PipeDream & LayerPipe

Nick Contini
29 October 2021
Outline

- Pipeline Parallelism Review
- PipeDream
  - Partitioning
  - Scheduling
  - Weight Stashing
- LayerPipe
  - Splitting the $\delta$
  - Scheduling Part II
Review

- What are the disadvantages of Data and Model Parallelism?
• Pipeline Parallelism
  - Exploits low gradient rate of change
  - Distributes layers across multiple processors
  - Make better use of hardware than Model Parallelism
  - Avoids excessive communication
  - Overlaps communication and computation
  - Do we see any issues with Gpipe’s implementation?
Unsolved Challenges

- How do we divide layers into pipeline stages less naively and schedule computations to keep the pipeline busy?
- How do we calculate gradients effectively and efficiently?
Outline

- Pipeline Parallelism Review
- PipeDream
  - Partitioning
  - Scheduling
  - Weight Stashing
- LayerPipe
  - Splitting the $\delta$
  - Scheduling Part II
PipeDream

- Developed By Microsoft
- Addresses many of the issues ignored by Gpipe
Partitioning

- Naively dividing the layers is suboptimal
  - Layers have non-uniform memory and computation requirements
  - Bandwidth varies between pairs of processes

- Solution: use dynamic programming to optimally partition the layers
  - Profile computation of each layer
  - Measure bandwidth of each level of the network
  - Replicate layers where necessary
Scheduling

- Pipeline is bi-directional
- Backward direction must progress in order to issue new entries in the forward direction
- PipeDream opts for a static schedule using the 1F1B-RR strategy
Weight Stashing

- Naive implementations do not use same weights on forward and backward pass

- This can make our training ineffective!

- Weight stashing
  - Store weights for the n active minibatches
  - Use delayed weights for corresponding backward pass within a stage

- Vertical sync is a similar method, but focuses on using the same set of weights across stages
Results

(a) Cluster-A.

(b) Cluster-B.

(a) GNMT-16

(b) VGG-16
Outline

• Pipeline Parallelism Review

• PipeDream
  – Partitioning
  – Scheduling
  – Weight Stashing

• LayerPipe
  – Splitting the $\delta$
  – Scheduling Part II
LayerPipe

- Framework that improves upon PipeDream by splitting up the backward pass.
- Developed by University of Minnesota
Splitting the $\delta$

- The backward pass is almost always longer to execute than the forward pass
- Communication overhead for each variable is different per layer
- Goal: split up the backward pass so communication overhead is minimized while allowing more parallelism
Splitting the $\delta$
Scheduling

- Instead of using dynamic programming, LayerPipe uses a greedy approach
- Profile computations
- Label computations as “movable” if they are within memory and communication restraints
- Move as much movable computation to layer $l + 1$ as possible.
- If computation is too long, split into $\delta'$ and $\delta''$ and move $\delta''$ to $l + 1$

Prioritize “critical loop” operations

Algorithm 1 Partitioning algorithm for balanced pipeline generation.

Input: DFG of DNN, #processors ($N_p$), processors

Output: Critical loops $C_l$, processor allocation $P_{a}$

1: //Step 1: Find all $C_{loops}$ and DNN layers $L$ in the DFG
2: $C_l, L =$ find_critical_loops(DFG)

3: //Step 2 starts here

4: //Profile: For each DNN layer $l$ in DFG find layer compute time $t_{c,l}$, fixed computation time $t_{fix,l}$, and flexible computation time $t_{flex,l}$. Store in $T_c, T_{fix,l}$, and $T_{flex}$
5: $T_c, T_{fix}, T_{flex} =$ profile(DFG)

6: $T_{flex} =$ for $i$ in $T_{c}[i]$ do sum($T_{i}$)

7: //Find maximum processor time $T_{p}$
8: $T_p = \frac{L}{N_p}$

9: //Step 3 starts here

10: //Initialize flag, processor index $p_{idx}$ to 0 and processor idle time $T_{idle}$ to $T_p$
11: flag = 0

12: while flag = 0 do

13: $p_{idx} = 0; T_{idle} = T_p$

14: for each $l$ in reversed($L$) do

15: if $T_{flex}[l] < T_{idle}$ then

16: allocate $T_{flex}[l]$ to processors[$p_{idx}$] and update $P_{a}$

17: $T_{idle} = T_{idle} - T_{flex}[l]$

18: else

19: // Partition $T_{flex}[l]$ ($\delta'$ and $\delta''$) with operational parallelism (OP)

20: $\delta', \delta'' = OP(T_{idle}, T_{flex}[l])$

21: allocate $\delta''$ to processors[$p_{idx}$] and update $P_{a}$

22: $p_{idx} = p_{idx} + 1$

23: allocate $\delta'$ to processors[$p_{idx}$] and update $P_{a}$

24: $T_{idle} = T_p - \delta'$

25: end if

26: if $T_{fix}[l] > T_{idle}$ then

27: $p_{idx} = p_{idx} + 1$

28: end if

29: allocate $T_{fix}[l]$ to processors[$p_{idx}$] and update $P_{a}$

30: $T_{idle} = T_{idle} - T_{fix}[l]$

31: end for

32: if $p_{idx} > N_p$ then

33: //Relax the max processor time constraint and flags stays 0

34: $T_p = \alpha \times T_p$

35: else

36: flag = 1

37: end if

38: end while

39: return $L_c, P_{a}$
Scheduling

- Note that none of the backward pass computations depend on $G$
- $\delta'$ and $\delta''$ can also be computed in parallel
Results

**TABLE III**

Comparison of computation times in cycles of pipeline parallelism algorithms for the sample four-layer network.

<table>
<thead>
<tr>
<th>Processors</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>PipeDream</td>
<td>$4.35 \times 10^7$</td>
<td>$3.07 \times 10^7$</td>
<td>$2.23 \times 10^7$</td>
</tr>
<tr>
<td>LayerPipe</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Assigned computation</td>
<td>$3.22 \times 10^6$</td>
<td>$2.09 \times 10^7$</td>
<td>$2.23 \times 10^7$</td>
</tr>
<tr>
<td>Borrowed computation</td>
<td>$0.00 \times 10^9$</td>
<td>$1.13 \times 10^7$</td>
<td>$9.86 \times 10^6$</td>
</tr>
<tr>
<td>Total</td>
<td>$3.22 \times 10^7$</td>
<td>$3.22 \times 10^7$</td>
<td>$3.22 \times 10^7$</td>
</tr>
</tbody>
</table>
Fig. 8. Performance comparison between LayerPipe and PipeDream for different number of processors on the convolutional layers of a) VGG16 b) ResNet50. The results were averaged over various batch sizes, systolic array sizes, and normalized to a single processor’s performance. The individual distributions for VGG16 is shown for c) array size and d) batch size.
Questions?