SmoothQuant

MIT, NVIDIA 2023
Introduction
Serving Large Language Models Is Expensive

- Pre-trained language models have achieved remarkable performance

- Large language models often come in high precision formats such as FP16
  - Significant GPU memory requirements (size and bandwidth)
  - Slow matrix multiplication operations
Post-Training Quantization Reduces The Cost

- Post-training quantization (PTQ) reduces the cost of LLMs.
  - “Post-training:” no modifications to training => easy to implement and wide applicability
  - “Quantization:” lowers the bit-width and improves efficiency
  - Mitigate memory consumption and reduce computational overhead => higher performance
Existing methods cannot maintain accuracy and hardware efficiency at the same time

- **ZeroQuant**
  - Uses layer-by-layer knowledge distillation without the original training data
  - Delivers good accuracy for GPT-3-350M and GPT-J-6B
  - Can not maintain the accuracy for the large OPT model with 175 billion parameters

- **LLM.int8()**
  - Increases accuracy by keeping outliers in FP16 and uses INT8 for the other activations
  - The mixed-precision decomposition is not hardware friendly
Paper Key Takeaways

● Per-channel quantization is infeasible
  ○ GEMM kernels rely on a sequence of operations at a high throughput
  ○ But operations that apply different scales for each channel have a lower throughput

● Preprocessing the weights and activations is the solution
  ○ Activations are hard to quantize and weights are easy to quantize
  ○ Exploit the linearity of matrix multiplication to offload the quantization difficulty
  ○ Uniform quantization is supported by hardware

● SmoothQuant enables an INT8 quantization of both weights and activations for all the matrix multiplications in LLMs
  ○ Preserves accuracy
  ○ Hardware-friendly
  ○ Up to 1.56x speedup and 2x memory reduction
The Quantization Process

\[
\bar{X}_{\text{INT8}} = \left\lfloor \frac{X_{\text{FP16}}}{\Delta} \right\rfloor, \quad \Delta = \frac{\text{max}(|X|)}{2^{N-1} - 1}
\]

Fp16 vector

1.2  -0.5  -4.3  1.2  -3.1  0.8  2.4  5.4

Get max(abs)

5.4

Get quantisation factor \(\frac{1}{\Delta}\)

[-127, 127]

Quantized - int8 vector

28  -12  -101  28  -73  19  56  127
The Quantization Process

\[
\bar{X}_{\text{INT8}} = \left\lfloor \frac{X_{\text{FP16}}}{\Delta} \right\rfloor, \quad \Delta = \frac{\max(|X|)}{2^{N-1} - 1}
\]

How to get \( \Delta \)?

- **Dynamic Range Quantization**
  - At runtime
  - Use the runtime statistics of activations to get \( \Delta \)

- **Static Quantization**
  - Before runtime
  - Calculate \( \Delta \) offline with the activations of some calibration samples
  - This paper gets activation statistics from 512 random sentences from the pre-training dataset Pile
Quantization Granularity

- **Per-Tensor Quantization**
  - Uses a single step size for the entire matrix
- **Per-Token Quantization**
- **Per-Channel Quantization**
- **Group-Wise Quantization**
Quantization Granularity

- Per-Tensor Quantization
- Per-Token Quantization
  - Uses different quantization step sizes for activations associated with each token
- Per-Channel Quantization
- Group-Wise Quantization
Quantization Granularity

- Per-Tensor Quantization
- Per-Token Quantization
- Per-Channel Quantization
  - Uses different quantization step sizes for activations associated with each output channel of weights
- Group-Wise Quantization
Quantization Granularity

- Per-Tensor Quantization
- Per-Token Quantization
- Per-Channel Quantization
- Group-Wise Quantization
  - Different quantization steps for different channel groups
Key Idea #1: Per-Channel Quantization is Infeasible

- **Observations:**
  - Outliers lead to low effective quantization bits
  - Outliers exist in a small fraction of channels

- **Reasonable thought:**
  - If we could perform per-channel quantization, the quantization error would be much smaller compared to per-tensor quantization
Key Idea #1: Per-Channel Quantization is Infeasible

- Issue:
  - Hardware-accelerated GEMM kernels, that rely on a sequence of operations executed at a high throughput
  - Per-channel activation quantization relies on insertion of instructions with a lower throughput to apply different scales for each channel
  - GEMM kernels do not tolerate the insertion of instructions with a lower throughput
Key Idea #2: Migrating the quantization difficulty

Weights are easy to quantize, but activations are hard due to outliers

\[ Y = XW = (0.01X)(100W) \]
Key Idea #2: Migrating the quantization difficulty

● Issue from before: Activation outliers persist in fixed channels
  ○ Per-tensor quantization is limited
  ○ But per-channel quantization was infeasible.

● After preprocessing (smoothing): Linearity is exploited so that weights and activations can have similar degrees of outliers
  ○ Per-tensor quantization is effective again
  ○ Bake smoothing factor into previous layers (or residual branch for residual add)
SmoothQuant’s Per-Channel Smoothing Factor

\[ Y = (X \text{diag}(s)^{-1}) \cdot (\text{diag}(s)W) = \hat{X}\hat{W} \]

Push all quantization difficulty from activations to weights (all channels have same maximum magnitude):

\[ s_j = \max(|X_j|), j = 1, 2, ..., C_i. \]

Push all quantization difficulty from weights to activations:

\[ s_j = 1 / \max(|W_j|) \]

Share difficulty according to \( \alpha \):

\[ s_j = \frac{\max(|X_j|)^\alpha}{\max(|W_j|)^{1-\alpha}} \]
Choosing $\alpha$

Case-by-case decision

If the $\alpha$ is too large, weights will be hard to quantize. If too small, activations will be hard to quantize.

Goal: make activations and weights both easy to quantize.
Example of SmoothQuant

1) Apply Smoothing Factor
2) Quantize (constant step size)
SmoothQuant Hardware Efficiency

- Applying SmoothQuant to transformer blocks
  - Linear layers take up most of the parameters and computation
  - Smoothing factor can be fused into previous layers’ parameters offline
  - All linear layers are quantized with W8A8, as well as BMM operators in Attention computation
Four Baselines

LLM.int8 keeps outliers in FP16 (large latency overhead). W8A8 is the naive implementation. Outlier suppression uses token-wise clipping.

<table>
<thead>
<tr>
<th>Method</th>
<th>Weight</th>
<th>Activation</th>
</tr>
</thead>
<tbody>
<tr>
<td>W8A8</td>
<td>per-tensor</td>
<td>per-tensor dynamic</td>
</tr>
<tr>
<td>ZeroQuant</td>
<td>group-wise</td>
<td>per-token dynamic</td>
</tr>
<tr>
<td>LLM.int8()</td>
<td>per-channel</td>
<td>per-token dynamic + FP16</td>
</tr>
<tr>
<td>Outlier Suppression</td>
<td>per-tensor</td>
<td>per-tensor static</td>
</tr>
</tbody>
</table>
SmoothQuant O1 to O3

Gradually aggressive and efficient (lower latency) quantization levels

<table>
<thead>
<tr>
<th>Method</th>
<th>Weight</th>
<th>Activation</th>
</tr>
</thead>
<tbody>
<tr>
<td>SmoothQuant-O1</td>
<td>per-tensor</td>
<td>per-token dynamic</td>
</tr>
<tr>
<td>SmoothQuant-O2</td>
<td>per-tensor</td>
<td>per-tensor dynamic</td>
</tr>
<tr>
<td>SmoothQuant-O3</td>
<td>per-tensor</td>
<td>per-tensor static</td>
</tr>
</tbody>
</table>
Evaluation

- Three families of LLMs
  - OPT
    - $\alpha = 0.5$
  - BLOOM
    - $\alpha = 0.5$
  - GLM-130B
    - $\alpha$ is set to 0.75 since its activations are more difficult to quantize
- Seven zero-shot evaluation tasks e.g. LAMBADA, WikiText
- Focus on *relative* performance chance before/after quantization
## OPT-175B results

<table>
<thead>
<tr>
<th>OPT-175B</th>
<th>LAMBADA</th>
<th>HellaSwag</th>
<th>PIQA</th>
<th>WinoGrande</th>
<th>OpenBookQA</th>
<th>RTE</th>
<th>COPA</th>
<th>Average↑</th>
<th>WikiText↓</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP16</td>
<td>74.7%</td>
<td>59.3%</td>
<td>79.7%</td>
<td>72.6%</td>
<td>34.0%</td>
<td>59.9%</td>
<td>88.0%</td>
<td>66.9%</td>
<td>10.99</td>
</tr>
<tr>
<td>W8A8</td>
<td>0.0%</td>
<td>25.6%</td>
<td>53.4%</td>
<td>50.3%</td>
<td>14.0%</td>
<td>49.5%</td>
<td>56.0%</td>
<td>35.5%</td>
<td>93080</td>
</tr>
<tr>
<td>ZeroQuant</td>
<td>0.0%*</td>
<td>26.0%</td>
<td>51.7%</td>
<td>49.3%</td>
<td>17.8%</td>
<td>50.9%</td>
<td>55.0%</td>
<td>35.8%</td>
<td>84648</td>
</tr>
<tr>
<td>LLM.int8()</td>
<td>74.7%</td>
<td>59.2%</td>
<td>79.7%</td>
<td>72.1%</td>
<td>34.2%</td>
<td>60.3%</td>
<td>87.0%</td>
<td><strong>66.7%</strong></td>
<td><strong>11.10</strong></td>
</tr>
<tr>
<td>Outlier Suppression</td>
<td>0.00%</td>
<td>25.8%</td>
<td>52.5%</td>
<td>48.6%</td>
<td>16.6%</td>
<td>53.4%</td>
<td>55.0%</td>
<td>36.0%</td>
<td>96151</td>
</tr>
<tr>
<td>SmoothQuant-O1</td>
<td>74.7%</td>
<td>59.2%</td>
<td>79.7%</td>
<td>71.2%</td>
<td>33.4%</td>
<td>58.1%</td>
<td>89.0%</td>
<td>66.5%</td>
<td><strong>11.11</strong></td>
</tr>
<tr>
<td>SmoothQuant-O2</td>
<td>75.0%</td>
<td>59.0%</td>
<td>79.2%</td>
<td>71.2%</td>
<td>33.0%</td>
<td>59.6%</td>
<td>88.0%</td>
<td>66.4%</td>
<td><strong>11.14</strong></td>
</tr>
<tr>
<td>SmoothQuant-O3</td>
<td>74.6%</td>
<td>58.9%</td>
<td>79.7%</td>
<td>71.2%</td>
<td>33.4%</td>
<td>59.9%</td>
<td>90.0%</td>
<td><strong>66.8%</strong></td>
<td><strong>11.17</strong></td>
</tr>
</tbody>
</table>
## Results On Different LLMs

<table>
<thead>
<tr>
<th>Method</th>
<th>OPT-175B</th>
<th>BLOOM-176B</th>
<th>GLM-130B*</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP16</td>
<td>71.6%</td>
<td>68.2%</td>
<td>73.8%</td>
</tr>
<tr>
<td>W8A8</td>
<td>32.3%</td>
<td>64.2%</td>
<td>26.9%</td>
</tr>
<tr>
<td>ZeroQuant</td>
<td>31.7%</td>
<td>67.4%</td>
<td>26.7%</td>
</tr>
<tr>
<td>LLM.int8()</td>
<td>71.4%</td>
<td>68.0%</td>
<td>73.8%</td>
</tr>
<tr>
<td>Outlier Suppression</td>
<td>31.7%</td>
<td>54.1%</td>
<td>63.5%</td>
</tr>
<tr>
<td>SmoothQuant-O1</td>
<td><strong>71.2%</strong></td>
<td>68.3%</td>
<td><strong>73.7%</strong></td>
</tr>
<tr>
<td>SmoothQuant-O2</td>
<td>71.1%</td>
<td><strong>68.4%</strong></td>
<td>72.5%</td>
</tr>
<tr>
<td>SmoothQuant-O3</td>
<td>71.1%</td>
<td>67.4%</td>
<td>72.8%</td>
</tr>
</tbody>
</table>
Lossless W8A8 quantization for LLaMA models

Lower perplexity is better

<table>
<thead>
<tr>
<th>Wiki PPL↓</th>
<th>7B</th>
<th>13B</th>
<th>30B</th>
<th>65B</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP16</td>
<td>11.51</td>
<td>10.05</td>
<td>7.53</td>
<td>6.17</td>
</tr>
<tr>
<td>W8A8 SmoothQuant</td>
<td>11.56</td>
<td>10.08</td>
<td>7.56</td>
<td>6.20</td>
</tr>
</tbody>
</table>
Memory/Latency Savings

Only compares with LLM.int8() because it is the only method that maintains accuracy.
Hardware Efficiency

Similar or faster latency with half # GPUs
Overall Results

- SmoothQuant is faster than FP16 baseline under all settings
- LLM.int8() is usually slower than SmoothQuant
- Additionally: SmoothQuant can serve a >500B model within a single node (8×A100 80GB GPUs) at a negligible accuracy loss
My Thoughts

● Strengths
  ○ Novel suggestion of migrating quantization difficulty to weights
  ○ Surpasses SOTA PTQ methods in high accuracy and low latency
  ○ High quality of evaluation - many families of LLMs including LLaMA
  ○ Easy to use (no training required)

● Weaknesses
  ○ Selecting the \( \alpha \) hyperparameter is a little difficult to get right
  ○ Getting activation statistics requires some work

● Design Choices
  ○ Very thoughtful and simple use of matmul’s linearity
  ○ Efficiency relies on access to hardware accelerators

● Future Directions
  ○ Getting down to 4 bits for weights and/or activations
  ○ Could FP4 be viable?
Discussion

- Can SmoothQuant scale for even larger LLMs (e.g. 100 trillion parameters)?

- Are there any specific types of applications where SmoothQuant wouldn’t be the best choice?

- What prevents quantizing to reduced bit-widths of size 4, 2, or even 1?
Q&A