Scaling Distributed Deep Learning Workloads beyond the Memory Capacity with KARMA

Yi Mao
Outline

● Motivations
● Existing Solutions and their limitations
● KARMA
● Evaluations
Motivations

Today's models and datasets are too large to fit in a single GPU memory.

- ResNet-1001 on ImageNet, only 2 samples one time.
- Megatron-LM, 8.3B parameters, needs 8 V100
Why DP/MP is not enough?

- Data parallelism:
  memory pressure from model parameters and activations.
  high resolution medical or satellite images, can be large than 2GiB.

- model parallelism
  Complicated to modify the code for the model
OOC (Out Of Core) —— vDNN++

limitations :

- GPU inefficient
- not support multiple GPUS
Recomputing

limitations:

- at least require $O(\log N)$
- not support multiple GPUs
## Limitations Summary

<table>
<thead>
<tr>
<th>Name</th>
<th>Approach</th>
<th>Min. Req. Memory</th>
<th>Universal</th>
<th>Multi-node</th>
<th>Strong Scaling (MN)</th>
<th>Fault Tolerance (MN)</th>
</tr>
</thead>
<tbody>
<tr>
<td>vDNN++</td>
<td>OOC</td>
<td>None</td>
<td>x</td>
<td>x</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>ooc_cuDNN</td>
<td>OOC</td>
<td>None</td>
<td>x</td>
<td>x</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Gradient Checkpoint</td>
<td>RECOMP</td>
<td>$\mathcal{O}(\sqrt{N})$</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
</tr>
<tr>
<td>SuperNeurons</td>
<td>OOC &amp; RECOMP</td>
<td>$\mathcal{O}(\sqrt{N})$</td>
<td>x</td>
<td>x</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>PooCH</td>
<td>OOC &amp; RECOMP</td>
<td>$\mathcal{O}(\sqrt{N})$</td>
<td>x</td>
<td>x</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Graph Partitioning</td>
<td>Implicit MP</td>
<td>None</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>✓</td>
</tr>
<tr>
<td>FlexFlow</td>
<td>Explicit MP</td>
<td>$\mathcal{O}(\sqrt{P})$</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>KARMA (This work)</td>
<td>OOC &amp; RECOMP</td>
<td>None</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
How to quantify each layer and transform to an optimization problem?
Swap In/Out, Processing.

(a) Forward:
- Swap-In/Out: D → HD → HD → HD → HD → HD → HD → HD → H
- Processing: 1 2 3 4 5 6

(b) Backward:
- Swap-In/Out: D → HD → HD → HD → HD → HD → HD → HD → H
- Processing: 1 2 3 4 5 6

(c) Swap-In/Out:
- Swap-In/Out: D → HD → H
- Processing: 1 3 3 1

Start swap-in when memory available.
No swap-out if memory available.
No swap needed: layers recomputed.
Objective function to optimize

\[ \Phi_j \approx \begin{cases} \frac{B_{avail}^j}{B_{req}^j} & \text{for } B_{avail}^j > B_{req}^j \\ 1 & \text{for } B_{avail}^j \leq B_{req}^j \end{cases} \]

\[ B_{avail} = \begin{cases} B_{avail}^{j-1} - \sum_{i=1}^{j-1} (B_{swapped-in}^i - B_{processed}^i) & \text{for } B_{avail}^{j-1} \leq B_{avail}^j - B_{processed}^j \\ 0 & \text{for } B_{avail}^{j-1} \leq B_{avail}^j - B_{processed}^j \end{cases} \]

\[ B_{swapped-in}^j = \min \{ T_{swap-in} \cdot T_{proc}(b), B_{avail}^{j-1} \} \]

\[ \Phi_j \approx \max \left\{ \frac{B_{avail}^j}{\sum_b (B_{processed}^j(b) + T_{swap-in} \cdot T_{proc}(b))}, 1 \right\} \]
Quantify each layer ——To estimate block proc time

- Convolution Layer: For the kernel size $K \cdot K$ and $C$ channels, a convolution layer needs $K \cdot K \cdot C$ multiply and add operations. The number of operations for direct convolution of one sample is:

  $$|Y_i| \cdot K \cdot K \cdot C = W_{i+1} \cdot H_{i+1} \cdot C_{i+1} \cdot K \cdot K \cdot C_i$$

- ReLu Layer: The output of a ReLu layer is $y_i = \max(0,x_i)$. This requires $|Y_i|$ comparison operations.
- Pooling Layer: The total number of operations is:

  $$|Y_i| \cdot K \cdot K \cdot C \cdot c = W_{i+1} \cdot H_{i+1} \cdot C_{i+1} \cdot K \cdot K \cdot C_i$$
Memory quantify

Conduct a set of experiments to gather information on the memory required per layer, and also fused layers commonly occurring together, using PyTorch's memory stats() API that monitors the memory occupied by tensors.
Problem Formulation 1

For a layer set \( L := \{L_1, \ldots, L_l\} \) split to a set of blocks \( B := \{B_1, \ldots, B_b\} \), finding the execution schedule results in set of stages \( S := \{S_1, \ldots, S_s\} \) having occupancies of \( O(S_i) \).

--- Optimization Problem 1 ---

**Input**
- Layers \( L := L_{1:t} \)
- Blocks \( B := B_{1:b} \)

**Output**

\[
\text{Maximize} \quad \sum_{t=1}^{T} O_{\text{Occupancy}_t} \quad (9)
\]

**Subject to**

\[
\sum_{i=1}^{l} \delta_{ij} L_i = 1 \quad , \forall j \in \{1, \ldots, b\} \quad (9.1)
\]

\[
\sum_{i} \delta_{ij} = 1 \quad , \forall i \forall j L_i \in B_j \quad (9.2)
\]

\[
C_{ij} = 0 \quad , \forall \delta_{jm} = 1 \forall \delta_{im} = 1, m < n \quad (9.3)
\]

\[
\text{Mem}(t) \leq \text{Capacity} \quad , \forall q \in \{1, \ldots, s\} \quad (9.4)
\]

**Where**

- \( \sum \text{Occupancy} \) Occupancy over time \( T \) (Eq. 9)
- \( \text{Mem}(t) \) Returns device memory required at step \( t \)
- \( \text{Capacity} \) Is device memory capacity avail. for tensors
- \( \delta_{ij} \in \{0, 1\} \) \( \delta_{ij} = 1 \) if layer \( L_i \) is in block \( B_j \)
- \( C_{ij} \in \{0, 1\} \) \( C_{ij} = 1 \) if layer \( L_i \) feeds Layer \( L_j \)
Problem Formulation 2

Next, the stages $S$ are refined to new stages $S'$ such that a block is recomputed if the recompute time until the next checkpoint is less than the swapin time of the previous block in the path to the checkpoint.

--- Optimization Problem 2 ---

**Input**
- Blocks $B_{1 \rightarrow b}$

**Output**
- New Blocks $B'_{1 \rightarrow b}$

**Minimize**

$$\sum_{i=1}^{b} Comp(B_i)$$

**Subject to**

$$\sum_{d \in D} Comp(B_{d}^k) < \sum_{k} Swap(B_{k}^{d+1}), \; k \in \{1, \ldots, b\}$$

**Where**

- $\Delta$ Set of Blocks until next checkpoint
- $Comp(B_m)$ Returns the compute time for block $B_m$
- $Swap(B_m)$ Returns the swap time for block $B_m$
Solve and Output

Integer Linear Programming (ILP) to solve the problem

Write an algo to output the computation sequence:

\[ F_1 \rightarrow F_2 \parallel S_1 \text{ out} \rightarrow F_3 \rightarrow F_4 \parallel S_3 \text{ out} \rightarrow F_5 \rightarrow F_6 \rightarrow B_6 \parallel S_3 \text{ in} \rightarrow B_5 \rightarrow F_4 \rightarrow B_4 \parallel S_1 \text{ in} \rightarrow B_3 \rightarrow F_2 \rightarrow B_2 \rightarrow B_1 \]
Distributed learning — Challenge

Weight update requires a separate step. This separate weight update step follows after computing the gradients and exchanging them among nodes (or GPUs). However, in out-of-core methods, the layers (including their weights) do not entirely reside on the GPU after the end of the backward phase. Since the layers would not entirely fit on the device memory, a trivial workaround is to move the layers entirely to the CPU after the backward phase to apply the weight update there, but this yields an unacceptable performance penalty.
Evaluation (Speed)

Fig. 5: Performance using a V100 SMX2 GPU (16GB). For all figures, only the first reported mini-batch size (x-axis) fits in memory.
Evaluation (Multi-GPUs)


<table>
<thead>
<tr>
<th>H</th>
<th>A</th>
<th>L</th>
<th>P</th>
<th>Megatron-LM († Num. GPUs)</th>
<th>DP KARMA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>MP†</td>
<td>MP+DP†</td>
</tr>
<tr>
<td>1152</td>
<td>12</td>
<td>18</td>
<td>0.7B</td>
<td>1</td>
<td>64</td>
</tr>
<tr>
<td>1536</td>
<td>16</td>
<td>40</td>
<td>1.2B</td>
<td>2</td>
<td>128</td>
</tr>
<tr>
<td>1920</td>
<td>20</td>
<td>54</td>
<td>2.5B</td>
<td>4</td>
<td>256</td>
</tr>
<tr>
<td>2304</td>
<td>24</td>
<td>64</td>
<td>4.2B</td>
<td>8</td>
<td>512</td>
</tr>
<tr>
<td>3072</td>
<td>32</td>
<td>72</td>
<td>8.3B</td>
<td>16</td>
<td>1024</td>
</tr>
</tbody>
</table>
Thank you!