FlexGen
High-throughput Generative Inference of Large Language Models with a Single GPU

Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Daniel Y. Fu, Zhiqiang Xie, Beidi Chen, Clark Barrett, Joseph E. Gonzalez, Percy Liang, Christopher Ré, Ion Stoica, Ce Zhang

Presented by Steven Gao
Outline

- Challenges in Inference
- Motivation
- Block Schedule
- Cost Model
- Compression
- Evaluation
From Training to Inference

- Latency
  - Small batch sizes
  - Memory bound
  - Maximize memory bandwidth utilization when reading weights for feed-forward layers

- Decoding throughput
  - Overlapping compute with reading model weights
    - We can increase throughput if latency remains the same per layer
    - Maximize batch size
  - Autoregressive decoding
    - Dependency on previous token (doesn’t exist in training)
    - Need to maintain KV cache longer
    - Higher memory requirements
Latency-Optimal Schedule

- Previous works go row by row in the compute graph
Latency-Optimal Schedule

- Previous works go row by row in the compute graph
  - Low latency
Motivation

- Make large models more accessible
Motivation

Latency oriented tasks (e.g. chatbots) → Throughput oriented tasks (e.g. offline document processing)
Throughput-Optimal Schedule

- Previous works go row by row in the compute graph
  - Low latency
  - High IO

![Diagram showing Throughput-Optimal Schedule](image)

<table>
<thead>
<tr>
<th>Parameters (26GB, 65%)</th>
<th>KV Cache (&gt;30%)</th>
<th>Others</th>
</tr>
</thead>
<tbody>
<tr>
<td>NVIDIA A100 40GB</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Throughput-Optimal Schedule

- Reuse layer weights
- Offload activations and KV cache
Overlapping Memory Access

Algorithm 1 Block Schedule with Overlapping

```
for i = 1 to generation.length do
    for j = 1 to num.layers do
        // Compute a block with multiple GPU batches
        for k = 1 to num.GPU_batches do
            // Load the weight of the next layer
            load.weight(i, j + 1, k)
            // Store the cache and activation of the prev batch
            store.activation(i, j, k - 1)
            store.cache(i, j, k - 1)
            // Load the cache and activation of the next batch
            load.cache(i, j, k + 1)
            load.activation(i, j, k + 1)
            // Compute this batch
            compute(i, j, k)
            // Synchronize all devices
            synchronize()
        end for
    end for
end for
```
Overlapping Memory Access

Algorithm 1 Block Schedule with Overlapping

```plaintext
for i = 1 to generation_length do
    for j = 1 to num_layers do
        // Compute a block with multiple GPU batches
        for k = 1 to num_GPU_batches do
            // Load the weight of the next layer
            load_weight(i, j + 1, k)
            // Store the cache and activation of the prev batch
            store.activation(i, j, k - 1)
            store.cache(i, j, k - 1)
            // Load the cache and activation of the next batch
            load.cache(i, j, k + 1)
            load.activation(i, j, k + 1)
            // Compute this batch
            compute(i, j, k)
        // Synchronize all devices
        synchronize()
    end for
end for
```
Tensor Placement

wg, wc, wd: percentage of **weights** placed on GPU, CPU, and disk

hg, hc, hd: **activations**

cg, cc, cd: **KV cache**
Tensor Placement

- Weights: layer granularity
- KV cache, activations: tensor granularity


Cost Model

\[ T = T_{pre} \cdot l + T_{gen} \cdot (n - 1) \cdot l \]

\( T \): latency

\( T_{pre} \): prefilling latency for one layer (multiple blocks)

\( l \): number of layers

\( T_{gen} \): decoding latency for one layer (multiple blocks)

\( n \): generation length
Cost Model

\[ T_{pre} = \max(ctog^p, gtoc^p, dtoc^p, ctod^p, comp^p) \]

**ctog**: latency of transfer from CPU -> GPU, etc.
Cost Model

\[ T_{pre} = \max (ctog^p, gtoc^p, dtoc^p, ctod^p, comp^p) \]

\( ctog \): latency of transfer from CPU -> GPU, etc.

\[ T_{gen} = \max (ctog^g, gtoc^g, dtoc^g, ctod^g, comp^g) \]
Cost Model Assumptions

- Perfect overlapping
- All latencies are estimated by summing up IO events
- Computation term is estimated by summing up matrix multiplication
Cost Model

Bls: effective batch size (batch size * # GPU blocks)

\[
ctog^p = \frac{weights_{ctog^p} + act_{ctog^p}}{ctog_{bdw}}
\]

\[
= \frac{1}{ctog_{bdw}} ((wc + wd)(8h_1^2 + 4h_1 \cdot h_2) + 2(hc + hd)s \cdot h_1 \cdot bls)
\]
Policy Search

\[
\begin{align*}
\min_{p} & \quad T/\text{bls} \\
\text{s.t.} & \quad \text{gpu peak memory} \leq \text{gpu mem capacity} \\
& \quad \text{cpu peak memory} \leq \text{cpu mem capacity} \\
& \quad \text{disk peak memory} \leq \text{disk mem capacity} \\
& \quad wg + wc + wd = 1 \\
& \quad cg + cc + cd = 1 \\
& \quad hg + hc + hd = 1
\end{align*}
\]
CPU Compute

Size of KV cache: $b \times s \times h_1 \times 4$

Size of activation: $b \times h_1 \times 4$

IO reduction by $s$ times if we compute attention scores on CPU
CPU Compute

\[ f_{\text{Softmax}} \left( \frac{\mathbf{x}_Q^i \mathbf{x}_K^i T}{\sqrt{h}} \right) \cdot \mathbf{x}_V^i \]
Multiple GPUs

- Pipeline parallelism
  - Low communication costs
- Add micro-batches in inner loop

**Algorithm 1** Block Schedule with Overlapping

```plaintext
for i = 1 to generation_length do
  for j = 1 to num_layers do
    // Compute a block with multiple GPU batches
    for k = 1 to num_GPU_batches do
      // Load the weight of the next layer
      load_weight(i, j + 1, k)
      // Store the cache and activation of the prev batch
      store.activation(i, j, k - 1)
      store.cache(i, j, k - 1)
      // Load the cache and activation of the next batch
      load_cache(i, j, k + 1)
      load.activation(i, j, k + 1)
      // Compute this batch
      compute(i, j, k)
      // Synchronize all devices
      synchronize()
    end for
  end for
end for
```
Quantization

- Group quantization to 4 bits for weights and KV cache

\[ x_{\text{quant}} = \text{round} \left( \frac{x - \text{min}}{\text{max} - \text{min}} \times (2^b - 1) \right) \]

- Group weights along output dim and KV cache along hidden dim
- CPU compute is turned off due to overhead
- Further reduces IO cost
Sparse Attention

- Compute scores given Q
- Only load top 10% K with highest scores
- Select corresponding Vs
Experimental Setup

- T4 GCP instances
- 2 GB/s read and 1 GB/s write for SSD
- OPT models: 6.7B, 30B, 175B
- Synthetic dataset with padded prompts
- Prompt length: 512, 1024
- Generation length: 32

<table>
<thead>
<tr>
<th>Device</th>
<th>Model</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU</td>
<td>NVIDIA T4</td>
<td>16 GB</td>
</tr>
<tr>
<td>CPU</td>
<td>Intel Xeon @ 2.00GHz</td>
<td>208 GB</td>
</tr>
<tr>
<td>Disk</td>
<td>Cloud default SSD (NVMe)</td>
<td>1.5 TB</td>
</tr>
</tbody>
</table>
Baseline

- DeepSpeed Inference
- HuggingFace Accelerate
- Petals
  - Collaborative inference over network
  - 10 ms latency and 1Gbps bandwidth
## Throughput Results

- Petals uses 1, 4, and 24 GPUs respectively for the three model sizes

<table>
<thead>
<tr>
<th>Seq. length</th>
<th>512</th>
<th>1024</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model size</td>
<td>6.7B</td>
<td>175B</td>
</tr>
<tr>
<td>Accelerate</td>
<td>25.12</td>
<td>0.01</td>
</tr>
<tr>
<td>DeepSpeed</td>
<td>9.28</td>
<td>0.01</td>
</tr>
<tr>
<td>Petals</td>
<td>8.25</td>
<td>0.08</td>
</tr>
<tr>
<td>FlexGen</td>
<td>25.26</td>
<td>0.69</td>
</tr>
<tr>
<td>FlexGen (c)</td>
<td>29.12</td>
<td>1.12</td>
</tr>
</tbody>
</table>

For comparison, the table above shows the throughput results in Gbps for different models and sequence lengths. The Petals model uses 1, 4, and 24 GPUs respectively for the three model sizes. The throughput values are shown in Gbps for each model and sequence length combination. For example, Petals achieves a throughput of 8.25 Gbps for a sequence length of 512 when using 1 GPU. The results are displayed in a table format with columns for sequence length (Seq. length), model size, and throughput values for different models (Accelerate, DeepSpeed, Petals, FlexGen, FlexGen (c)).
Multi-GPU Scaling

- 4 GPUs, per GPU throughput reported
- Superlinear scaling for decoding throughput with pipeline parallelism
- Generation (prefilling + decode) doesn’t scale superlinearly with $n = 32$

<table>
<thead>
<tr>
<th>Model size</th>
<th>Generation Throughput</th>
<th>Decoding Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6.7B</td>
<td>30B</td>
</tr>
<tr>
<td>FlexGen (1)</td>
<td>25.26</td>
<td>7.32</td>
</tr>
<tr>
<td>FlexGen (4)</td>
<td>201.12</td>
<td>23.61</td>
</tr>
<tr>
<td>DeepSpeed (4)</td>
<td>50.00</td>
<td>6.40</td>
</tr>
</tbody>
</table>
Latency-Throughput Tradeoff

- **FlexGen (c)**
- **FlexGen**
- **DeepSpeed**
- **Accelerate**

**OPT-175B**

**OPT-30B**

Generation throughput (token/s)

Latency (s)
## Quantization Results

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Lambda (acc)</th>
<th></th>
<th></th>
<th></th>
<th>WikiText (ppl)</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>FP16</td>
<td>4-bit</td>
<td>4-bit-S</td>
<td></td>
<td>FP16</td>
<td>4-bit</td>
<td>4-bit-S</td>
<td></td>
</tr>
<tr>
<td>OPT-30B</td>
<td>0.725</td>
<td>0.724</td>
<td>0.718</td>
<td></td>
<td>12.72</td>
<td>12.90</td>
<td>12.90</td>
<td></td>
</tr>
<tr>
<td>OPT-175B</td>
<td>0.758</td>
<td>0.756</td>
<td>0.756</td>
<td></td>
<td>10.82</td>
<td>10.94</td>
<td>10.94</td>
<td></td>
</tr>
</tbody>
</table>

*Table 5. The accuracy (higher is better) and perplexity (lower is better) with approximate methods.*
Comparisons with Collaborative Inference

![Graph showing comparisons between FlexGen 1xT4 and Petals 4xT4 with different latency and throughput values.]

- FlexGen 1xT4
- Petals 4xT4 10ms 0.1Gbps
- Petals 4xT4 100ms 0.1Gbps

- Full generation latency (s) vs. Output sequence length
- Throughput per GPU (token/s) vs. Output sequence length
# Ablation Study

Table 23. Ablation study of proposed techniques. The numbers are generation throughput on 1 T4 GPU with prompt length 512 and generating length 32. The gray tuple denotes a policy (GPU batch size $\times$ #GPU-batch, $wg$, $wc$, $cg$, $cc$, $hg$, $hc$).

<table>
<thead>
<tr>
<th>Model size</th>
<th>30B</th>
<th>175B</th>
</tr>
</thead>
<tbody>
<tr>
<td>All optimizations</td>
<td><strong>7.32</strong> $(48\times3, 20, 80, 0, 100, 0, 100)$</td>
<td><strong>0.69</strong> $(32\times8, 0, 50, 0, 0, 0, 100)$</td>
</tr>
<tr>
<td>No policy search</td>
<td><strong>7.26</strong> $(48\times3, 0, 100, 0, 100, 0, 100)$</td>
<td><strong>0.27</strong> $(32\times1, 0, 50, 0, 0, 0, 100)$</td>
</tr>
<tr>
<td>No overlapping</td>
<td>5.86 $(48\times3, 20, 80, 0, 100, 0, 100)$</td>
<td><strong>0.59</strong> $(32\times8, 0, 50, 0, 0, 0, 100)$</td>
</tr>
<tr>
<td>No CPU compute</td>
<td>4.03 $(48\times3, 20, 80, 0, 100, 0, 100)$</td>
<td><strong>0.62</strong> $(32\times8, 0, 50, 0, 0, 0, 100)$</td>
</tr>
<tr>
<td>No disk</td>
<td>7.32 $(48\times3, 20, 80, 0, 100, 0, 100)$</td>
<td>OOM</td>
</tr>
<tr>
<td>w/ DeepSpeed policy</td>
<td><strong>1.57</strong> $(8\times1, 0, 100, 100, 0, 100, 0)$</td>
<td><strong>0.01</strong> $(2\times1, 0, 0, 100, 0, 100, 0)$</td>
</tr>
</tbody>
</table>
### Runtime Breakdown

*Table 8.* Execution time breakdown (seconds) for OPT-175B. The prompt length is 512. (R) denotes read and (W) denotes write.

<table>
<thead>
<tr>
<th>Stage</th>
<th>Total</th>
<th>Compute</th>
<th>Weight (R)</th>
<th>Cache (R)</th>
<th>Cache (W)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prefill</td>
<td>2711</td>
<td>2220</td>
<td>768</td>
<td>0</td>
<td>261</td>
</tr>
<tr>
<td>Decoding</td>
<td>11315</td>
<td>1498</td>
<td>3047</td>
<td>7046</td>
<td>124</td>
</tr>
</tbody>
</table>
## Compression Throughput

(batch x bls, wg, wc, cg, cc, hg, hc)

<table>
<thead>
<tr>
<th>Seq. length</th>
<th>512 + 32</th>
<th>175B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model size</td>
<td>6.7B</td>
<td>30B</td>
</tr>
<tr>
<td>Accelerate</td>
<td>183.177 (16×1, 100, 0, 100, 0, 100, 0)</td>
<td>2.077 (13×1, 0, 100, 100, 0, 100, 0)</td>
</tr>
<tr>
<td>DeepSpeed</td>
<td>38.027 (32×1, 0, 100, 100, 0, 100, 0)</td>
<td>3.889 (12×1, 0, 100, 100, 0, 100, 0)</td>
</tr>
<tr>
<td>FlexGen</td>
<td>233.756 (28×1, 100, 0, 100, 0, 100, 0)</td>
<td>5.726 (4×15, 25, 75, 40, 60, 100, 0)</td>
</tr>
<tr>
<td>FlexGen (c)</td>
<td><strong>120.178</strong> (144×1, 100, 0, 100, 0, 100, 0)</td>
<td><strong>16.547</strong> (96×2, 25, 75, 0, 100, 100, 0)</td>
</tr>
</tbody>
</table>
Takeaways

● Efficient offload strategy
  ○ Formulate cost model and search space for offloading policy
● 4-bit quantization on KV cache and weights
● Scaled to multi-GPU with pipeline parallelism