A Implementation Details

In this section, we present some works that need to be done to actually accelerate the training process on hardware.

A.1 BMM in Attention

In attention, there are batch matrix multiplications (BMMs) that need to be dealt with. We now show that our method for MMs can be extended to BMMs.

Consider the following BMM product:

$$\mathbf{T} = \text{BMM}(\mathbf{Q}, \mathbf{K}^\top),$$

where we define $\mathbf{T} \in \mathbb{R}^{B\times N\times P}, \mathbf{Q} \in \mathbb{R}^{B\times N\times M}, \mathbf{K} \in \mathbb{R}^{B\times P\times M}$. The Hadamard matrix is defined as :

$$\hat{\mathbf{H}} = \text{Repeat}_B(\mathbf{H}) = \text{Repeat}_B(\text{BlockDiag}(\mathbf{H}_k, \ldots, \mathbf{H}_k)),$$

where $\hat{\mathbf{H}} \in \mathbb{R}^{B\times M\times M}, \mathbf{H} \in \mathbb{R}^{M\times M}, \mathbf{H}_k \in \mathbb{R}^{2^k\times 2^k}$. In this case,

$$\mathbf{T} \approx \text{BMM}\big(\text{BMM}(\mathbf{Q}, \hat{\mathbf{H}}), \text{BMM}(\mathbf{K}, \hat{\mathbf{H}})^\top\big),$$

which verifies that our HQ can be applied to BMMs.

For backward, the gradient of weight and activation can be calculated by the straight-through estimator $\lfloor x\rceil' = 1$ and the chain rule:

$$\begin{aligned}
&\nabla_{\mathbf{Q}} = s_Q\left(\text{BMM}(\nabla_{\mathbf{T}}^\top, \hat{\mathbf{K}}) \circ \mathbb{I}_Q\right)\mathbf{H}^\top,\\
&\nabla_{\mathbf{K}} = s_K \mathbb{I}_K \circ \text{BMM}(\nabla_{\mathbf{T}}, \hat{\mathbf{Q}})\mathbf{H}^\top = s_K \text{BMM}(\mathbb{I}_K \circ \nabla_{\mathbf{T}}, \hat{\mathbf{Q}})\mathbf{H}^\top,
\end{aligned}$$

where we define $s_Q \in \mathbb{R}^B, s_k \in \mathbb{R}^B$ being the batch step size, $\hat{\mathbf{K}} = \text{int}_{s_K}\left(\text{BMM}(\mathbf{K}, \hat{\mathbf{H}})\right)$, $\hat{\mathbf{Q}} = \text{int}_{s_Q}\left(\text{BMM}(\mathbf{Q}, \hat{\mathbf{H}})\right)$, $\mathbb{I}_Q = \mathbb{I}(-Q_N \leq \mathbf{Q}/s_Q \leq Q_P)$, and $\mathbb{I}_K = \mathbb{I}(-Q_N \leq \mathbf{K}/s_K \leq Q_P)$.

Similar to Sec. 4.2, we only focus on $\text{BMM}(\nabla_{\mathbf{T}}^\top, \hat{\mathbf{K}})$ and $\nabla_{\mathbf{T}}$, since we do leverage sampling on them.

For $\text{BMM}(\nabla_{\mathbf{T}}^\top, \hat{\mathbf{K}})$, we define the sample probability p_i and sample the $\tilde{\mathbf{M}}$ in the same way as MMs. The matrix can be computed as $\text{BMM}(\text{BMM}({\nabla_{\mathbf{T}}^{\updownarrow}}^\top, \hat{\tilde{\mathbf{H}}}), \hat{\mathbf{K}}^{\updownarrow})$, where $\hat{\tilde{\mathbf{H}}}$ is defined as $\text{CONCAT}(\tilde{\mathbf{H}}_1, \cdots, \tilde{\mathbf{H}}_B)$, ${\nabla_{\mathbf{T}}^{\updownarrow}}^\top$ and $\hat{\mathbf{K}}^{\updownarrow}$ follows the same definition of Eq. 6and the leverage score is $c_{b,i} := \|\nabla^{\updownarrow}_{\mathbf{T}b,i,:}\|\|\mathbf{K}^{\updownarrow}_{b,i,:}\|$ for $0 \leq b \leq B, 0 \leq i \leq 2M$.

For $\nabla_{\mathbf{T}}$, similarly, can be viewed as $\nabla_{\mathbf{T}} = \text{BMM}(\hat{\mathbf{I}}^{\updownarrow}, \nabla_{\mathbf{T}}^{\updownarrow})$,where we define $\nabla_{\mathbf{Y}}^{\updownarrow} = \text{CONCAT}([s_{\uparrow b}\nabla^{\uparrow}_{\mathbf{T}b}; s_{\downarrow b}\nabla^{\downarrow}_{\mathbf{T}b}]) \in \mathbb{R}^{B\times 2N\times P}$, $\hat{\mathbf{I}}^{\updownarrow} = \text{CONCAT}([\mathbf{I} \quad \mathbf{I}]) \in \mathbb{R}^{B\times N\times 2N}$, $s_{\uparrow b}, \nabla^{\uparrow}_{\mathbf{T}b}, s_{\downarrow b}, \nabla^{\downarrow}_{\mathbf{T}b}$ follows the definition of Eq.5. So it can be computed as $\text{BMM}(\text{BMM}(\hat{\mathbf{I}}^{\updownarrow}, \hat{\tilde{\mathbf{H}}}), \nabla_{\mathbf{T}}^{\updownarrow})$, where $\hat{\tilde{\mathbf{H}}}$ is defined as $\text{CONCAT}(\tilde{\mathbf{H}}_1, \cdots, \tilde{\mathbf{H}}_B)$, and the leverage score is $c_{b,i} := \|\nabla^{\updownarrow}_{\mathbf{T}b,i,:}\|$ for $0 \leq b \leq B, 0 \leq i \leq 2M$, which verifies that our LSS can be applied to BMM.

A.2 Computing Leverage Score

In the previous discussion, we find the optimal sample probability p_i that can minimize the variance of the gradient. However, it is likely for the proportional p_i is larger than one, which is invalid for the Bernoulli distribution. Accordingly, we propose an algorithm to solve this issue.

Define the probability array as

$$P = [p_1^0, \cdots, p_{2N}^0], \sum_{i=1}^{2N} p_i^0 = N,$$

we first clamp the array to $p_i^1 \in [0, 1]$. In this case, $\sum_{i=1}^{2N} p_i^1 \leq N$, so we scale the p_i which is smaller than 1 to make sure their sum is again N. However, this will probably introduce some more elements larger than 1, so we cycle through the above operations until all the $p_i \in [0, 1]$. This process will certainly stop, since if after the scaling operation, no element is larger than 1, then we get a valid distribution. Otherwise, the number larger than 1 is reduced by at least one, thus the process will halt after at most $O(N)$ times.

A.3 Learning Quantizer Parameters

In this section, we discuss the detail of how to calculate the gradient of activation and quantization step size.

For gradient of activation, the coefficient $c_i := \|\nabla_{\mathbf{Y}}^{\updownarrow}{}_i\|$ is the *leverage score* for activation gradient, and the variance achieves its minimum When $p_i \propto c_i$ by the Cauchy Inequality.

Putting everything together, we propose the following MM procedure to compute activation gradient:

```
Procedure LSS-MM
1. Quantize $\nabla_{\mathbf{Y}}$ with BS to obtain $\nabla_{\mathbf{Y}}^{\uparrow}$ and $\nabla_{\mathbf{Y}}^{\downarrow}$ in INT4.
2. Compute the leverage score $\|\nabla_{\mathbf{Y}}^{\updownarrow}{}_i\|$ in FP16.
3. Sample the masks $\{m_i\}$.
4. Sample rows of $\nabla_{\mathbf{Y}}$ given the masks $\{m_i\}$.
5. Compute $\tilde{\mathbf{IM}}^{\uparrow}\nabla_{\mathbf{Y}}^{\uparrow}$ and $\tilde{\mathbf{IM}}^{\downarrow}\nabla_{\mathbf{Y}}^{\downarrow}$ by discard some of its rows.
6. Compute INT4 MMs $\tilde{\mathbf{IM}}^{\uparrow}\nabla_{\mathbf{Y}}^{\uparrow}\hat{\mathbf{W}}$ and $\tilde{\mathbf{IM}}^{\downarrow}\nabla_{\mathbf{Y}}^{\downarrow}\hat{\mathbf{W}}$.
7. Dequantize and sum up the resultant INT32 matrices to obtain the FP16 result $\hat{\mathbf{I}}^{\updownarrow}\nabla_{\mathbf{Y}}^{\updownarrow}\hat{\mathbf{W}}$.
```

The two matrix multiplications in Step 5 take about $2NCD$ INT4 MACs in expectation.

For the quantization step sizes. Following the chain rule, we have

$$\nabla_{s_W} = g(s_W)\nabla_{\mathbf{Y}}^{\top}\hat{\mathbf{X}} \circ \delta_{\mathbf{W}}(s_W), \ \nabla_{s_X} = g(s_X)\nabla_{\mathbf{Y}}\hat{\mathbf{W}} \circ \delta_{\mathbf{X}}(s_X),$$

where we define $g(s_W) = 1/\sqrt{Q_p N_W}$, $g(s_X) = 1/\sqrt{Q_p N_X}$, N_W and N_X being the number of elements of weight and activation, $\delta_{\mathbf{X}}(s_X) = \mathrm{int}_{s_X}(\mathbf{X}) - \mathbb{I}_X \circ (\mathbf{X}/s_X)$, and $\delta_{\mathbf{W}}(s_W) = \mathrm{int}_{s_W}(\mathbf{W}) - \mathbb{I}_W \circ (\mathbf{W}/s_W)$.

Notice that for computing ∇_{s_W} and ∇_{s_X}, the most expensive MMs are $\nabla_{\mathbf{Y}}^{\top}\hat{\mathbf{X}}$ and $\nabla_{\mathbf{Y}}\hat{\mathbf{W}}$, which are already calculated through Eq. (7) and Eq. (8) during previous calculations, so it does not require extra computation. The elementwise multiplication with $\delta_{\mathbf{X}}(s_X)$ and $\delta_{\mathbf{W}}(s_W)$ requires minor computation.

A.4 Cold Start Problem

There is a *cold start problem.* When the model is trained from scratch (i.e., from a random initialization), distributions of weights and activations can change rapidly in the early stage of optimization. In this case, jointly optimizing the quantization step size and the weights would cause the training to be unstable. As a remedy, we do not learn the step size in the first few iterations, and use a heuristic rule to dynamically set the step size for each tensor $\mathbf{X}$ to $2\mathrm{mean}(\mathbf{X})/\sqrt{Q_p}$ in each iteration.

A.5 Choose hadamard matrix size

For the hadamard matrix, let the hadamard matrix to be $\mathbf{H} \in \mathbb{R}^{D \times D}$: $\mathbf{H} = \mathrm{BlockDiag}(\mathbf{H}_k, \ldots, \mathbf{H}_k)$, where D is a multiple of 2^k. We first define

$$\bar{\mathbf{X}}_k = s_X \mathrm{int}_{s_X}(\mathbf{X}\mathbf{H})\,\mathbf{H}^{\top}, \quad \bar{\mathbf{W}} = s_W \mathrm{int}_{s_W}(\mathbf{W}\mathbf{H})\,\mathbf{H}^{\top},$$

where $\bar{\mathbf{X}}$ and $\bar{\mathbf{W}}$ can be viewed as an approximation of $\mathbf{X}$ and $\mathbf{W}$. Then, we define the quantization error to be $\mathrm{MSE}(\bar{\mathbf{X}}, \mathbf{X}) \times \mathrm{MSE}(\bar{\mathbf{W}}, \mathbf{W})$. We search for the optimal k that can minimize this quantization error. For fine-tuning tasks, once the hadamard matrix size has been calculated, we fix it through the training process. For the pre-training task, since the distribution shifts greatly as we train the model, we empirically define a time when we re-initialize the hadamard matrix size and the LSQ step size. Usually, we do this when the first 2 epochs finish.

A.6 GPU Implementation

In the previous discussion, we get to know `HQ-MM` and `LSS-MM` from an algorithm level, nevertheless it is not enough to actually implement it on hardware. In this section, we will delve deeper into hardware implementation details as well as extra limitations.

`HQ-MM` can be divided into 5 parts: Hadamard matrix multiplication, Quantize, Data Pack, INT4 GEMM, and Dequantize.

For the Hadamard matrix multiplication process, since it can be interpreted as a half float matrix multiplication process where the two matrices involved in the operation are input/weight matrix and hadamard matrix, respectively, we implement it in Python, because PyTorch MM uses CublassGemm and is more efficient then CutlassGemm.

In the quantize process, we quantize input/weight into INT4 data respectively, and also preserve a corresponding FP16 version for the LSQ Back Propagation process to use.

In the previous discussion, we assume the quantize part of `HQ-MM` is quantizing the resultant matrices to INT4, however, the smallest representation unit of data is INT8. As a result, we actually use INT8 data type to represent quantized data and pack two adjacent data into one data using $(data[1] << 4)|(data[0]\&15)$ in the data packing process, which means we use one INT8 data to represent two adjacent INT4 data. With both input matrices' data packed in this way, we then use cutlass tensor-core INT4 GEMM to do the matrix multiplication.

For the GEMM process, we choose Nvidia CutlassGemm because it's the most efficient open-source operator library we can find. We use INT4 Tensor Core Gemm for our implementation and it requires the two input matrices A&B to be RowMajor and ColMajor, respectively. Since the default Pytorch tensor is RowMajor, we have to use Transpose+Contiguous operations to make it ColMajor, which is very time-consuming and needs further optimization in the future.

Finally, we dequantize the INT GEMM result back into FP16 output using a dequantize kernel, which is the final output of the forward kernel.

As compared, `LSS-MM` is more complicated, and can be divided into 7 parts: Quantization of higher lower 4-bit, Leverage Score Calculating, Sampling, Data Pack, INT4 GEMM, Dequantize, and LSQ Back Propagation.

In the Quantize process, we fuse the quantize operation of higher 4-bit and lower 4-bit into a single kernel for acceleration. In the Leverage Score Calculating process, we use the quantized INT8 data to calculate the score and scale up it in the final because integer arithmetic is far more efficient than float arithmetic.

In the sampling process, we sample out rows/columns given the previously calculated leverage score. Note that in Section. A.2, we repeat our proposed algorithm for several loops to sample out specific elements, which is effective but not efficient. According to experiments, however, we notice that simply selecting elements whose leverage score is bigger than 0 can also work well, even better than our proposed algorithm in some cases. So in real quantization implementation, we just sample out rows/ columns whose Euclidean norm is bigger than 0 to accelerate our training process.

Pack, Gemm, and Dequantize processes are as similar as before. It's worth noting that for Int4 Tensor Core Gemm, suppose two input matrices have shape $M \times K$ and $K \times N$, K needs to be a multiple of 32 so that the Tensor core Gemm address can be aligned. We do not need to consider this in the Forward Propagation process because the input data shape always satisfies. However, in the Back Propagation process, the matrix shape may not meet the requirement after sampling. As a result, we need zero_padding the sampled matrix so that K can be a multiple of 32.

Finally, we utilize the dequantized data to do the LSQ Back Propagation. We also fuse all operations into a single Cuda kernel for acceleration, and the metric remains.

Besides the component of `HQ-MM` and `LSS-MM` , there is still something that needs to be mentioned.

1. We omit the Quantization and Leverage Score Calculating process in LSSinput, and use the same value as LSSWeight to accelerate the training process.
2. For Element-Wise kernel, we set block size as 256, grid size as input.numel()/256. For Reduction kernels like sum and min/max, we set block size as 32, grid size as RowNum,

reducing elements in each row to the first 32 elements. We find this setting to be most efficient through experiments.

B Proofs.

In this section, we present the proofs of the leverage score.

B.1 Proof of Proposition. 4.1

Proposition B.1. *(LSS variance for weight gradient)*

$$\mathrm{Var}\left[\sum_{i=1}^{2N} \frac{m_i}{p_i} \nabla^{\updownarrow}_{\mathbf{Y}\,:,i}{}^{\top} \mathbf{X}^{\updownarrow}_{i}\right] = \sum_{i=1}^{2N} \frac{1-p_i}{p_i} \|\nabla^{\updownarrow}_{\mathbf{Y}\,i,:}\|^2 \|\mathbf{X}^{\updownarrow}_{i,:}\|^2.$$

Proof.

$$\begin{aligned}
Var(\nabla_{\mathbf{W}}) &= Var\Big(\sum_{i=1}^{2N} \frac{1}{p_i}(m_i \nabla^{\updownarrow}_{\mathbf{Z}:,i}{}^{\top} \mathbf{X}^{\updownarrow}_{i})\Big) \\
&= Var\Big(\sum_{i=1}^{2N} \frac{1}{p_i}(\sum_{j=1}^{C}\sum_{k=1}^{D} m_i \nabla^{\updownarrow}_{\mathbf{Z}j,i}{}^{\top} \mathbf{X}^{\updownarrow}_{i,k})\Big) \\
&= \sum_{i=1}^{2N} \frac{p_i(1-p_i)}{p_i^2} Var\Big((\sum_{j=1}^{C}\sum_{k=1}^{D} \nabla^{\updownarrow}_{\mathbf{Z}j,i}{}^{\top} \mathbf{X}^{\updownarrow}_{i,k})\Big) \\
&= \sum_{i=1}^{2N} \frac{1-p_i}{p_i}(\sum_{j=1}^{C}\sum_{k=1}^{D} \nabla^{\updownarrow}_{\mathbf{Z}j,i}{}^{\top 2} \mathbf{X}^{\updownarrow}_{i,k}{}^{2}).
\end{aligned}$$

□

So that

$$\begin{aligned}
Var(\nabla_{\mathbf{W}}) &= \sum_{i=1}^{2N}(\frac{1}{p_i}-1)(\sum_{j=1}^{C} \nabla^{\updownarrow}_{\mathbf{Z}j,i}{}^{\top 2})(\sum_{k=1}^{D} \mathbf{X}^{\updownarrow}_{i,k}{}^{2}) && (9) \\
&= \sum_{i=1}^{2N}(\frac{1}{p_i}-1)\|\nabla^{\updownarrow}_{\mathbf{Z}:,i}{}^{\top}\|^2 \|\mathbf{X}^{\updownarrow}_{i,:}\|^2, && (10)
\end{aligned}$$

which proves.

B.2 Proof of Activation Leverage Score in Sec. 4.2

we divide the matrix multiplication into the sum of $2N$ smaller multiplications:

$$\hat{\mathbf{I}}^{\updownarrow}\nabla^{\updownarrow}_{\mathbf{Y}} = \sum_{i=1}^{2N} \hat{\mathbf{I}}^{\updownarrow}_{:,i}\nabla^{\updownarrow}_{\mathbf{Y}i} = \sum_{i=1}^{2N} \hat{\nabla}_{\mathbf{Y}_i}, \tag{11}$$

where we define $\hat{\nabla}_{\mathbf{Y}_i} = \hat{\mathbf{I}}^{\updownarrow}_{:,i}\nabla^{\updownarrow}_{\mathbf{Y}i}$.

We assigns each $\nabla_{\mathbf{Y}_i}$ a probability $p_i \in [0,1], i = 1, \cdots, 2N$, that satisfies $\sum_{i=1}^{2N} p_i = N$. We define random masks $m_i \sim \mathrm{Bern}(p_i)$, and define $\tilde{\mathbf{M}} = \mathrm{diag}\left(\frac{m_1}{p_1}, \ldots, \frac{m_{2N}}{p_{2N}}\right)$, and make an unbiased estimation:

$$\hat{\mathbf{I}}^{\updownarrow}\nabla^{\updownarrow}_{\mathbf{Y}} \approx \hat{\mathbf{I}}^{\updownarrow}\tilde{\mathbf{M}}\nabla^{\updownarrow}_{\mathbf{Y}} = \sum_{i=1}^{2N} \frac{m_i}{p_i}\nabla^{\updownarrow}_{\mathbf{Y}i}.$$

Define $\mathbf{M}^{\uparrow}$ to be the top-left $N \times N$ submatrix of $\mathbf{M}$ and $\mathbf{M}^{\downarrow}$ to be the bottom-right one, we have

$$\hat{\mathbf{I}}^{\updownarrow}\tilde{\mathbf{M}}\nabla_{\mathbf{Y}}^{\updownarrow} = s_{\uparrow}\mathbf{I}\tilde{\mathbf{M}}^{\uparrow}\nabla_{\mathbf{Y}}^{\uparrow} + s_{\downarrow}\mathbf{I}\tilde{\mathbf{M}}^{\downarrow}\nabla_{\mathbf{Y}}^{\downarrow},$$

In this case, $\mathbf{I}\tilde{\mathbf{M}}^{\uparrow}\nabla_{\mathbf{Y}}^{\uparrow}$ and $\mathbf{I}\tilde{\mathbf{M}}^{\downarrow}\nabla_{\mathbf{Y}}^{\downarrow}$ both only have parts of its rows being non zero, and the rest rows are zeros since they are discarded. Then, when we multiply it by $\hat{\mathbf{W}}$, there are half of rows being zeros in $\mathbf{I}\tilde{\mathbf{M}}^{\uparrow}\nabla_{\mathbf{Y}}^{\uparrow}\hat{\mathbf{W}}$ and $\mathbf{I}\tilde{\mathbf{M}}^{\downarrow}\nabla_{\mathbf{Y}}^{\downarrow}\hat{\mathbf{W}}$. So there's no need to calculate them, and we successfully cut off half of the computation in this case.

Now focus on the variance that

Proposition B.2. *(LSS variance for activation gradient)*

$$\mathrm{Var}\left[\sum_{i=1}^{2N}\hat{\mathbf{I}}_{:,i}^{\updownarrow}\nabla_{\mathbf{Y}i}^{\updownarrow}\right] = \sum_{i=1}^{2N}\frac{1-p_i}{p_i}\|\nabla_{\mathbf{Y}i}^{\updownarrow}\|^2.$$

Proof.

$$\begin{aligned}
Var(\nabla_{\mathbf{X}}) &= Var\Big(\sum_{i=1}^{2N}\frac{1}{p_i}(m_i\hat{\mathbf{I}}_{:,i}^{\updownarrow}\mathbf{X}_i^{\updownarrow})\Big) \\
&= Var\Big(\sum_{i=1}^{2N}\frac{1}{p_i}(\sum_{j=1}^{C}\sum_{k=1}^{D}m_i\hat{\mathbf{I}}_{j,i}^{\updownarrow}\nabla_{\mathbf{Y}i,k}^{\updownarrow})\Big) \\
&= \sum_{i=1}^{2N}\frac{p_i(1-p_i)}{p_i^2}Var\Big((\sum_{j=1}^{C}\sum_{k=1}^{D}\hat{\mathbf{I}}_{j,i}^{\updownarrow}\nabla_{\mathbf{Y}i,k}^{\updownarrow})\Big) \\
&= \sum_{i=1}^{2N}\frac{1-p_i}{p_i}\big(\sum_{j=1}^{C}\sum_{k=1}^{D}(\hat{\mathbf{I}}_{j,i}^{\updownarrow})^2(\nabla_{\mathbf{Y}i,k}^{\updownarrow})^2\big) \\
&= \sum_{i=1}^{2N}(\frac{1}{p_i}-1)\big(\sum_{j=1}^{C}(\hat{\mathbf{I}}_{j,i}^{\updownarrow})^2)(\sum_{k=1}^{D}(\nabla_{\mathbf{Y}i,k}^{\updownarrow})^2\big) \\
&= \sum_{i=1}^{2N}(\frac{1}{p_i}-1)\|\hat{\mathbf{I}}_{:,i}^{\updownarrow}\|^2\|\nabla_{\mathbf{Y}i}^{\updownarrow}\|^2 \\
&= \sum_{i=1}^{2N}(\frac{1}{p_i}-1)\|\nabla_{\mathbf{Y}i}^{\updownarrow}\|^2.
\end{aligned}$$

$\square$

In this way, the coefficient $c_i := \|\nabla_{\mathbf{Y}i}^{\updownarrow}\|$ is the *leverage score*.

C Experiments.

In this section, we present more details for experiments in Sec. 5.

C.1 Experiments setup

For the GLUE, QA, SWAG, and CONLL tasks, we implement our algorithm based on `https://github.com/huggingface/transformers`. For the machine translation task, we implement our algorithm based on `https://github.com/facebookresearch/fairseq`. For the ViT fine-tuning task, we implement our algorithm based on `https://github.com/jeonsworld/ViT-pytorch`. For the deit pretraining task, we implement our algorithm based on `https://github.com/facebookresearch/deit`.

We employed NVIDIA GeForce RTX 3090 for running most of the experiments, while the NVIDIA A40 was utilized to evaluate the performance of BERT-Large and ViT-L. Furthermore, we conducted runtime measurements using the NVIDIA T4, 3090, and A100 GPUs.

Table 2: GLUE results on BERT-base-uncased and BERT-large uncased. FP refers to full precision training, INT8 refers to INT8 training, LSQ + LUQ refers to learned step size quantization for forward and logarithmic unbiased quantization for backward, and HQ + LSS refers to Hadamard quantization for forward and leverage score sampling for backward.

		QUANTIZATION METHODS			
MODEL	DATASET	FP	INT8	LSQ+LUQ	HQ+LSS
BERT-BASE	COLA	$56.89_{0.64}$	$56.15_{0.94}$	$18.76_{3.58}$	$\mathbf{52.46_{1.46}}$
	STSB	$88.14_{0.73}$	$87.05_{0.38}$	$84.31_{0.29}$	$\mathbf{87.77_{0.30}}$
	RTE	$64.80_{1.26}$	$62.27_{1.26}$	$56.80_{0.92}$	$\mathbf{62.45_{1.08}}$
	MRPC	$88.61_{0.66}$	$86.85_{0.76}$	$86.23_{0.67}$	$\mathbf{86.54_{0.83}}$
	SST2	$92.72_{0.06}$	$92.37_{0.17}$	$90.37_{0.46}$	$\mathbf{92.49_{0.29}}$
	QNLI	$91.52_{0.22}$	$90.92_{0.24}$	$87.33_{0.48}$	$\mathbf{90.53_{0.23}}$
	QQP	$91.09_{0.11}$	$90.57_{0.05}$	$89.26_{0.03}$	$\mathbf{89.80_{0.05}}$
	MNLI	$84.52_{0.22}$	$84.10_{0.08}$	$81.79_{0.18}$	$\mathbf{83.59_{0.12}}$
	MNLI-MM	$84.68_{0.20}$	$84.49_{0.31}$	$82.22_{0.33}$	$\mathbf{83.75_{0.28}}$
BERT-LARGE	COLA	$60.33_{0.49}$	$58.80_{1.52}$	$0.00_{0.00}$	$\mathbf{53.46_{1.17}}$
	STSB	$87.59_{2.39}$	$86.53_{0.20}$	$83.08_{0.41}$	$\mathbf{87.57_{0.78}}$
	RTE	$71.12_{1.80}$	$63.71_{1.26}$	$53.06_{0.72}$	$\mathbf{64.62_{0.78}}$
	MRPC	$91.06_{0.28}$	$87.57_{1.47}$	$82.56_{0.59}$	$\mathbf{87.62_{0.51}}$
	SST2	$93.98_{0.17}$	$93.75_{0.63}$	$83.94_{0.69}$	$\mathbf{93.52_{0.40}}$
	QNLI	$92.26_{0.05}$	$92.29_{0.29}$	$63.18_{13.10}$	$\mathbf{91.53_{0.38}}$
	QQP	$91.04_{0.63}$	$90.71_{0.00}$	$75.62_{12.44}$	$\mathbf{90.77_{0.02}}$
	MNLI	$86.71_{0.19}$	$85.82_{0.08}$	$33.42_{1.38}$	$\mathbf{85.86_{0.10}}$
	MNLI-MM	$86.41_{0.35}$	$85.87_{0.14}$	$33.54_{1.55}$	$\mathbf{85.82_{0.07}}$

C.2 GLUE results

In this section, we present the detailed result of fine-tuning the GLUE dataset on BERT-base-uncased and BERT-large-uncased.

On BERT-base, on STSB, SST2, QNLI, and QQP, HQ+LSS only has $< 0.5\%$ accuracy degradation. On the most challenging tasks CoLA and RTE, our accuracy degradation is much smaller compared to LSQ+LUQ. On QQP and MNLI, our method achieves $< 1.3\%$ degradation, while LSQ + LUQ has $\geq 1.8\%$ degradation. The trend is that the more difficult the task is, the more significant our advantage over LSQ+LUQ.

On BERT-large, the improvement is significant. On CoLA, QNLI, and MNLI, the accuracy improvement compared with LSQ+LUQ $> 30\%$. On other datasets like SST2 and QQP, the accuracy improvement is $> 10\%$. On RTE the accuracy improvement is $> 5\%$, and on STSB and MRPC the improvement is $> 3\%$.

We suspect that for those challenging tasks, there is more information stored in the outliers, which results in a larger gap between our method and LSQ+LUQ.

C.3 More Granular Quantization Methods

In this section, in Table 4, we show that the more granular quantization methods, such as per-token quantization and per-channel quantization, or smoothing techniques, such as SmoothQuant, do not work under the 4-bit FQT setting. Meanwhile, combining these methods with HQ will not bring significant improvement.

We find that LSQ is beneficial for all of these more granular quantization methods under low-bit settings, which highlights the importance of LSQ. Meanwhile, we also notice that the smoothquant will even harm the result of LSQ when the bit-width is low. Our explanation is that the motivation of LSQ is to learn a trade-off between outliers and inliers, while smoothquant aims to sacrifice the

Table 3: Experiments on GPT2-base and Bert-large. Total time spent for epoch 1-5 are reported.

		TRAINING METHODS		
MODEL	(HIDDEN_SIZE, INTERMIDIATE_SIZE, BATCH_SIZE)	FP16	HQ+LSS	SPEEDUP
BERT-LARGE	(2560, 10240, 2048)	15.094s	18.949s	**−25.5%**
	(4096, 16384, 1280)	32.016s	30.594s	**4.4%**
	(5120, 20480, 960)	47.418s	39.482s	**16.7%**
	(7680, 30720, 600)	95.832s	67.253s	**29.8%**
	(8960, 35840, 480)	128.441s	83.388s	**35.1%**
	(9600, 38400, 160)	161.114s	114.325s	**29.0%**
	(12800, 51200, 100)	326.265s	255.966s	**21.5%**
	(14400, 57600, 96)	409.291s	346.354s	**15.3%**
GPT2-BASE	(2560, 10240, 1536)	17.253s	22.037s	**−27.7%**
	(4096, 16384, 960)	35.937s	35.694s	~
	(5120, 20480, 768)	52.723s	46.548s	**11.7%**
	(7680, 30720, 260)	113.855s	92.548s	**18.7%**
	(8960, 35840, 200)	150.680s	114.881s	**23.8%**
	(9600, 38400, 180)	172.182s	126.540s	**26.5%**
	(12800, 51200, 112)	320.757s	236.433s	**26.3%**

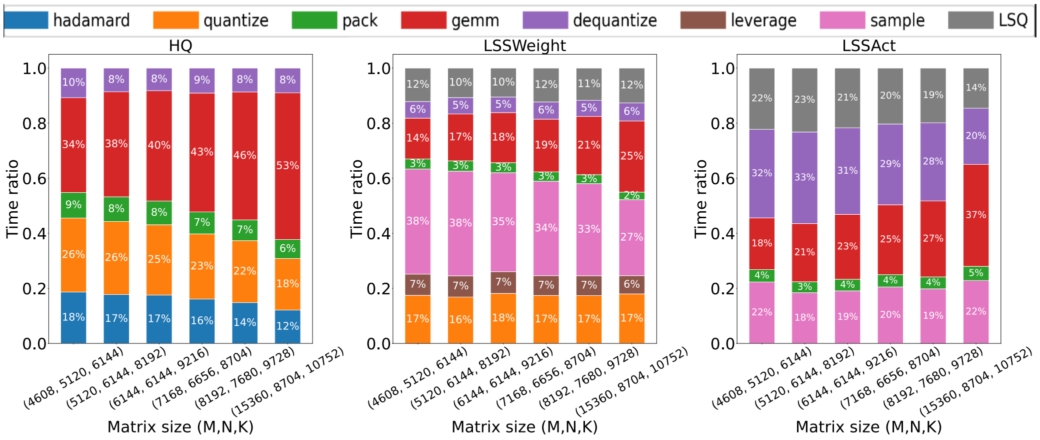

Figure 6: Time proportion for each part in `HQ-MM` and `LSS-MM` operator.

precision of inliers in order to exactly maintain the information of outliers. When the bitwidth is high, this is not a problem, since there are still enough bits to quantize the inliers. But when the bitwidth is low, such sacrifice will cause severe problems since the inlier information is discarded.

C.4 Large Language Model Operator Speed

In this section, we show that our hardware-friendly INT4 training method can really accelerate the training process on Large Language Models. We run distributed training on a system of 8 A100 cards and our implementation uses distributed data parallel training with zero-3, gradient checkpointing, and optimizer offloading.

We experimented with two architectures: BERT-Large and GPT2-base. We vary the network width and batch size to make full utilization of the GPU memory and show the end-to-end performance for fine-tuning these models on the SuperGLUE RTE dataset in Table 3.

C.5 More experiments on Operator Speed

Time proportion We examine the proportion of time for each part of computation in `HQ-MM` and `LSS-MM` operator in Fig. 6 when the shapes of input matrices vary. In HQ, hadamard means multiplying the input matrix with the Hadamard matrix, pack means packing input data into INT4 data, gemm means the matrix multiplication of two INT4 matrices. In LSSWeight, quantize corresponds to

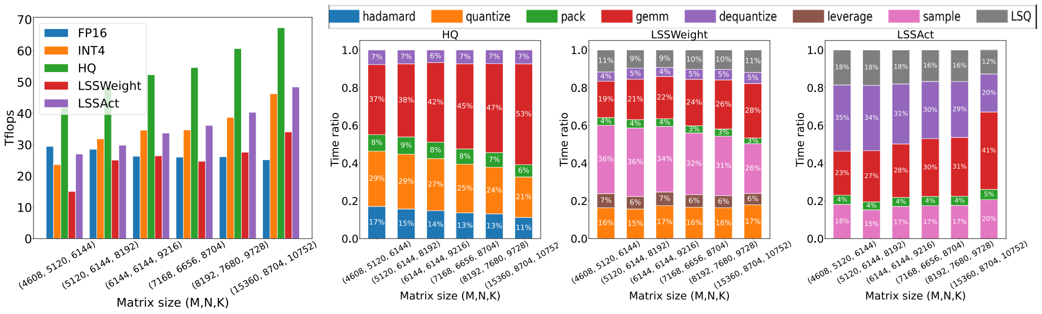

Figure 7: Real quantization performance on Nvidia T4.

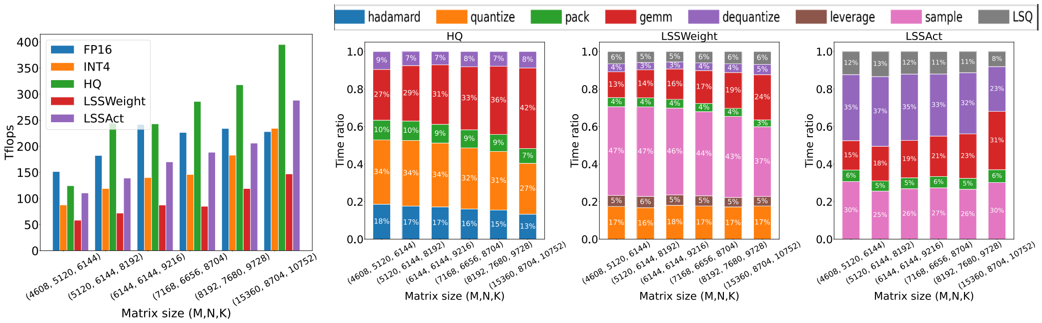

Figure 8: Real quantization performance on Nvidia A100.

Table 4: Comparison of different quantization methods, quantize the activation and weight into the same bit-width from 2 to 8. Per-token refers to quantize activation per-token, while Per-channel refers to quantize weight per-channel.

	Quantize Bits						
quantization methods	2	3	4	5	6	7	8
Per-tensor	0	0	0	0	0	50.2	54.6
Per-token	0	0	0	0	31.4	52.8	56
Per-channel	0	0	0	0	0	51.9	56.7
smoothquant	0	0	0	0	0	49.4	57.7
Per-token + Per-channel + smoothquant	0	0	0	0	40.7	55.7	56.7
LSQ	0	9.16	24.2	37.3	39.6	45.3	51.4
Per-token + LSQ	0	15.3	27.8	31.6	42.9	46	54.4
Per-channel + LSQ	0	8	23.9	29.3	40	45.5	50.7
smoothquant + LSQ	0	0	0	0	49.6	54.9	57
Per-token + Per-channel + smoothquant + LSQ	0	0	0	0	28.8	52.4	55.2
HQ	0	45.2	54.6	54.2	56.5	57.4	58.4
HQ + Per-token + Per-channel	0	48.4	54.1	54.9	55	56	56
HQ + Per-token + Per-channel + smoothquant	0	0	46.6	54.9	55.9	55.8	56.5

the quantization of higher and lower 4-bit, leverage means computing leverage score, sample means sample out rows/columns given the leverage score, dequantize is the process of dequantizing INT data back into FP16 data, and LSQ is the backpropagation process of LSQ method. In LSSAct, we ignore quantize and leverage process, using the same value as LSSWeight for saving time, other processes share the same meaning with LSSWeight. Note that our implementation is not fully optimized, and optimizations like operator fusion can further improve the performance.

Operator Speed on more GPUs On an Nvidia RTX 3090 GPU with a Cuda capability of sm_86., we show the comparison of FP16 MM, HQ, and LSS operators in Section 5.3 as well as time proportion in each operator in Figure. 6. We also adjust our hardware implementation and test its performance on Nvidia T4 GPU and Nvidia A100 GPU, which have Cuda capability of sm_75 and sm_80 , respectively. The result is shown in Fig. 7 and Fig. 8.