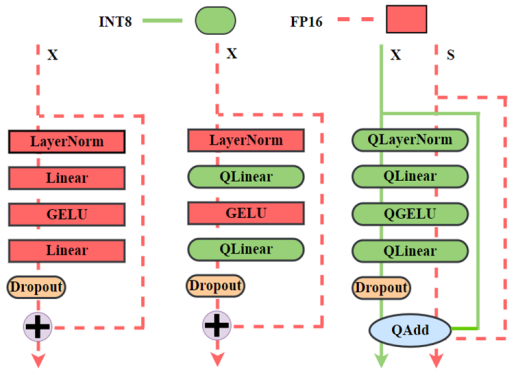



*Figure 1.* Visualization of INT8 data flow. (a) Floating point training with FP data flow. (b) Existing works on quantized training with FP data flow. (c) Ours INT8 training forward process, with INT8 data flow. $\mathbf{X}$ refers to the activation, and $\mathbf{S}$ refers to the corresponding quantization scale factors.

To accelerate the pre-training of transformers, fully quantized training (FQT) has emerged as a promising technique to speed up both the forward and backward passes. FQT integrates quantizers and dequantizers into the original full-precision computational graph. In this way, the expensive floating-point operations during training are replaced with cheaper low-precision alternatives, and activations saved for the backward pass are stored with fewer bits. Thus, both computations and memory bandwidths are largely reduced. Typical FQT works include (Banner et al., 2018; Wang et al., 2018b; Micikevicius et al., 2018; Chen et al., 2020; Wortsman et al., 2023; Xi et al., 2023; Sun et al., 2020; Chmiel et al., 2021; Sun et al., 2019).

However, the existing FQT studies still have three limitations: 1) Existing FQT methods are not accurate enough for Transformer models. Previous FQT methods were mainly designed for CNNs (Zhu et al., 2020; Zhao et al., 2021a), and directly applying these methods to transformer models will result in significant accuracy degradation. Those few papers that focus on transformers often encounter significant quantization errors when computing weight gradients.

[*]This work was done during an internship in the Department of Computer Science and Technology, Tsinghua University. [1]Dept. of Comp. Sci. and Tech., Institute for AI, BNRist Center, THBI Lab, Tsinghua-Bosch Joint ML Center, Tsinghua University [2]Institute for Interdisciplinary Information Sciences, Tsinghua University. Correspondence to: Jianfei Chen <jianfeic@tsinghua.edu.cn>.

*Proceedings of the 41st International Conference on Machine Learning*, Vienna, Austria. PMLR 235, 2024. 

Therefore, they leave this part in floating-point precision, which limits its overall speedup. 2) Most FQT methods only focus on the reduction of computations, regardless of data access overheads (Wortsman et al., 2023). Nevertheless, for a transformer block, only the linear layers are compute-bounded; other layers, such as LayerNorm and activation functions, are normally memory-bounded. Failing to optimize the memory access leads to suboptimal training acceleration. 3) Some FQT techniques require specialized hardware and are not applicable to general computing platforms. For instance, FP8 (Peng et al., 2023; Perez et al., 2023) training is only supported on GPUs with Hopper architecture (nvi, 2022). Not to mention that those hybrid-format quantized training methods rely on application-specific integrated circuits to deliver the desired performance.

To address these limitations, in this work, we propose Jetfire, an INT8 pretraining method for transformers. Specifically, to improve training efficiency, we propose using **INT8 data flow**. As shown in Fig. 1, the INT8 data flow simply refers to the utilization of 8-bit integers for data movement among operators. Compared to the FP16 data flow, the INT8 data flow is 2x faster in theory. In particular, the INT8 data flow considerably enhances the speed of memory-constrained operators, including Layernorm and GELU.

In addition to INT8 flow, we propose **per-block quantization** that is specialized for transformer pretraining. On one hand, compared to per-tensor or per-token quantization (Wortsman et al., 2023), our per-block quantization better preserves the accuracy of pretrained transformers. On the other hand, per-block quantization brings practical training speedup on tensor cores compared to per-channel quantization. Furthermore, our method is applicable to a wide range of computing platforms supporting INT8 matrix multiplications (MMs).

We validate our INT8 FQT method for transformers across a diverse range of tasks, including machine translation, image classification, and generative model pretraining. Jetfire consistently attains comparable accuracy with the FP16 training baseline and has superior accuracy compared with the existing works of INT8 training (Wortsman et al., 2023). On NVIDIA RTX 4090 GPUs, our custom linear and nonlinear operators achieve speedups of 1.4x and 1.8x, respectively, compared to the FP16 baselines. Besides, our Jetfire achieves a speedup of 1.42x for a single transformer block and 1.49x memory reduction compared with the FP16 baseline.

## 2. Related Work

**Post-Training Quantization and Quantization-Aware Training** Post-Training Quantization (PTQ) and Quantization-Aware Training (QAT) aim to find a good low-precision representation for a full-precision model. Post-Training Quantization (PTQ) (Chee et al., 2023; Xiao et al., 2023; Dettmers et al., 2022; Kim et al., 2023; Lin et al., 2023; Kim et al., 2021; Jacob et al., 2018; Liu et al., 2021) converts the pre-trained model's weights to lower-bit representations directly. Quantization-Aware Training (QAT) (Dong et al., 2019b;a; Shen et al., 2019; Zhang et al., 2020; Bai et al., 2020; Tang et al., 2022; Esser et al., 2019) involves retraining the model to adapt its weights and regain accuracy after the quantization process.

*Table 1.* Comparison with related works. SB refers to SwitchBack (Wortsman et al., 2023), TE refers to TransformerEngine (Nvidia, 2022).

| SUPPORT | JETFIRE (OURS) | SB | TE | FP8-LM | DAQ |
|---|---|---|---|---|---|
| TRANSFORMERS | ✓ | ✓ | ✓ | ✓ | × |
| INT8 QUANTIZATION | ✓ | ✓ | × | × | ✓ |
| 8-BIT GRADIENT | ✓ | × | × | ✓ | × |
| 8-BIT DATA FLOW | ✓ | × | × | × | × |

**Fully Quantized Training** Fully Quantized Training (FQT)(Wang et al., 2018b; Banner et al., 2018; Xi et al., 2023; Perez et al., 2023; Wortsman et al., 2023; Zhu et al., 2020; Zhao et al., 2021a; Micikevicius et al., 2018; Nvidia, 2022) has been introduced as a technique to accelerate the training process of neural networks. FQT requires quantizing both the forward propagation and backward propagation to actually accelerate the whole training process. Nowadays 16-bit quantization has been commonly employed with float16 and bfloat16 data formats in training. It introduces loss scaling to prevent underflow and overflow problem.

For INT8 training, the majority of the work focuses on quantization of CNNs (Zhu et al., 2020; Zhao et al., 2021a; Zhou et al., 2021). **SwitchBack** (Wortsman et al., 2023) introduces per-token quantization and successfully applies INT8 training to CLIP models for the first time, but still leaves the calculation of weight gradient in FP. To be more specific, in the forward process of $\mathbf{Y} = \mathbf{X}\mathbf{W}^\top$, they apply per-token quantization for $\mathbf{X}$ and per-channel quantization for $\mathbf{W}^\top$. In the backward process, for $\nabla\mathbf{X} = \nabla\mathbf{Y}\mathbf{W}$, they apply per-token quantization for $\nabla\mathbf{Y}$ and per-channel quantization for $\mathbf{W}$, and leave the calculation of $\nabla\mathbf{W} = \nabla\mathbf{Y}^\top\mathbf{X}$ in full precision. For LLM pre-training, per-token quantization still results in significant accuracy loss due to

With the introduction of the Hopper architecture, FP8 training has also gained attention. TransformerEngine (Nvidia, 2022) incorporates per-layer scaling to reduce quantization errors and proposes using E4M3 during forward and E5M2 during backward passes to adapt. (Perez et al., 2023) explores adjusting per-tensor scaling biases to improve accuracy, while (Peng et al., 2023) investigates further quantizing optimizer states and the weight's master copy to FP8. However, these methods rely on GPUs with the Hopper architecture and cannot be applied to a wider range of GPUs.

As summarized in Table. 1, our method supports INT8 quantization, 8-bit gradient, and 8-bit data flow at the same time, compared to other FQT methods.

## 3. INT8 Data Flow

In this section, we introduce our approach for INT8 training with INT8 data flow. We begin by defining the concept of Fully Quantized Training (FQT).

### 3.1. Fully Quantized Training

Consider a network consisting of linear and nonlinear layers. In the forward pass, these layers can be represented as $\mathbf{Y} = \mathbf{F}(\mathbf{X}, \mathbf{W})$, where $\mathbf{X}$ is the activation, $\mathbf{W}$ is the weight, and $\mathbf{Y}$ is the output, also the next layer's activation. In the backward pass, each layer takes the gradient $\nabla_{\mathbf{Y}}$, $\mathbf{X}$, and $\mathbf{W}$ as inputs and computes the activation gradient and weight gradient by $\nabla_{\mathbf{X}}, \nabla_{\mathbf{W}} = \mathbf{dF}(\nabla_{\mathbf{Y}}, \mathbf{X}, \mathbf{W})$.

Quantization accelerates training by utilizing low-precision computing units on hardware. One notable example is matrix multiplication (MM) in the form of $\mathbf{Y} = \mathbf{X}\mathbf{W}^\top$. When both input matrices are in low-precision format, the MM can have 2x theoretical flops relative to an MM with full-precision inputs, where in this paper we assume that the full-precision format is FP16 and the low-precision format is INT8. Most FQT methods utilize such low-precision MM by a *quantize-compute-dequantize (QCD) approach*: (1) temporarily converting FP16 input matrices to INT8 with a *quantizer* $Q(\cdot)$; (2) perform the INT8 MM to get an INT32 output; and (3) convert the output matrix back to FP16 with a *dequantizer* $Q^{-1}(\cdot)$. With QCD, a MM operator can be formulized as $\mathbf{Y} = \text{QCD-MM}(\mathbf{X}, \mathbf{W}) = Q^{-1}(Q(\mathbf{X})Q(\mathbf{W}^\top))$. As the QCD-MM operator has identical interface to FP16 MMs (i.e., both input and output are still in FP16), we can accelerate training by simply replacing all FP16 MM operators with QCD MMs.

However, QCD only reduces the *computing precision* to INT8, while leaving the *data flow precision* in FP16. That is, MMs are performed under INT8, but the input, output, and data transferred between layers are still in FP16, as illustrated in Fig. 1. The practical speedup of QCD is limited by the *memory bandwidth*. Modern GPUs have excessive computational power, while the GPU memory bandwidth is scarce. An algorithm must have a high arithmetic intensity (i.e., ratio of computing operations to memory accesses) to run fast on GPUs. Unfortunately, the QCD approach's arithmetic intensity is low: the computation is cut by half, but the memory access is not reduced as much, since data are still represented in FP16. More specifically, QCD has three drawbacks:

1. Frequent quantization and dequantization operations incur additional memory access overhead.

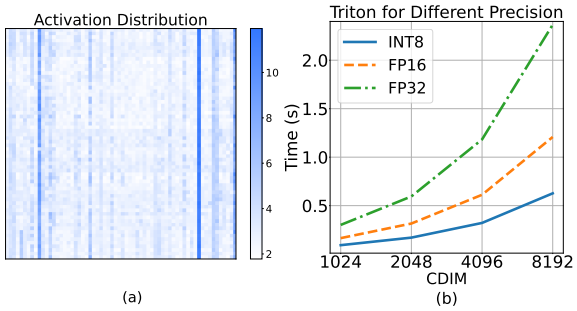



*Figure 2.* (a) Channel-wise outliers in activation distribution. (b) Non-linear operator is memory-bounded.

2. Nonlinear operators cannot be accelerated.
3. GPU memory consumption and communication costs remain high.

### 3.2. FQT with INT8 Data Flow

To address these challenges, we directly utilize INT8 *data flow* throughout the network. That is, we employ the INT8 data format for activations, weights, and gradients, and both our linear and non-linear operators directly take INT8 matrices as inputs and get INT8 tensors as outputs.

To achieve this, we directly represent activation, weight, and gradient tensors in a custom INT8 format defined in Sec. 4. Then, we redesign and implement all operators used in transformer training, including linear operators (Sec. 5) and nonlinear operators (Sec. 6), allowing them to directly use our custom INT8 format as inputs/outputs rather than FP16. The custom INT8 format is carefully designed to ensure that the operators can be implemented efficiently on GPUs, while maintaining accuracy. Such INT8 data flow is compared with QCD in Fig. 1.

With the INT8 data flow, we reduced the amount of memory access in the training algorithm, resulting in better efficiency. In a nutshell, our operators read/write INT8 data from global memory in a block-wise fashion, and perform the quantize/dequantize/compute operations on chip within shared memory and registers. In this way, both computation and memory access can be reduced by half, and the arithmetic intensity remains high. A direct consequence is that, our method can accelerate nonlinear operators, since their memory access is also cut by half. Finally, as the data are stored in INT8 format, the activation memory consumption and amount of communication (tensor / pipeline parallelism) can be also cut by half, effectively avoiding memory capacity and communication bottlenecks.

## 4. Per Block Quantization

In this section, we introduce our INT8 numerical format. Typically, we can approximate an FP16 matrix with an

INT8 matrix $\mathbf{X}^{\text{INT8}}$ and a FP16 scale factor $\mathbf{S}_{\mathbf{X}}^{\text{FP16}}$, that is $\mathbf{X}^{\text{INT8}}, \mathbf{S}_{\mathbf{X}}^{\text{FP16}} = Q(\mathbf{X}^{\text{FP16}})$. Depending on the shape of the scale factor, there are different quantization methods, including per-tensor quantization, per-token quantization, and per-channel quantization. The INT8 numerical format must accurately support the following three MMs of a linear layer in forward and back propagation:

$$\mathbf{Y} = \mathbf{X}\mathbf{W}^\top, \quad \nabla_{\mathbf{X}} = \nabla_{\mathbf{Y}}\mathbf{W}, \quad \nabla_{\mathbf{W}} = \nabla_{\mathbf{Y}}^\top\mathbf{X}.$$

Researchers have observed that activations in transformers are difficult to quantize (Dettmers et al., 2022; Xiao et al., 2023) due to the presence of **channel-wise outliers**. We visualize this problem in Fig. 2a. Per-token quantization assigns different scale factors for different tokens and often results in large quantization errors since outliers appear channel-wise. On the other hand, per-channel quantization assigns different scale factors for different channels and has relatively lower quantization errors, as shown in Sec. 7.3. In addition, gradient outliers also appear along the token axis (Chen et al., 2020; Xi et al., 2023), which poses challenges for computing the weight gradient $\nabla_{\mathbf{W}} = \nabla_{\mathbf{Y}}^\top\mathbf{X}$. In this case, per-token quantization should be applied to the output gradient $\nabla_{\mathbf{Y}}$ to avoid large quantization error.

However, applying per-channel quantization for forward propagation or applying per-token quantization for computing weight gradients both pose challenges in practical hardware implementations. For a MM in the form $\mathbf{C} = \mathbf{AB}$, we call the 0th axis of $\mathbf{A}$ and the 1st axis of $\mathbf{B}$ to be outer axes, as $\mathbf{C}$ has them; the other two axes are inner axes. INT8 MMs are performed with tensor core WMMA (Warp Matrix Multiply-Accumulate) operations (Markidis et al., 2018), and scaling can only be performed at the outer axis of MM if we want to utilize tensor core. As a compromise, (Wortsman et al., 2023) only use per-token quantization for forward propagation, sacrificing accuracy; and fall back to FP16 when computing weight gradients, sacrificing speed.

We propose *per-block quantization* to achieve computational efficiency and preserve accuracy at the same time. For a matrix $\mathbf{X} \in \mathbb{R}^{N\times C}$, we partition $\mathbf{X}$ into blocks $\mathbf{X}_{ij} \in \mathbb{R}^{B\times B}$ along both row axis and column axis, where $B$ is quantization block size, $i, j$ is the index of quantization block along the token and channel axis. We assign a scale factor $\mathbf{s}_{ij}$ for each block $\mathbf{X}_{ij}$ that corresponds to the maximum absolute value in the block. The method can be formulated as:

$$Q(\mathbf{X}_{ij}) = \left\lceil \frac{\mathbf{X}_{ij}}{\mathbf{s}_{ij}} \right\rfloor, \quad Q^{-1}(\mathbf{X}_{ij}^{\text{INT8}}, \mathbf{s}_{ij}) = \mathbf{X}_{ij}^{\text{INT8}}\mathbf{s}_{ij}, \qquad (1)$$

where $\lceil\cdot\rfloor$ is the round operator. We visualize this method in Fig. 3 for better understanding. Since our per-block quantization method partitions along the inner axis, it restricts the impact of an outlier channel/token within a block. Therefore the quantization error is controlled. We will demonstrate in the next section that per-block quantization can be also efficiently implemented on GPUs.

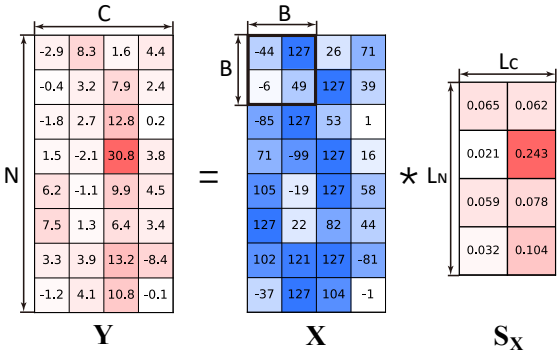



*Figure 3.* Visualization of the per-block quantization methodology. When the original tensor has some outliers, our method can restrict its effect to a $B \times B$ block.

## 5. Linear Layer Operator

In this section, we mainly discuss how our per-block quantization method should be applied to linear layers. We highlight that our linear operator adopts INT8 data flow, that takes INT8 as input and produces INT8 as output.

### 5.1. Notations

We consider the CUDA implementation of the following MM as an example in this section:

$$\mathbf{Y} = \mathbf{X}\mathbf{W}^\top, \text{where } \mathbf{X} \in \mathbb{R}^{N\times C}, \mathbf{W} \in \mathbb{R}^{D\times C}, \mathbf{Y} \in \mathbb{R}^{N\times D}, \quad (2)$$

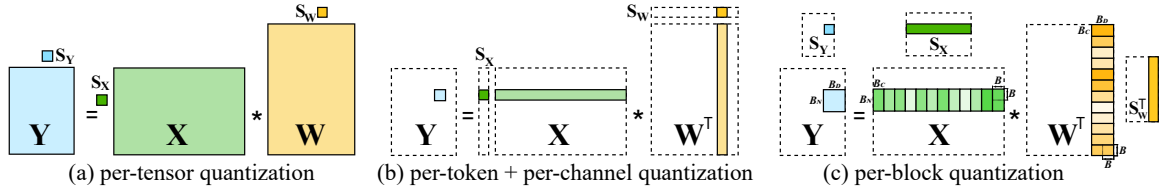
which dimensions are represented as $N \times C \times D$.

In our MM operator, each input and output matrix is represented in per-block INT8 format: a INT8 matrix and a FP16 scale matrix, as defined in Sec. 4. In this case, we have INT8 input denoted as $\mathbf{X}$ and $\mathbf{W}$, and we have scale factors denoted as $\mathbf{S}_{\mathbf{X}} \in \mathbb{R}^{L_N\times L_C}, \mathbf{S}_{\mathbf{W}} \in \mathbb{R}^{L_D\times L_C}$, where $L_N = \frac{N}{B}, L_C = \frac{C}{B}, L_D = \frac{D}{B}$ is the number of quantization blocks along every axis, and $B$ is the quantization block size in Eq. (1). We utilize tensor cores to perform INT8 WMMA. For a single INT8 WMMA instruction, the inputs are two INT8 matrices of shape $16 \times 16$ and the output is an INT32 matrix of shape $16 \times 16$.

### 5.2. 3-Level Tiling of MM

An efficient MM implementation must organize the computation into blocks ("tiling") based on the GPU architecture. We tile the computation in 3 levels. The block dimensions are listed in Table 2.

**CUDA block level** When implementing our MM operator in CUDA, we first parallelize the computation along the $N$ and $D$ axis. Every time we only calculate a submatrix




(a) per-tensor quantization (b) per-token + per-channel quantization (c) per-block quantization

*Figure 4.* Different quantization methods for linear layer.

$\mathbf{B}_{\mathbf{Y}} \in \mathbb{R}^{B_N \times B_D}$ of the output matrix $\mathbf{Y}$. We further divide $C$ into small segments of size $B_C$, and accumulate along this axis. The CUDA block size $B_N \times B_C \times B_D$ is architecture specific. Depending on shared memory capacity and number of threads, typical values are 32, 64, 128, or 256. We define $T_N = \lceil N/B_N \rceil, T_C = \lceil C/B_C \rceil, T_D = \lceil D/B_D \rceil$ to be the number of blocks along each axis of the MM.

For every iteration, we load submatrix $\mathbf{X}_{ik} \in \mathbb{R}^{B_N \times B_C}$ and $\mathbf{W}_{jk} \in \mathbb{R}^{B_D \times B_C}$ from global memory to shared memory and compute the output submatrix $\mathbf{Y}_{ij} \in \mathbb{R}^{B_N \times B_D}$, where $1 \le i \le T_N, 1 \le j \le T_D, 1 \le k \le T_C$ are the CUDA block index along the $N, D, C$ axis.

**Quantization block level** We set $B_N$ and $B_D$ to be multiples of the quantization block size $B$, and set $B_C = B$. In this case, $\mathbf{X}_{ik}$ consists of $R_N = \lceil B_N/B \rceil$ quantization blocks along its 0th-axis, and we use $\mathbf{X}_{ik,p}$ to denote the $p$-th quantization block. Similarly, $\mathbf{W}_{jk}$ consists of $R_D = \lceil B_D/B \rceil$ quantization blocks along its 0th-axis with $\mathbf{W}_{jk,q}$ as the $q$-th block.

We use two nested *for loops* to iterate over $R_N$ and $R_D$, load $\mathbf{X}_{ik,p} \in \mathbb{R}^{B \times B}$ and $\mathbf{W}_{jk,q} \in \mathbb{R}^{B \times B}$ from shared memory to register and performing INT8 WMMA separately to get INT32 output $\mathbf{Y}_{ij,pq} \in \mathbb{R}^{B \times B}$, where $1 \le p \le R_N, 1 \le q \le R_D$ is the quantization block index along $R_N, R_D$ axis.

**WMMA operation level** Within the computation of single quantization blocks, we utilize the INT8 WMMA instruction for computation on register. Therefore, when we set $B = 32$ as an example, we need to perform $2^3 = 8$ WMMA instructions to complete the computation, since a single WMMA instruction can only compute $16 \times 16 \times 16$ MM.

In summary, we divide the implementation of the MM operator into three levels. First, at the CUDA block level, we divide the operator into sizes of $B_N \times B_C \times B_D$ for computation. Then, at the quantization block level, we further divide each CUDA block into sizes of $B \times B \times B$. Finally, at the WMMA operation level, we divide the computation of each quantization block based on the dimensions of the WMMA operation.

*Table 2.* Meaning of Key Constants.

| | |
|---|---|
| $B_N/B_C/B_D$ | CUDA block size in MM |
| $T_N/T_C/T_D$ | Number of CUDA blocks along each axis |
| $B$ | Quantization block size |
| $R_N/R_C/R_D$ | Number of quantization blocks in a CUDA block along each axis |

### 5.3. Quantize and Dequantize

We now discuss how to integrate the quantize and dequantize operators in our algorithm. Since different quantization blocks have different scale factors, after every INT8 WMMA operation, we need to dequantize the INT32 output into FP32 and accumulate in FP32. By applying the same index notation as the previous section, we have

$$\mathbf{Y}^{\text{INT32}}_{ij,pq} = \mathbf{X}_{ik,p}\mathbf{W}^{\top}_{jk,q}, \quad \mathbf{Y}^{\text{FP32}}_{ij,pq} = \sum_{k=1}^{T_C} \mathbf{s}^{\text{FP16}}_{\mathbf{X}_{ik,p}} \mathbf{Y}^{\text{INT32}}_{ij,pq} \mathbf{s}^{\text{FP16}}_{\mathbf{W}_{jk,q}},$$

where $\mathbf{s}_{\mathbf{X}}, \mathbf{s}_{\mathbf{W}}$ is scale factor and both $\mathbf{Y}$s are accumulators.

After the calculation of $\mathbf{Y}^{\text{FP32}}_{ij,pq}$, we quantize it to get a INT8 submatrix $\mathbf{Y}^{\text{INT8}}_{ij,pq}$ and a scale factor $\mathbf{s}_{\mathbf{Y}_{ij,pq}}$.

We formalize our algorithm in Algorithm 1. In the algorithm, we have omitted the details of the quantization block level and WMMA operation level for simplicity. We highlight the overhead introduced by our method in red. We further compare it with per-tensor quantization MM (Banner et al., 2018) and per-token quantization MM (Wortsman et al., 2023) in Fig. 4a.

Our algorithm accurately quantizes channel-wise outliers while introducing only a small amount of overhead for dequantize and quantize operations. We calculate the overhead within the computation of a submatrix $\mathbf{Y}_{ij}$ and compare our method with basic INT8 MM and SwitchBack. Results are reported in Table 3. The time complexity of MM is $O(B_N * C * B_D)$. while our method's overhead time complexity is $O(B_N * T_C * B_D) + (BN + B_D)C$. Since $\frac{C}{T_C} = B_C$ is typically set to 32 or 64 and $B_N, B_D$ is 128 or 256, the overhead is negligible.

## 6. Non-Linear Operator

In this section, we mainly discuss how our per-block quantization method should be applied to non-linear layers. By reducing the precision of the input and output to INT8, we can achieve acceleration for these operators as well.

*Table 3.* Time complexity of different operations in MM.

| Operation | Method | | |
|---|---|---|---|
| | Basic INT8 | SwitchBack | Ours |
| MM | $B_N B_D C$ | $B_N B_D C$ | $B_N B_D C$ |
| 16-bit Load/Store | $(B_N + B_D)C + B_N B_D$ | $\frac{B_N B_C}{T_D} + \frac{B_D B_C}{T_N} + B_N B_D$ | - |
| 8-bit Load/Store | - | $(B_N + B_D)C$ | $(B_N + B_D)C + B_N B_D$ |
| Dequantize | - | $B_N B_D$ | $B_N B_D T_C$ |
| Quantize | - | $\frac{B_N B_C}{T_D} + \frac{B_D B_C}{T_N}$ | $B_N B_D$ |

**Algorithm 1** INT8 Linear Layer

**Require:** INT8 Matrices $\mathbf{X} \in \mathbb{R}^{N \times C}, \mathbf{W} \in \mathbb{R}^{D \times C}$, FP16 scale matrices $\mathbf{S_X} \in \mathbb{R}^{L_N \times L_C}, \mathbf{S_W} \in \mathbb{R}^{L_D \times L_C}$, CUDA Block size $B_N \times B_C \times B_D$

Define $T_N = \left\lceil \frac{N}{B_N} \right\rceil, T_C = \left\lceil \frac{C}{B_C} \right\rceil, T_D = \left\lceil \frac{D}{B_D} \right\rceil$
Define $R_N = \left\lceil \frac{B_N}{B} \right\rceil, R_C = \left\lceil \frac{B_C}{B} \right\rceil R_D = \left\lceil \frac{B_D}{B} \right\rceil$
**for** $1 \leq \mathrm{i} \leq T_N$ **do**
  **for** $1 \leq \mathrm{j} \leq T_D$ **do**
    Initialize accumulator $\mathbf{Y}_{ij}^{\mathbf{FP32}}$, $\mathbf{Y}_{ij}^{\mathbf{INT32}}$
    **for** $1 \leq \mathrm{k} \leq T_C$ **do**
      Load INT8 Block $\mathbf{X}_{ik} \in \mathbb{R}^{B_N \times B_C}$ and scale factor $\mathbf{S}_{\mathbf{X}_{ik}} \in \mathbb{R}^{R_N \times R_C}$
      Load INT8 Block $\mathbf{W}_{jk}^{\top} \in \mathbb{R}^{B_C \times B_D}$ and scale factor $\mathbf{S}_{\mathbf{W}_{jk}^{\top}} \in \mathbb{R}^{R_C \times R_D}$
      On chip, compute INT8 Matmul: $\mathbf{Y}_{ij}^{\mathbf{INT32}} = \mathbf{X}_{ik}\mathbf{W}_{jk}^{\top}$
      On chip, dequantize to FP32 and accumulate: $\mathbf{Y}_{ij}^{\mathbf{FP32}} \leftarrow \mathbf{Y}_{ij}^{\mathbf{FP32}} + \mathbf{S}_{\mathbf{X}_{ik}} \mathbf{Y}_{ij}^{\mathbf{INT32}} \mathbf{S}_{\mathbf{W}_{jk}^{\top}}$
    **end for**
    On Chip, quantize the output $\mathbf{Y}_{ij}^{\mathbf{FP32}}$ to get $\mathbf{Y}_{ij}^{\mathbf{INT8}} \in \mathbb{R}^{B_N \times B_D}$ and scale $\mathbf{S}_{ij} \in \mathbb{R}^{R_N \times R_D}$
    Save $\mathbf{Y}_{ij}^{\mathbf{INT8}}$ and $\mathbf{S}_{\mathbf{Y}_{ij}}$ to global memory.
  **end for**
**end for**

### 6.1. Non-Linear Operators are Memory-Bounded

We have observed that non-linear operators are memory-bounded, which means that the speed of these operators is primarily limited by memory bandwidth, rather than by computation. We validate this by manipulating the data format (INT8, FP16, FP32) for global memory read/write operations in the GELU operator, while internally converting them to FP32 for computation. Fig. 2b illustrates that even computations are kept in FP32, simply reducing the read/write precision can already obtain near-linear speedup. As our method reduces the data flow precision from FP16 to INT8, we anticipate ∼2x speedup for all nonlinear operators. In contrast, QCD cannot accelerate nonlinear operators.

### 6.2. Triton Implementation

Based on the observations above, our main idea is to load-/write in INT8 and leave all calculations within the shared memory through kernel fusion. Specifically, after loading the INT8 input into shared memory, we dequantize it to FP32 and apply the non-linear operators, then quantize the FP32 output back to INT8 format before writing the data into global memory.

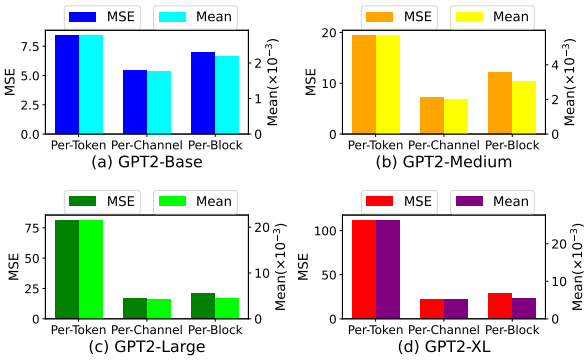



*Figure 5.* Quantization error for different quantization methods. Per-Block refers to our Jetfire quantization method.

We primarily focus on non-linear operators like GELU (Hendrycks & Gimpel, 2016), LayerNorm (Ba et al., 2016), Dropout (Fan et al., 2019), and Add (He et al., 2016), and implement them with Triton (Tillet et al., 2019).

We define $f$ to be the element-wise operator, $\mathbf{X}, \mathbf{Y} \in \mathbb{R}^{N \times C}$ to be the INT8 input and output, $\mathbf{S_X}, \mathbf{S_Y} \in \mathbb{R}^{L_N \times L_C}$ are scale factors, where $L_N = \frac{N}{B}, L_C = \frac{C}{B}$ are number of quantization blocks along each axis and $B$ is the quantization block size. Similar to CUDA, we also do tiling to parallelize the computation. For a single block (whose shape is defined as Triton Block Size) we denote $\mathbf{X}_{ij}^{\mathbf{INT8}}$ to be the input tensor and $\mathbf{S}_{\mathbf{X}_{ij}}^{\mathbf{FP16}}$ to be the scale. The computation process can be represented as

$$\mathbf{Y}_{ij}^{\mathbf{FP32}} = f(Q^{-1}(\mathbf{X}_{ij}^{\mathbf{INT8}}, \mathbf{s}_{\mathbf{X}_{ij}}^{\mathbf{FP16}})); \quad \mathbf{Y}_{ij}^{\mathbf{INT8}}, \mathbf{s}_{\mathbf{Y}_{ij}}^{\mathbf{FP16}} = Q(\mathbf{Y}_{ij}^{\mathbf{FP32}}),$$

where $Q^{-1}$ and $Q$ is the dequantizer and quantizer, $\mathbf{Y}_{ij}^{\mathbf{INT8}}$ is the output tensor, and $\mathbf{s}_{\mathbf{Y}_{ij}}^{\mathbf{FP16}}$ is the scale factor. This algorithm can be expressed as Algorithm 2, where we omit the quantization block level for simplicity.

## 7. Experiments

### 7.1. Settings

We evaluate our INT8 training algorithm Jetfire on a wide variety of tasks including machine translation, image classification, and generative model pretraining. We adopt default architectures, optimizers, and schedulers for all the evaluated models. We adopt the default hyperparameter except for generative model pretraining.

We quantize all of the linear layers in the MLP and attention module and non-linear layers (including GELU, LayerNorm and Dropout) to INT8, and leave multi-head attention in FP16 by employing FlashAttention (Dao et al., 2022). The master copy of the weights is kept in FP32. We quantize linear layers' weights to INT8 prior to each matmul, but leave layernorm's weight and bias to floating-point since

*Table 4.* Results on machine translation, deit pretraining, GPT2 pretraining, and GLUE fine-tuning result based on the pretrained model. FP refers to floating-point, SwitchBack refers to per-token quantization. '–' means the model does not converge.

| Model | #Params(M) | Metric | Baseline | | | Ours |
|---|---|---|---|---|---|---|
| | | | FP | SwitchBack | Per-Tensor | Jetfire |
| Transformer-Base | 61 | BLEU | 26.49 | 26.46 | 26.04 | **26.49** |
| Deit-tiny | 5 | Top1 Acc | 64.08 | 63.55 | – | **63.95** |
| Deit-small | 22 | | 73.43 | 72.80 | – | **73.31** |
| Deit-base | 86 | | 75.67 | 75.62 | – | **76.03** |
| GPT2-base | 124 | Valid loss | 2.9074 | 3.0796 | 3.1638 | **2.8597** |
| GPT2-medium | 350 | | 2.6612 | 2.9141 | 3.1795 | **2.4195** |
| GPT2-large | 774 | | 2.5993 | 3.0512 | 2.9775 | **2.4696** |
| GPT2-base | 124 | GLUE score | $78.50_{0.45}$ | $78.15_{0.15}$ | $76.33_{0.94}$ | $\mathbf{78.18_{0.34}}$ |
| GPT2-medium | 350 | | $82.00_{0.50}$ | $80.03_{0.15}$ | $79.19_{0.22}$ | $\mathbf{81.60_{0.26}}$ |
| GPT2-large | 774 | | $83.01_{0.24}$ | $78.74_{0.24}$ | $75.88_{0.35}$ | $\mathbf{82.94_{0.70}}$ |

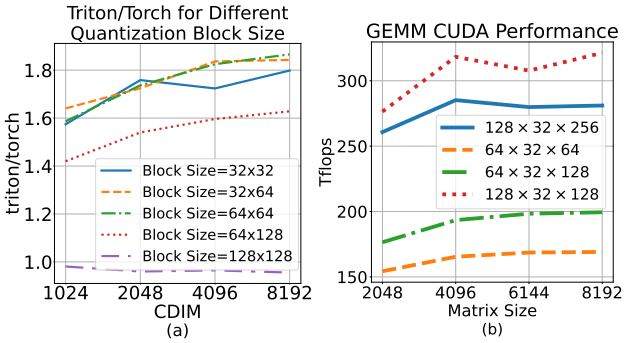



*Figure 6.* Speed test of GELU and GEMM operator. (a) Triton kernel speedup with different Triton block sizes.(b) GEMM CUDA kernel speed with different CUDA block sizes.

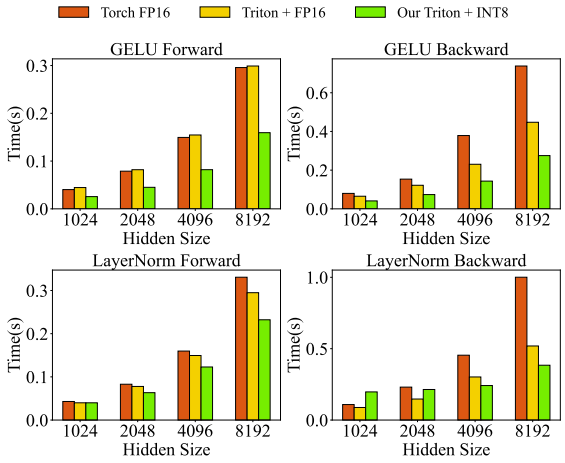



*Figure 7.* Speed comparision between our INT8 non-linear operator and pytorch FP16 implementation.

| Model | Swin-tiny | Swin-small | Swin-base |
|---|---|---|---|
| FP | 77.55 | 80.39 | 80.45 |
| Jetfire | 77.51 | 80.39 | 80.37 |
| | ViT-base | ViT-large | |
| FP | 83.45 | 85.72 | |
| Jetfire | 83.48 | 85.67 | |

*Table 5.* Comparison of FP and Jetfire

they are relatively small. We compare our method with floating point training baseline (denoted as FP), per-tensor quantization, and SwitchBack (Wortsman et al., 2023)). We do not compare with FP8 training algorithms as they require specialized Hopper architecture GPU to run, making them less accessible. We emphasize that only our method adopts an INT8 data flow and quantizes non-linear layers.

We implement our linear operators with CUDA and implement non-linear operators with Triton. CUDA block size is set to $128 \times 32 \times 128$ and Triton block size is set to $64 \times 64$. The quantization block size is set to $B = 32$.

### 7.2. Converged Model Accuracy

**Machine Translation** We validate our Jetfire's effectiveness on the translation task. We train a Transformer-base model on WMT 14 En-De dataset (Bojar et al., 2014) based on Nvidia's recipe [1]. In Table 4 we report the BLEU (Papineni et al., 2002) score result. Our method has no degradation compared with the FP baseline, while the SwitchBack baseline has $0.03\%$ BLEU score degradation, and the per-tensor quantization baseline has $0.4\%$ degradation.

**Image Classification - Deit** We do pretraining for Deit-Tiny, Deit-Small, and Deit-Base (Touvron et al., 2021) model on ImageNet1K (Deng et al., 2009) for 90 epochs based on facebook research's recipe [2]. Results are listed on Table 4. In all experiments, Our method has less than $0.1\%$ accuracy degradation compared with the floating-point base-

[1] https://github.com/NVIDIA/DeepLearningExamples/tree/master/PyTorch/Translation/Transformer

[2] https://github.com/facebookresearch/deit

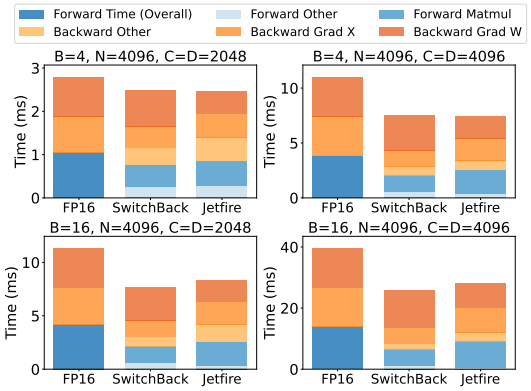



*Figure 8.* Matrix Multiplication Speed test for Different Methods in different settings ($B$=Batch Size, $N$=Sequence Length).

line, and for Deit-base, our method shows $0.4\%$ improvement. For Deit-tiny and Deit-small models, Switchback has over $0.5\%$ accuracy degradation, and per-tensor quantization does not converge. This indicates that our method can accurately quantize channel-wise outliers. Comparison with more baselines (Wang & Kang, 2023; Zhao et al., 2021b) can be found in Appendix C

**Image Classification - Swin Transformers and ViT** We do pretraining for Swin-Transformers(Swin-tiny, Swin-small, Swin-base) for 90 epochs and fine-tuned ViT(ViT-base, ViT-large) for 100 epochs without pre-training (MAE includes pretraining and finetuning) on ImageNet1K. We adopt the official training recipe [3][4] and default hyperparameters, and only compare with the full precision training baseline. The results are shown in Figure 5. In all of the experiments, our method achieves less than 0.1% accuracy degradation, which proves the accuracy of our method.

**Generative Model Pretraining** We evaluate our method by training three GPT2 (Radford et al., 2019) models with different sizes: GPT2-base for 300k steps, GPT2-medium for 200k steps, and GPT2-large for 120k steps on the OpenWebText (Peterson et al., 2019) dataset based on NanoGPT[5] (Hyperparameters: Learning Rate = $1.5 \times 10^{-4}$, Weight Decay = $10^{-1}$). We report the validation loss and the fine-tuning average accuracy on the GLUE (Wang et al., 2018a) dataset over 3 seeds. The results are shown in Table 4.

We found that SwitchBack resulted in 0.1 valid loss degradation on GPT-base and led to 0.3-0.4 valid degradation on GPT-medium and GPT-large. Our method achieves even lower valid loss compared to the FP baseline, which may be attributed to the regularization effect of quantization.

For fine-tuning, our method shows less than 0.3% degradation compared to baseline, while SwitchBack has a degradation of 0.3% on GPT2-base, 1.8% on GPT2-medium, and 4.3% on GPT2-large. This indicates that for LLM pretraining, the influence of channel-wise outliers is significant, and our quantization method effectively preserves accuracy.

### 7.3. Ablation Study

**Quantization Error** We study the quantization error of different quantization methods on four different sizes of GPT2 models to show our method's effectiveness. We focus on the activation of the final layer and calculate the mean squared error (MSE) and the mean error after quantization. The results are shown in Fig. 5. For all models, per-channel quantization consistently resulted in smaller quantization errors compared to per-token quantization. Jetfire (ours) achieves lower quantization error than per-token quantization while performing on par with per-channel quantization.

**CUDA kernel and Triton kernel block size** We have found that the selection of the block size for Triton and CUDA kernels is crucial. A large block size leads to a decrease in parallelism, while a small block size results in low utilization of bandwidth and computational resources. Both cases can result in low kernel speed. In Fig. 6a 6b, we test the kernel's speed under different block sizes and find that optimal efficiency is achieved when we set Triton block size $= 64 \times 64$ and CUDA block size $= 128 \times 32 \times 128$.

### 7.4. Operator and End-to-End experiments

**Linear layer speedup** We test the speedup of our custom linear layer on RTX 4090. We analyzed the time consumption of each component in forward and backward passes and compared the speed of our implementation with FP16 and SwitchBack linear layers. The results are shown in Fig. 8. Our MM **operator** provides about 60% speed improvement compared to FP16. Other overhead components like quantizing and reshaping have a relatively minor impact. Our method achieves 40% overall speedup (forward + backward), which is comparable to the acceleration result of SwitchBack, where SwitchBack leaves the calculation of weight gradient in FP. The speedup becomes larger when the matrix size increases since the overhead proportion decreases, which is demonstrated in Table 10. Acceleration results on RTX 3090 can be found in Appendix D.2.

**Non-linear operator speedup** We also test the speedups offered by our custom non-linear layers, which is the **first** quantized training work to achieve acceleration for these non-linear operators.

Our INT8 GELU operator achieves 80% speedup in both

[3]https://github.com/microsoft/Swin-Transformer

[4]https://github.com/facebookresearch/mae?tab=readme-ov-file

[5]https://github.com/karpathy/nanoGPT.

*Table 6.* Acceleration ratios for End-to-end comparison (SB refers to SwitchBack basic version) on GPT2 model.

| | FORWARD | | BACKWARD | | OVERALL | |
|---|---|---|---|---|---|---|
| HIDDEN SIZE | SB | OURS | SB | OURS | SB | OURS |
| 4096 | 1.50 | 1.32 | 1.18 | 1.46 | 1.27 | **1.42** |
| 2048 | 1.53 | 1.29 | 1.24 | 1.41 | 1.32 | **1.37** |
| 1024 | 0.94 | 0.97 | 1.14 | 1.11 | **1.07** | **1.07** |

*Table 7.* Activation memory reduction ratios for End-to-end comparison (SB refers to SwitchBack Memeory Efficient version) on GPT2 model.

| | BS=1 | | BS=2 | | BS=4 | |
|---|---|---|---|---|---|---|
| LAYER NUM | SB | OURS | SB | OURS | SB | OURS |
| 12 | 1.19 | **1.33** | 1.14 | **1.31** | 1.11 | **1.29** |
| 24 | 1.24 | **1.49** | 1.18 | **1.47** | 1.14 | **1.45** |

forward and backward passes compared to PyTorch's FP16 operators. Our INT8 LayerNorm operator achieves 40% speed up in its forward pass and up to 90% speedup in its backward pass when hidden size $= 8192$ but does not accelerate when the hidden size is small. These results indicate that the global memory access is indeed the bottleneck for these non-linear operators, and our INT8 data flow can effectively solve the bottleneck, resulting in near-ideal speedup.

**End-to-end speedup** We experimented with GPT2 models and varied the network hidden size to show the end-to-end speedup for our Jetfire method over PyTorch's FP16 training on RTX 4090. We integrated all linear and non-linear operators and reported the speedup of a transformer layer. We compared the forward, backward, and overall runtime speedup with the SwitchBack layer. Results in Table 6 showed that our method achieved comparable or improved acceleration compared to SwitchBack. This is primarily because our linear operators in backpropagation are faster than SwitchBack, and we can accelerate all of the non-linear operators in both forward and backward propagation. Acceleration results on RTX 3090 can be found in Appendix 11.

**End-to-End Memory Reduction** We experimented with GPT2 models and varied the network depth and batch size to show the memory reduction of our method. We report the reduction ratio of activation memory. The results are shown in Table 7. Our method achieved up to 1.49x activation memory reduction, which is better than SwitchBack since we reduced the memory footprint of non-linear operators.

## 8. Conclusion

In this work, we propose Jetfire, an INT8 pretraining method for transformer models. For the first time, we propose to use INT8 data flow in pretraining to reduce computation, memory access, memory usage, and communication at the same time. We also propose to use per-block quantization for all of the activations, weights, and gradients for both linear and non-linear layers to preserve accuracy. Extensive experiments demonstrate that our proposed method performs on par with FP baselines, and can effectively accelerate the training speed and reduce the memory footprint.

## Acknowledgements

The authors would like to thank Bingrui Li, Ziteng Wang, Jiayi Zhong, Cheng Lu for the helpful discussions. This work was supported by the National Science and Technology Major Project (2021ZD0110502), NSFC Projects (Nos. 62376131, 62061136001, 62106123, 62076147, U19A2081, 61972224), Tsinghua Institute for Guo Qiang, and the High Performance Computing Center, Tsinghua University. J.Z is also supported by the XPlorer Prize.

## Impact Statement

Our INT8 fully quantized training (FQT) method significantly improves the efficiency of deep learning by reducing computations and memory usage of training transformers. This contributes substantially to energy conservation and emission reduction, and aligns with the objective of global sustainability. Besides, our method promotes the democratization of artificial intelligence (AI) by making transformer training more accessible to cheap and low-resource platforms. Nevertheless, this method could also be misused to expedite the training of "evil models" designed to generate harmful content.

## References


Nvidia h100 tensor core gpu architecture. https://resources.nvidia.com/en-us-tensor-core, 2022.

Anil, R., Dai, A. M., Firat, O., Johnson, M., Lepikhin, D., Passos, A., Shakeri, S., Taropa, E., Bailey, P., Chen, Z., et al. Palm 2 technical report. *arXiv preprint arXiv:2305.10403*, 2023.

Ba, J. L., Kiros, J. R., and Hinton, G. E. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

Bai, H., Zhang, W., Hou, L., Shang, L., Jin, J., Jiang, X., Liu, Q., Lyu, M., and King, I. Binarybert: Pushing the limit of bert quantization. *arXiv preprint arXiv:2012.15701*, 2020.

Banner, R., Hubara, I., Hoffer, E., and Soudry, D. Scalable methods for 8-bit training of neural networks. In *Advances in Neural Information Processing Systems*, pp. 5145–5153, 2018.

Bojar, O., Buck, C., Federmann, C., Haddow, B., Koehn, P., Leveling, J., Monz, C., Pecina, P., Post, M., Saint-Amand, H., et al. Findings of the 2014 workshop on statistical machine translation. In *Proceedings of the ninth workshop on statistical machine translation*, pp. 12–58, 2014.

Chee, J., Cai, Y., Kuleshov, V., and De Sa, C. Quip: 2-bit quantization of large language models with guarantees. *arXiv preprint arXiv:2307.13304*, 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.

Chmiel, B., Banner, R., Hoffer, E., Yaacov, H. B., and Soudry, D. Logarithmic unbiased quantization: Practical 4-bit training in deep learning. *arXiv preprint arXiv:2112.10769*, 2021.

Dao, T., Fu, D., Ermon, S., Rudra, A., and Ré, C. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems*, 35:16344–16359, 2022.

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pp. 248–255. Ieee, 2009.

Dettmers, T., Lewis, M., Belkada, Y., and Zettlemoyer, L. Llm. int8 (): 8-bit matrix multiplication for transformers at scale. *arXiv preprint arXiv:2208.07339*, 2022.

Dong, Z., Yao, Z., Cai, Y., Arfeen, D., Gholami, A., Mahoney, M. W., and Keutzer, K. Hawq-v2: Hessian aware trace-weighted quantization of neural networks. *arXiv preprint arXiv:1911.03852*, 2019a.

Dong, Z., Yao, Z., Gholami, A., Mahoney, M., and Keutzer, K. Hawq: Hessian aware quantization of neural networks with mixed-precision. *ICCV*, 2019b.

Esser, S. K., McKinstry, J. L., Bablani, D., Appuswamy, R., and Modha, D. S. Learned step size quantization. *arXiv preprint arXiv:1902.08153*, 2019.

Fan, A., Grave, E., and Joulin, A. Reducing transformer depth on demand with structured dropout. *arXiv preprint arXiv:1909.11556*, 2019.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

Hendrycks, D. and Gimpel, K. Gaussian error linear units (gelus). *arXiv preprint arXiv:1606.08415*, 2016.

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.

Kim, S., Gholami, A., Yao, Z., Mahoney, M. W., and Keutzer, K. I-bert: Integer-only bert quantization. In *International conference on machine learning*, pp. 5506–5518. PMLR, 2021.

Kim, S., Hooper, C., Gholami, A., Dong, Z., Li, X., Shen, S., Mahoney, M. W., and Keutzer, K. Squeezellm: Dense-and-sparse quantization. *arXiv preprint arXiv:2306.07629*, 2023.

Lin, J., Tang, J., Tang, H., Yang, S., Dang, X., and Han, S. Awq: Activation-aware weight quantization for llm compression and acceleration. *arXiv preprint arXiv:2306.00978*, 2023.

Liu, Z., Wang, Y., Han, K., Zhang, W., Ma, S., and Gao, W. Post-training quantization for vision transformer. *Advances in Neural Information Processing Systems*, 34: 28092–28103, 2021.

Markidis, S., Der Chien, S. W., Laure, E., Peng, I. B., and Vetter, J. S. Nvidia tensor core programmability, performance & precision. In *2018 IEEE international parallel and distributed processing symposium workshops (IPDPSW)*, pp. 522–531. IEEE, 2018.

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. In *International Conference on Learning Representations*, 2018.

Nvidia. Nvidia transformer engine. https://docs.nvidia.com/deeplearning/transformer-engine/index.html, 2022.

OpenAI. Gpt-4 technical report, 2023.

Papineni, K., Roukos, S., Ward, T., and Zhu, W.-J. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pp. 311–318, 2002.

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. *arXiv preprint arXiv:2310.18313*, 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. *arXiv preprint arXiv:2309.17224*, 2023.

Peterson, J., Meylan, S., and Bourgin, D. Open clone of openai's unreleased webtext dataset scraper, 2019.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., Sutskever, I., et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

Shen, S., Dong, Z., Ye, J., Ma, L., Yao, Z., Gholami, A., Mahoney, M. W., and Keutzer, K. Q-bert: Hessian based ultra low precision quantization of bert. *arXiv preprint arXiv:1909.05840*, 2019.

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*, pp. 4901–4910, 2019.

Sun, X., Wang, N., Chen, C.-Y., Ni, J., Agrawal, A., Cui, X., Venkataramani, S., El Maghraoui, K., Srinivasan, V. V., and Gopalakrishnan, K. Ultra-low precision 4-bit training of deep neural networks. In *Advances in Neural Information Processing Systems*, volume 33, 2020.

Tang, H., Zhang, X., Liu, K., Zhu, J., and Kang, Z. Mkq-bert: Quantized bert with 4-bits weights and activations. *arXiv preprint arXiv:2203.13483*, 2022.

Tillet, P., Kung, H.-T., and Cox, D. 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*, pp. 10–19, 2019.

Touvron, H., Cord, M., Douze, M., Massa, F., Sablayrolles, A., and Jégou, H. Training data-efficient image transformers & distillation through attention. In *International conference on machine learning*, pp. 10347–10357. PMLR, 2021.

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. *arXiv preprint arXiv:2302.13971*, 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. *arXiv preprint arXiv:1804.07461*, 2018a.

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*, pp. 7675–7684, 2018b.

Wang, S. and Kang, Y. Gradient distribution-aware int8 training for neural networks. *Neurocomputing*, 541: 126269, 2023.

Wortsman, M., Dettmers, T., Zettlemoyer, L., Morcos, A., Farhadi, A., and Schmidt, L. Stable and low-precision training for large-scale vision-language models. *arXiv preprint arXiv:2304.13013*, 2023.

Xi, H., Li, C., Chen, J., and Zhu, J. Training transformers with 4-bit integers. *arXiv preprint arXiv:2306.11987*, 2023.

Xiao, G., Lin, J., Seznec, M., Wu, H., Demouth, J., and Han, S. Smoothquant: Accurate and efficient post-training quantization for large language models. In *International Conference on Machine Learning*, pp. 38087–38099. PMLR, 2023.

Zhang, W., Hou, L., Yin, Y., Shang, L., Chen, X., Jiang, X., and Liu, Q. Ternarybert: Distillation-aware ultra-low bit bert. *arXiv preprint arXiv:2009.12812*, 2020.

Zhao, K., Huang, S., Pan, P., Li, Y., Zhang, Y., Gu, Z., and Xu, Y. Distribution adaptive int8 quantization for training cnns. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pp. 3483–3491, 2021a.

Zhao, K., Huang, S., Pan, P., Li, Y., Zhang, Y., Gu, Z., and Xu, Y. Distribution adaptive int8 quantization for training cnns. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pp. 3483–3491, 2021b.

Zhou, Q., Guo, S., Qu, Z., Guo, J., Xu, Z., Zhang, J., Guo, T., Luo, B., and Zhou, J. Octo: INT8 training with loss-aware compensation and backward quantization for tiny on-device learning. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pp. 177–191, 2021.

Zhu, F., Gong, R., Yu, F., Liu, X., Wang, Y., Li, Z., Yang, X., and Yan, J. Towards unified int8 training for convolutional neural network. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 1969–1979, 2020.

## A. Triton Implementation of Non-Linear Operators

For the GELU function, its forward and backward operator is:

$$\mathrm{GELU}(x) = x \cdot \Phi(x), \quad \frac{\mathrm{dGELU}(x)}{\mathrm{d}x} = \frac{x}{\sqrt{2\pi}} e^{-\frac{x^2}{2}} + \Phi(x).$$

For Dropout, its forward and backward operator is:

$$\mathrm{Drop}(x) = \frac{1}{1-p} x \circ m, \quad \frac{\mathrm{dDrop}(x)}{\mathrm{d}x} = \frac{1}{1-p} m.$$

For Add, when we calculate the residual connection $y = x + f(x)$, we also need to perform $\mathrm{d}x = \mathrm{d}f(y) + \mathrm{d}y$ in the backward process. This addition operator can be represented as:

$$\mathrm{Add}(x_1, x_2) = x_1 + x_2.$$

**Algorithm 2** INT8 Non-Linear Operator

**Require:** INT8 Matrix $\mathbf{X} \in \mathbb{R}^{N \times C}$, FP16 scale matrix $\mathbf{S_X} \in \mathbb{R}^{L_N \times L_C}$, element-wise function $f$

Define $T_N = \left\lceil \frac{N}{B_N} \right\rceil, T_C = \left\lceil \frac{C}{B_C} \right\rceil$
Define $R_N = \left\lceil \frac{B_N}{B} \right\rceil, R_C = \left\lceil \frac{B_C}{B} \right\rceil$
**for** $1 \leq \mathrm{i} \leq T_N$ **do**
  **for** $1 \leq \mathrm{j} \leq T_C$ **do**
    Load INT8 block $\mathbf{X}_{ij} \in \mathbb{R}^{B_N \times B_C}$, $\mathbf{S}_{\mathbf{X}_{ij}} \in \mathbb{R}^{R_N \times R_C}$
    Dequantize $\mathbf{X}_{ij}$ and $\mathbf{S}_{\mathbf{X}_{ij}}$ to get $\mathbf{X}_{ij}^{\mathbf{FP32}}$
    Operate: $\mathbf{Y}_{ij}^{\mathbf{FP32}} = f(\mathbf{X}_{ij}^{\mathbf{FP32}})$
    Quantize $\mathbf{Y}_{ij}^{\mathbf{FP32}}$ to get $\mathbf{Y}_{ij}^{\mathbf{INT32}} \in \mathbb{R}^{B_N \times B_C}$ and scale factor $\mathbf{S}_{\mathbf{Y}_{ij}} \in \mathbb{R}^{R_N \times R_C}$
    Save $\mathbf{Y}_{ij}^{\mathbf{INT32}}$ and $\mathbf{S}_{\mathbf{Y}_{ij}}$ to global memory.
  **end for**
**end for**

Differing from non-linear operators above, LayerNorm involves interactions between elements. Therefore, performing calculations separately for each $B_N \times B_C$ block is not feasible. In order to solve the problem, we observed that both pre-norm and post-norm models encountered the ADD operator before LayerNorm.

We make the following modifications to our ADD operator: We will calculate the mean and sum of squares for each row of the block $(B_N, B_C)$ and store these values. We will then get the mean matrix and sum of squares matrix of size $N \times \frac{C}{B_C}$, which reduces the amount of data we need to load and store by $\frac{1}{B_C}$. Before the LayerNorm operator, we use these values to compute the mean and variance for each row, which size is $N \times 1$. This allows the LayerNorm to directly access these values. The implementation of the remaining part of LayerNorm is similar to that of GELU.

## B. Detailed Results of GLUE Fine-Tuning Test

*Table 8.* Detailed Results of GLUE fine-tuning test based on the pretrained model. FP refers to floating-point, SwitchBack refers to per-token quantization. '–' means the model does not converge.

| | | | BASELINE | | | OURS |
|---|---|---|---|---|---|---|
| MODEL | #PARAMS(M) | METRIC | FP | SWITCHBACK | PER-TENSOR | JETFIRE |
| GPT2-BASE | 124 | COLA | $43.97_{3.32}$ | $\mathbf{44.68_{1.06}}$ | $41.43_{2.23}$ | $41.98_{2.51}$ |
| | | STSB | $83.20_{0.95}$ | $81.08_{1.39}$ | $75.63_{5.35}$ | $\mathbf{81.66_{0.45}}$ |
| | | RTE | $64.02_{1.37}$ | $64.74_{0.83}$ | $64.14_{0.75}$ | $\mathbf{65.22_{0.91}}$ |
| | | MRPC | $85.48_{0.78}$ | $84.30_{0.49}$ | $83.68_{0.61}$ | $\mathbf{85.01_{0.49}}$ |
| | | SST2 | $91.40_{0.30}$ | $\mathbf{92.09_{0.50}}$ | $90.79_{0.26}$ | $91.93_{0.35}$ |
| | | QNLI | $87.97_{0.35}$ | $\mathbf{87.58_{0.19}}$ | $86.64_{0.25}$ | $87.57_{0.19}$ |
| | | QQP | $90.07_{0.001}$ | $89.94_{0.07}$ | $88.84_{0.09}$ | $\mathbf{90.14_{0.03}}$ |
| | | MNLI | $81.84_{0.22}$ | $80.76_{0.24}$ | $79.51_{0.36}$ | $\mathbf{81.92_{0.18}}$ |
| GPT2-MEDIUM | 350 | COLA | $53.55_{2.89}$ | $50.97_{0.58}$ | $48.82_{1.43}$ | $\mathbf{53.38_{1.08}}$ |
| | | STSB | $86.17_{0.78}$ | $83.21_{1.61}$ | $81.66_{0.50}$ | $\mathbf{84.98_{0.32}}$ |
| | | RTE | $68.23_{1.44}$ | $63.78_{0.91}$ | $67.27_{0.21}$ | $\mathbf{68.47_{0.21}}$ |
| | | MRPC | $88.16_{0.91}$ | $84.79_{0.68}$ | $84.98_{0.15}$ | $\mathbf{86.74_{1.36}}$ |
| | | SST2 | $93.73_{0.29}$ | $\mathbf{94.00_{0.18}}$ | $91.78_{0.13}$ | $93.46_{0.23}$ |
| | | QNLI | $90.49_{0.14}$ | $89.61_{0.20}$ | $88.05_{0.16}$ | $\mathbf{90.15_{0.17}}$ |
| | | QQP | $90.98_{0.15}$ | $90.54_{0.15}$ | $90.02_{0.08}$ | $\mathbf{90.93_{0.08}}$ |
| | | MNLI | $84.69_{0.37}$ | $83.37_{0.36}$ | $80.93_{0.31}$ | $\mathbf{84.65_{0.11}}$ |
| GPT2-LARGE | 774 | COLA | $52.22_{0.15}$ | $48.70_{1.20}$ | $35.85_{1.23}$ | $\mathbf{54.06_{1.62}}$ |
| | | STSB | $84.99_{1.37}$ | $79.49_{0.15}$ | $80.00_{0.15}$ | $\mathbf{84.99_{1.37}}$ |
| | | RTE | $76.65_{1.67}$ | $65.10_{1.10}$ | $64.26_{2.60}$ | $\mathbf{74.73_{2.53}}$ |
| | | MRPC | $86.70_{0.92}$ | $83.75_{1.00}$ | $83.09_{0.45}$ | $\mathbf{87.06_{1.21}}$ |
| | | SST2 | $94.84_{0.50}$ | $92.47_{0.07}$ | $90.86_{0.24}$ | $\mathbf{94.65_{0.18}}$ |
| | | QNLI | $91.35_{0.23}$ | $88.42_{0.12}$ | $85.25_{0.36}$ | $\mathbf{91.19_{0.52}}$ |
| | | QQP | $91.40_{0.08}$ | $89.92_{0.19}$ | $89.21_{0.03}$ | $\mathbf{91.19_{0.02}}$ |
| | | MNLI | $85.88_{0.14}$ | $82.05_{0.20}$ | $78.53_{0.15}$ | $\mathbf{85.61_{0.30}}$ |

## C. Comparisons with methods targeting CNNs

In this section, We tested two INT8 training for CNN models (Wang & Kang, 2023; Zhao et al., 2021b) on the DeiT pretraining experiment. As reported in Table 9, both of them showed significant accuracy degradation. This indicates that these methods are not sufficient to be applied to transformer models.

| Model | FP | SwitchBack | Per-tensor | Jetfire | GDA | DAQ |
|---|---|---|---|---|---|---|
| Deit-tiny | 64.08 | 63.55 | - | 63.95 | 62.14 | 61.80 |
| Deit-small | 73.43 | 72.80 | - | 73.31 | 70.98 | 70.66 |
| Deit-base | 75.67 | 75.62 | - | 76.03 | 73.06 | 72.40 |

*Table 9.* Comparison of different methods on various Deit models

## D. Acceleration Experiments

### D.1. Overhead portion in Linear Layer

We tested the percentage of time taken by all quantization, dequantization, transpose, and other overhead processes during the forward and backward passes in a linear layer. We find that in Table 10, the relative overhead from quantization/dequantization diminishes with increasing model size, leading to more significant speed improvements.

| | | Sequence Length | | | | |
|---|---|---|---|---|---|---|
| Batch Size | Place | 1024 | 2048 | 4096 | 6144 | 8192 |
| 4 | forward | 44.5 | 22.1 | 10.2 | 6.8 | 5.5 |
| 4 | backward | 30.4 | 29.0 | 16.6 | 12.2 | 9.2 |
| 4 | overall | 34.5 | 26.7 | 14.5 | 10.3 | 7.9 |
| 16 | forward | 23.1 | 8.1 | 2.7 | 1.7 | 1.3 |
| 16 | backward | 24.6 | 26.4 | 8.7 | 7.3 | 6.2 |
| 16 | overall | 23.7 | 18.3 | 11.5 | 9.9 | 8.9 |

*Table 10.* Percentage of overhead in a linear layer.

| Size Settings | Fwd-SB | Fwd-Ours | Bwd-SB | Bwd-Ours | All-SB | All-Ours |
|---|---|---|---|---|---|---|
| Linear Layer, C=D=2048 | 1.60 | 1.53 | 1.38 | 1.31 | 1.45 | 1.38 |
| Linear Layer, C=D=4096 | 2.43 | 1.87 | 1.37 | 1.48 | 1.62 | 1.60 |
| Linear Layer, C=D=8192 | 2.56 | 1.70 | 1.24 | 1.40 | 1.51 | 1.49 |
| End-to-End, hidden=1024 | 1.08 | 0.94 | 1.08 | 1.10 | 1.08 | 1.05 |
| End-to-End, hidden=2048 | 1.34 | 1.18 | 1.15 | 1.36 | 1.21 | 1.29 |
| End-to-End, hidden=4096 | 1.27 | 1.23 | 1.18 | 1.37 | 1.24 | 1.32 |

*Table 11.* Speed up result on the RTX 3090 GPUs. SB refers to SwitchBack, Ours refers to Jetfire.

### D.2. Acceleration result on other hardware

Besides RTX 4090, we tested our linear operator and end-to-end speed up result on the RTX 3090 GPUs, as reported in Table 11. The results indicate that our method can achieves significant speedups on multiple kinds of GPUs.