BigDL: A Distributed Deep Learning Framework for Big Data

ACM Symposium on Cloud Computing 2019 [Acceptance Rate: 24%]
Jason Jinquan Dai, Yiheng Wang, Xin Qiu, Ding Ding, Yao Zhang, Yanzhang Wang, Xianyan Jia, Cherry Li Zhang, Yan Wan, Zhichao Li, Jiao Wang, Sheng-sheng Huang, Zhongyuan Wu, Yang Wang, Yuhao Yang, Bowen She, Dongjie Shi, Qi Lu, Kai Huang, Guoqiong Song. [Intel Corporation]

Presenter: Ezgi Korkmaz
Outline

- Deep learning frameworks
- BigDL applications
- Motivation for end-to-end framework
- Drawbacks of prior approaches
- BigDL framework
- Experimental Setup and Results of BigDL framework
- Critique of the paper
Deep Learning Frameworks

- Big demand from organizations to apply deep learning to big data
- Deep learning frameworks:
  - **Torch** [2002 Collobert et al.] [C, Lua]
  - **Caffe** [2014 Berkeley BAIR] [C++]
  - **TensorFlow** [2015 Google Brain] [C++, Python, CUDA]
  - **Apache MXNet** [2015 Apache Software Foundation] [C++]
  - **Chainer** [2015 Preferred Networks] [Python]
  - **Keras** [2016 Francois Chollet] [Python]
  - **PyTorch** [2016 Facebook] [Python, C++, CUDA]
- Apache Spark is an open-source distributed general-purpose cluster-computing framework.
  - Provides interface for programming clusters with data parallelism.
BigDL

- A library on top of Apache Spark
- Provides integrated data-analytics within a unified data analysis pipeline
- Allows users to write their own deep learning applications
- Running directly on big data clusters
- Supports similar API to Torch and Keras
- Supports both large scale distributed training and inference
- Able to run across hundreds or thousands servers efficiently by uses underlying Spark framework
BigDL

- Developed as an open source project
- Used by
  - Mastercard
  - WorldBank
  - Cray
  - Talroo
  - UCSF
  - JD
  - UnionPay
  - GigaSpaces
- Wide range of applications: transfer learning based image classification, object detection, feature extraction, sequence-to-sequence prediction, neural collaborative filtering for recommendation etc.
Motivation

▶ Normally in research established datasets and benchmarks

What if we had dynamic data?

▶ We need to care about compatibility and efficiency in another level
  ▶ End-to-end integrated data analytics and deep learning frameworks
▶ Real world data is dynamic, messy and implicitly labeled
▶ Requires more complex data processing
▶ Moreover, it is not single shot i.e. ETL (extract, transform, load)
▶ It is iterative and recurrent (back-and-forth development and debugging)
Prior Approach Drawbacks

- It is highly inefficient to run on big data and deep learning systems separately
- Connector approach:
  - TFX
  - CaffeOnSpark
  - TensorFlowOnSpark
  - SageMaker
    - provides proper interface to connect data processing and deep learning frameworks
- Results in very large overheads in practice
  - inter-process communication
  - data serialization
  - impedance matching (crossing boundaries between heterogenous components)
  - persistency etc.
Prior Approach

- If a Spark worker fails, Spark system relaunch the worker
  - Highly incompatible with TensorFlow execution model
  - Causing the entire workflow to be blocked indefinitely

- BigDL
  - Directly implements the distributed deep learning in the big data
  - End-to-end single framework
  - Eliminates the impedance mismatch
  - Efficient
Data-parallel training in BigDL

**Figure**: The “model forward-backward” spark job computing the local gradients for each model replica in parallel.
Parameter Synchronization in BigDL

Figure: Parameter synchronization in BigDL. Each local gradient is evenly divided in N partitions; then each task n in the “parameter synchronization” job aggregates these local gradients and updates the weights for the n\textsuperscript{th} partition.
Evaluation

- In their experimental setup authors use
  - Neural Collaborative Filtering (NCF)
    - NCF on MovieLens 20Million dataset
    - 20 million ratings
    - 465000 tags
    - 27000 movies
    - 138000 users
  - Convolutional Neural Networks (CNNs)
    - Inception-v1 on ImageNet
Experiments for NCF

- NCF using BigDL trained on dual-socket Intel Skylake [29.8 min]
  - 56 cores and 384 GB memory
- NCF using PyTorch [Reported by MLPerf]
  - Single Nvidia P100 GPU
Experiments on ImageNet

Figure: Left: Overhead of parameter synchronization measured as a fraction of the average model computation time. Right: Throughput of ImageNet Inception-v1 training in BigDL.

- ImageNet Inception-v1 using BigDL
  - Each node dual-socket Intel Broadwell 2.1GHz
  - Synchronization overhead is small (less than 7%)
  - Scales linearly up to 96 nodes.
Experiments on ImageNet Task Scheduling

Figure: Overheads of task scheduling and dispatch for ImageNet Inception-v1 in BigDL.

- Needs to schedule very large number of tasks across cluster
  - Low for 100-200 tasks per iteration
  - Grows to over 10% close to 500 tasks per iteration
  - Using group scheduling introduced by Drizzle (low latency execution engine) can reduce this overhead
Experiments on Feature Extraction

Figure: Throughput of GPU clusters and Xeon clusters for the image features extraction pipeline benchmarked by JD.

- GPU cluster consists of 20 NVIDIA Tesla K40 cards
- Xeon Cluster consists of 1200 logical cores running 50 logical cores
Critique

- Some metric to actually quantify the discussions in the experiment section
  - Server utilization?
  - Cost for equipment?
  - Network traffic?
  - Power dissipation?

- As it stands hard to interpret the actual contribution
- Explicit explanation needs to be added on speed comparison with proper metrics
PyTorch Distributed: Experiences on Accelerating Data Parallel Training

Shen Li et al

VLDB Endowment 2020
Contributions

- Describes the design and implementation of a widely adopted industrial state-of-the-art distributed training solution
- Highlights real-world caveats
- Share performance tuning experiences collected from serving internal teams and open-source community users and summarized several directions for future improvements.
Design Principles

- API
  - Non-intrusive
  - Allow the implementation to intercept signals and trigger appropriate algorithms promptly

```python
import torch
import torch.nn as nn
import torch.nn.parallel as par
import torch.optim as optim

# initialize torch.distributed properly
# with init_process_group

# setup model and optimizer
net = nn.Linear(10, 10)
net = par.DistributedDataParallel(net)
opt = optim.SGD(net.parameters(), lr=0.01)

# run forward pass
inp = torch.randn(20, 10)
exp = torch.randn(20, 10)
out = net(inp)

# run backward pass
nn.MSELoss()(out, exp).backward()

# update parameters
opt.step()
```
Design Principles

● Gradient Reduction
  ○ Allreduce
    ■ Performs poorly on small tensors
    ■ No opportunity to overlap computation with communication (since everyone must join)
  ○ Gradient bucketing
    ■ Still Allreduce
    ■ Concat small tensors into bigger chunks
    ■ Do not reduce all parameters in single shot
    ■ Triggered by autograd hook
Design Principles

- Communication Collectives
  - MPI
  - NCCL
  - Gloo
  - Use ProcessGroups to manage parallelisms (communication, CUDA, etc)
Evaluation platform

- 4 GPU servers
- Mellanox MT27700 ConnectX-4 100GB/s NIC
- 8 NVIDIA Tesla V100 GPUs per node
Latency improvement

- Overlapping approach on ResNet and BERT on 32 GPUs
  - NCCL attains 38.0% and 35.2% speedup
  - GLOO attains 26.8% and 21.5% speedup

Figure 6: Per Iteration Latency Breakdown
Scalability (Weak Scaling!)

Figure 9: Scalability

(a) ResNet50 on NCCL  
(b) ResNet50 on Gloo  
(c) BERT on NCCL  
(d) BERT on Gloo
Scalability ("Async" update")

(a) ResNet50 on NCCL

(b) ResNet50 on Gloo

Figure 10: Skip Gradient Synchronization
Conclusion

- Communication backends
- Bucket size (transfer size)
- Resource allocation (out-of-core training)
Discussion

- What are the limitation of hook based communication trigger?
- Can this scheme impact progress? (e.g. stall for bucket 0)
Beyond Data and Model parallelism For Deep Neural Networks

Zhihao Jia, Matei Zaharia, Alex Aiken

MLSys’19
- Background and Research Question
- Hybrid Parallelism
- Solution
- Optimization
- Related work and Conclusion
Background

- Data parallelism
- Model parallelism
Data parallelism

- Batches as finest granularity
- Synchronize every certain intervals
- Care required in synchronization and parameter tuning
Model parallelism

- Operations in model as lowest granularity
- All operations execute asynchronously, subject to dependency
- Difficult placement
  - Computation power
  - Communication cost
  - Degree of overlapping
What if....

The operations themselves can be parallelized?
FlexFlow: SOAP

- **Sample**
- **Operator**
- **Attribute**
- **Parameter**
FlexFlow: SOAP

- **Sample**
  - E.g. sample based data parallelism

- **Operator**
  - E.g. Conv / gemm

- **Attribute**
  - E.g. Length / Width

- **Parameter**
  - E.g. Channels
Table 2. Parallelizable dimensions for different operators. The *sample* and *channel* dimension index different samples and neurons, respectively. For images, the *length* and the combination of *height* and *width* dimensions specify a position in an image.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Parallelizable Dimensions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(S)ample (A)tribute (P)ararameter</td>
</tr>
<tr>
<td>1D pooling</td>
<td>sample length, channel</td>
</tr>
<tr>
<td>1D convolution</td>
<td>sample length</td>
</tr>
<tr>
<td>2D convolution</td>
<td>sample height, width</td>
</tr>
<tr>
<td>Matrix multiplication</td>
<td>sample channel</td>
</tr>
</tbody>
</table>

Figure 2. Example parallelization configurations for 1D convolution. Dashed lines show partitioning the tensor.
How to compute a matrix multiplication?

![Diagram](image)

**Figure 3.** An example parallelization configuration for a matrix multiplication operator.


![Image](image)
How to compute a matrix multiplication?

<table>
<thead>
<tr>
<th>Process</th>
<th>P0</th>
<th>P1</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>P3</td>
<td>P4</td>
<td>P5</td>
<td></td>
</tr>
<tr>
<td>P6</td>
<td>P7</td>
<td>P8</td>
<td></td>
</tr>
</tbody>
</table>

Matrix tiles:

<table>
<thead>
<tr>
<th></th>
<th>A00</th>
<th>A01</th>
<th>A02</th>
</tr>
</thead>
<tbody>
<tr>
<td>B00</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B01</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B02</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A11</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B11</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A20</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B20</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A21</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B21</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A22</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B22</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
How to compute a matrix multiplication?

![Matrix Multiplication Diagram]

<table>
<thead>
<tr>
<th>Buffer A</th>
<th>Buffer B</th>
<th>Buffer C</th>
</tr>
</thead>
<tbody>
<tr>
<td>A00</td>
<td>B00</td>
<td>A00</td>
</tr>
<tr>
<td>A01</td>
<td>B01</td>
<td>A00</td>
</tr>
<tr>
<td>A02</td>
<td>B02</td>
<td>A00</td>
</tr>
<tr>
<td>A10</td>
<td>B10</td>
<td>A11</td>
</tr>
<tr>
<td>A11</td>
<td>B11</td>
<td>A11</td>
</tr>
<tr>
<td>A12</td>
<td>B12</td>
<td>A11</td>
</tr>
<tr>
<td>A20</td>
<td>B20</td>
<td>A22</td>
</tr>
<tr>
<td>A21</td>
<td>B21</td>
<td>A22</td>
</tr>
<tr>
<td>A22</td>
<td>B22</td>
<td>A22</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
C_{00} &= A_{00} \times B_{00} + A_{01} \times B_{10} + A_{02} \times B_{20} \\
C_{10} &= A_{10} \times B_{00} + A_{11} \times B_{10} + A\ 12 \times B_{20} \\
C_{20} &= A_{20} \times B_{00} + A_{21} \times B_{10} + A_{22} \times B_{20} \\
C_{01} &= A_{00} \times B_{01} + A_{01} \times B_{11} + A_{02} \times B_{21} \\
C_{11} &= A_{10} \times B_{01} + A_{11} \times B_{11} + A_{12} \times B_{21} \\
C_{21} &= A_{20} \times B_{01} + A_{21} \times B_{11} + A_{22} \times B_{21} \\
C_{02} &= A_{00} \times B_{02} + A_{01} \times B_{12} + A_{02} \times B_{22} \\
C_{12} &= A_{10} \times B_{02} + A_{11} \times B_{12} + A_{12} \times B_{22} \\
C_{22} &= A_{20} \times B_{02} + A_{21} \times B_{12} + A_{22} \times B_{22}
\end{align*}
\]

#iterations = \sqrt{\#tiles} = \sqrt{9} = 3
How to compute a matrix multiplication?

### Buffer A

<table>
<thead>
<tr>
<th></th>
<th>A00</th>
<th>A00</th>
</tr>
</thead>
<tbody>
<tr>
<td>A11</td>
<td>A11</td>
<td>A11</td>
</tr>
<tr>
<td>A22</td>
<td>A22</td>
<td>A22</td>
</tr>
</tbody>
</table>

### Buffer B

<table>
<thead>
<tr>
<th></th>
<th>B00</th>
<th>B01</th>
<th>B02</th>
</tr>
</thead>
<tbody>
<tr>
<td>B10</td>
<td>B11</td>
<td>B12</td>
<td></td>
</tr>
<tr>
<td>B20</td>
<td>B21</td>
<td>B22</td>
<td></td>
</tr>
</tbody>
</table>

### Buffer C

<table>
<thead>
<tr>
<th></th>
<th>A00 B00</th>
<th>A00 B01</th>
<th>A00 B02</th>
</tr>
</thead>
<tbody>
<tr>
<td>A11 B10</td>
<td>A11 B11</td>
<td>A11 B12</td>
<td></td>
</tr>
<tr>
<td>A22 B20</td>
<td>A22 B21</td>
<td>A22 B22</td>
<td></td>
</tr>
</tbody>
</table>

**Equations:**

- \( C00 = \text{A00} \times \text{B00} + A01 \times B10 + A02 \times B20 \)
- \( C10 = A10 \times B00 + \text{A11} \times \text{B10} + A12 \times B20 \)
- \( C20 = A20 \times B00 + A21 \times B10 + \text{A22} \times \text{B20} \)

- \( C01 = \text{A00} \times \text{B01} + A01 \times B11 + A02 \times B21 \)
- \( C11 = A10 \times B01 + \text{A11} \times \text{B11} + A12 \times B21 \)
- \( C21 = A20 \times B01 + A21 \times B11 + \text{A22} \times \text{B21} \)

- \( C02 = \text{A00} \times \text{B02} + A01 \times B12 + A02 \times B22 \)
- \( C12 = A10 \times B02 + \text{A11} \times \text{B12} + A12 \times B22 \)
- \( C22 = A20 \times B02 + A21 \times B12 + \text{A22} \times \text{B22} \)
How to compute a matrix multiplication?

\[ C_{00} = A_{00} \times B_{00} + A_{01} \times B_{10} + A_{02} \times B_{20} \]
\[ C_{10} = A_{10} \times B_{00} + A_{11} \times B_{10} + A_{12} \times B_{20} \]
\[ C_{20} = A_{20} \times B_{00} + A_{21} \times B_{10} + A_{22} \times B_{20} \]
\[ C_{01} = A_{00} \times B_{01} + A_{01} \times B_{11} + A_{02} \times B_{21} \]
\[ C_{11} = A_{10} \times B_{01} + A_{11} \times B_{11} + A_{12} \times B_{21} \]
\[ C_{21} = A_{20} \times B_{01} + A_{21} \times B_{11} + A_{22} \times B_{21} \]
\[ C_{02} = A_{00} \times B_{02} + A_{01} \times B_{12} + A_{02} \times B_{22} \]
\[ C_{12} = A_{10} \times B_{02} + A_{11} \times B_{12} + A_{12} \times B_{22} \]
\[ C_{22} = A_{20} \times B_{02} + A_{21} \times B_{12} + A_{22} \times B_{22} \]
How to compute a matrix multiplication?

<table>
<thead>
<tr>
<th>Buffer A</th>
<th>Buffer B</th>
<th>Buffer C</th>
</tr>
</thead>
<tbody>
<tr>
<td>A02</td>
<td>A02</td>
<td>A02</td>
</tr>
<tr>
<td>A10</td>
<td>A10</td>
<td>A10</td>
</tr>
<tr>
<td>A21</td>
<td>A21</td>
<td>A21</td>
</tr>
<tr>
<td></td>
<td>B20</td>
<td>B21</td>
</tr>
<tr>
<td></td>
<td>B00</td>
<td>B01</td>
</tr>
<tr>
<td></td>
<td>B10</td>
<td>B11</td>
</tr>
<tr>
<td></td>
<td>B00</td>
<td>B01</td>
</tr>
<tr>
<td></td>
<td>B10</td>
<td>B11</td>
</tr>
<tr>
<td></td>
<td>A02</td>
<td>A02</td>
</tr>
<tr>
<td></td>
<td>A02</td>
<td>A02</td>
</tr>
<tr>
<td></td>
<td>A02</td>
<td>A02</td>
</tr>
</tbody>
</table>

- C00 = A00 x B00 + A01 x B10 + A02 x B20
- C10 = A10 x B00 + A11 x B10 + A12 x B20
- C20 = A20 x B00 + A21 x B10 + A22 x B20
- C01 = A00 x B01 + A01 x B11 + A02 x B21
- C11 = A10 x B01 + A11 x B11 + A12 x B21
- C21 = A20 x B01 + A21 x B11 + A22 x B21
- C02 = A00 x B02 + A01 x B12 + A02 x B22
- C12 = A10 x B02 + A11 x B12 + A12 x B22
- C22 = A20 x B02 + A21 x B12 + A22 x B22
Wait… But this is not as simple as ScaLAPACK?!
Challenges

● What we know
  ○ Configurations of tensors (S, A, P)

● What we do not know
  ○ How to optimally split an operation (O) subject to constraints
  ○ Resources
  ○ Communication
  ○ Overlapping
Parallelization Strategy

- Partitions an operator $O_i$ into $|c_i|$ independent tasks $t_i:1, \ldots, t_i:|c_i|$
- Configuration also includes the device assignment for each task
  - $t_i:k (1 \leq k \leq |c_i|)$
- Infer the necessary input tensors for each task using the output Tensors
Parallelization Strategy

- Define $S$ that describes one possible parallelization for each operation
- Configurations of each operator can be randomly chosen
Choosing strategy: Execution simulator

- Assumptions
  - Input independent and predictable execution time with low variance
  - Communication bandwidth can be fully utilized
  - FIFO scheduling policy
  - Negligible runtime overhead
Construct a task graph

1. Two kinds of tasks: Normal (compute, communicate, Edge (dependency))
2. Place all tasks of an operation in the graph
3. Connect the input and output tensors (device placement)
4. If two connected tasks are on the same device, add an edge
5. Else, add a communication task
Simulate

- Fill in a number of properties
  - Device
  - exeTime
  - ...
- Run a variant of Dijkstra’s algorithm
- Dequeue in order of ready time

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>exeTime</td>
<td>The elapsed time to execute the task.</td>
</tr>
<tr>
<td>device</td>
<td>The assigned device of the task.</td>
</tr>
<tr>
<td>$I(t)$</td>
<td>${t_{in}</td>
</tr>
<tr>
<td>$O(t)$</td>
<td>${t_{out}</td>
</tr>
</tbody>
</table>

Properties set in simulation

- readyTime: The time when the task is ready to run.
- startTime: The time when the task starts to run.
- endTime: The time when the task is completed.
- preTask: The previous task performed on device.
- nextTask: The next task performed on device.

Internal properties used by the full simulation algorithm

- state: Current state of the task, which is one of NOTREADY, READY, and COMPLETE.
Optimize task graph

- Introduces “Delta simulation algorithm”
  - Change configuration of one operator at a time
  - Only resimulate the affected operations

- Search
  - Use existing strategy (only data parallel, expert tuned) as initial condition
  - Replace one operator configuration at a time (randomly) by a random config
  - Use expected execution as cost function for minimization
Figure 4. Simulating an example parallelization strategy. The tasks’ exeTime and device are shown on the top of each column. In Figure 4c and 4d, the word “r” and “s” indicate the readyTime and startTime of each task, respectively, and the dashed edges indicate the nextTask.
Implementation

- FlexFlow (Not the one in HPCA’17 ?)
- cuDNN
- cuBLAS
- Legion (Task based runtime, SC’12)
# Evaluation: Applications

Table 4. Details of the DNNs and datasets used in evaluation.

<table>
<thead>
<tr>
<th>DNN</th>
<th>Description</th>
<th>Dataset</th>
<th>Reported Acc.</th>
<th>Our Acc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>AlexNet</td>
<td>A 12-layer CNN</td>
<td>Synthetic data</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Inception-v3</td>
<td>A 102-layer CNN with Inception modules (Szegedy et al., 2014)</td>
<td>ImageNet</td>
<td>78.0(^a)</td>
<td>78.0(^a)</td>
</tr>
<tr>
<td>ResNet-101</td>
<td>A 101-layer residual CNN with shortcut connections</td>
<td>ImageNet</td>
<td>76.4(^a)</td>
<td>76.5(^a)</td>
</tr>
<tr>
<td></td>
<td>Convolutional Neural Networks (CNNs)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RNNTC</td>
<td>4 recurrent layers followed by a softmax layer</td>
<td>Movie Reviews (Movies)</td>
<td>79.8%</td>
<td>80.3%</td>
</tr>
<tr>
<td>RNNLM</td>
<td>2 recurrent layers followed by a softmax layer</td>
<td>Penn Treebank (Marcus et al.)</td>
<td>78.4(^b)</td>
<td>76.1(^b)</td>
</tr>
<tr>
<td>NMT</td>
<td>4 recurrent layers followed by an attention and a softmax layer</td>
<td>WMT English-German (WMT)</td>
<td>19.67(^c)</td>
<td>19.85(^c)</td>
</tr>
</tbody>
</table>

\(^a\) top-1 accuracy for single crop on the validation dataset (higher is better).

\(^b\) word-level test perplexities on the Penn Treebank dataset (lower is better).

\(^c\) BLEU scores (Papineni et al., 2002) on the test dataset (higher is better).
Evaluation: Platforms

(a) The P100 Cluster (4 nodes). (b) The K80 Cluster (16 nodes).

Figure 5. Architectures of the GPU clusters used in the experiments. An arrow line indicates a NVLink connection. A solid line is a PCI-e connection. Dashed lines are Infiniband connections across different nodes.
Evaluation: Configuration search

- Data parallelism as initial condition
- 30 minutes search budget
Figure 6. Per-iteration training performance on six DNNs. Numbers in parenthesis are the number of compute nodes used in the experiments. The dash lines show the ideal training throughput.
Three figure of merits

(a) Per-iteration execution time.

(b) Overall data transfers per iteration.

(c) Overall task run time per iteration.

Figure 7. Parallelization performance for NMT on 64 K80 GPUs (16 nodes). FlexFlow reduces per-iteration execution time by 1.7-2.4× and data transfers by 2-5.5× compared to other approaches. FlexFlow achieves similar overall task computation time as expert-designed strategy, which is 20% fewer than data parallelism.
End-to-end training performance

*Figure 8.* Training curves of Inception-v3 in different systems. The model is trained on 16 P100 GPUs (4 nodes).
Speedup of configuration search time using Delta

Table 5. The end-to-end search time with different simulation algorithms (seconds).

<table>
<thead>
<tr>
<th>Num. GPUs</th>
<th>AlexNet</th>
<th>ResNet</th>
<th>Inception</th>
<th>RNNTC</th>
<th>RNNLM</th>
<th>NMT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Full</td>
<td>Delta</td>
<td>Speedup</td>
<td>Full</td>
<td>Delta</td>
<td>Speedup</td>
</tr>
<tr>
<td>-----------</td>
<td>----------</td>
<td>--------</td>
<td>-----------</td>
<td>-------</td>
<td>-------</td>
<td>-----</td>
</tr>
<tr>
<td>4</td>
<td>0.11</td>
<td>0.04</td>
<td>2.9×</td>
<td>14</td>
<td>4.1</td>
<td>3.4×</td>
</tr>
<tr>
<td>8</td>
<td>0.40</td>
<td>0.13</td>
<td>3.0×</td>
<td>66</td>
<td>17</td>
<td>3.9×</td>
</tr>
<tr>
<td>16</td>
<td>1.4</td>
<td>0.48</td>
<td>2.9×</td>
<td>388</td>
<td>77</td>
<td>5.0×</td>
</tr>
<tr>
<td>32</td>
<td>5.3</td>
<td>1.8</td>
<td>3.0×</td>
<td>1746</td>
<td>298</td>
<td>5.9×</td>
</tr>
<tr>
<td>64</td>
<td>18</td>
<td>5.9</td>
<td>3.3×</td>
<td>8817</td>
<td>1278</td>
<td>6.9×</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>9.2</td>
<td>2.3×</td>
<td>40</td>
<td>16</td>
<td>2.5×</td>
</tr>
<tr>
<td></td>
<td>178</td>
<td>65</td>
<td>2.7×</td>
<td>998</td>
<td>328</td>
<td>3.0×</td>
</tr>
<tr>
<td></td>
<td>2698</td>
<td>701</td>
<td>3.8×</td>
<td>8982</td>
<td>2190</td>
<td>4.1×</td>
</tr>
</tbody>
</table>
Simulation Accuracy

Figure 10. Comparison between the simulated and actual execution time for different DNNs and device topologies.
Figure 12. The best discovered strategy for parallelizing Inception-v3 on 4 P100 GPUs. For each operator, the vertical and horizontal dimensions indicate parallelism in the sample and parameter dimension, respectively. Each GPU is denoted by a color.
Figure 13. The best discovered strategy for parallelizing NMT on 4 P100 GPUs. For each operator, the vertical and horizontal dimensions indicate parallelism in the sample and parameter dimension, respectively. Each grey box denotes a layer, whose operators share the same network parameters. Each GPU is denoted by a color.
Conclusion

- FlexFlow that uses SOAP: A lower granularity of parallelization
- Transforming into a “task” runtime problem
- Uses traditional optimization techniques
Discussion

- What are the alternatives to task based runtime in this context?
- Can this scheme be modeled as a traditional parallel application?
- Is the full bandwidth utilization assumption over optimistic?
Data Parallelism (DP)
- Compute/communication efficiency
- Poor memory efficiency

Model Parallelism (MP)
- Favourable memory efficiency
- Poor compute/communication efficiency

Both keep all the model states over entire training process
Zero Redundancy Optimizer (ZeRO)

Goal: Achieve the best of both worlds

Contribution:

Reduce per-device memory footprint linearly with the increased degree of parallelism while keeping communication close to that of default DP

1. Improved training speed for large models
2. Independence of model size
An Example

1.5B parameter GPT-2 trained with ADAM

- Weights/parameters: 3GB with fp16 (2\(\Psi\))
- Gradients: 3GB with fp16 (2\(\Psi\))
- Optimizer state: fp32 copy of parameters, momentum, variance \(\rightarrow 18GB (4\Psi + 4\Psi + 4\Psi)\)

Residual memory:

- Activations: for a GPT like model \(12 \times \text{hidden_dim} \times \text{batch} \times \text{seq_length} \times \text{transformer_layers}\) (60 GB)
  - Checkpointing: trading-off memory for computation
- Temporary buffers: Gradient fusion for improved device throughput (6GB)
- Memory fragmentation: long-lived vs. short-lived memory
  - Reduces practically available amount of memory
  - OOM with over 30% memory still available
## Optimizing Model State Memory (ZeRO-DP)

**Assumption:** For large models, the majority of the memory is occupied by model states which includes optimizer states (momentum, variances), gradients and parameters.

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Parameters</th>
<th>Gradients</th>
<th>Optimizer States</th>
<th>Memory Consumed</th>
<th>Memory Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td></td>
<td></td>
<td></td>
<td>$(2 + 2 + K) \cdot \Psi$</td>
<td>120GB</td>
</tr>
<tr>
<td>$P_{os}$</td>
<td></td>
<td></td>
<td></td>
<td>$2\Psi + 2\Psi + \frac{K \cdot \Psi}{N_d}$</td>
<td>31.4GB, 4x</td>
</tr>
<tr>
<td>$P_{os+g}$</td>
<td></td>
<td></td>
<td></td>
<td>$2\Psi + \frac{(2+K) \cdot \Psi}{N_d}$</td>
<td>16.6GB, 8x</td>
</tr>
<tr>
<td>$P_{os+g+p}$</td>
<td></td>
<td></td>
<td></td>
<td>$\frac{(2+2+K) \cdot \Psi}{N_d}$</td>
<td>Linear with $N_d$, 1.9GB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Gradients</th>
<th>Optimizer States</th>
</tr>
</thead>
<tbody>
<tr>
<td>Green</td>
<td>Orange</td>
<td>Blue</td>
</tr>
</tbody>
</table>

Memory Reduction:

- **4x**
- **8x**
- Linear with $N_d$
Optimizing Residual State Memory (Zero-R)

That is: **Activation**, **temporary buffers** and unusable **fragmented memory**.

1. Activation partitioning and CPU offloading
2. Constant size temporary buffer size
3. Proactive management of memory with respect to tensor lifetime.
Do we still need MP, and when?

ZeRO-DP is at least as effective in reducing memory as MP, or even more effective when MP cannot divide model evenly. + scales better.

1. MP can be extended with ZeRO-R
2. Smaller models, MP might have better convergence due to large batch size in Zero-DP.
Relation to Other Optimizations

Pipeline Parallelism:
  Often incurs functionality, performance and convergence

Activation Memory Optimization:
  Compression, checkpointing or live analysis (complementary)

CPU Offloading:
  Can be avoided due to memory reduction

Memory efficient optimizers:
  Impact convergence (orthogonal, ZeRO does not change the optimizer)

Training Optimizers:
  ZeRO makes more sophisticated optimizers possible
ZeRO-DP

Zero-DP
GPU0 initially has parameters of M0 -> broadcast
1,2,3 delete parameters and 1 continues broadcasting parameters of M1
Forward Pass complete -> Loss
The Backward Pass starts on M3

GPU\textsubscript{0,1,2} will hold a temporary buffer M\textsubscript{3} gradients on Data\textsubscript{0,1,2}
M3 on all devices

The backwards pass proceeds on $M_3$.
The activations for $M_3$ are recomputed from the saved partial activations.
GPU_{0,1,2} pass their M_3 gradients to GPU_3
GPU_3 performs gradient accumulation and holds final M_3 for all Data
To Start M2, GPU2 broadcasts parameters

GPU$_2$ passes M$_2$’s parameters to GPU$_{0,1,3}$ so they can run the backwards pass and compute gradients for M$_2$
Update Parameters For Local Partition

Now every GPU has its respective gradients (accumulated from all datasets)
We can compute the updated parameters
Optimizer Runs in Parallel

The optimizer runs
Iteration Complete

The fp16 weights become the model parameters for the next iteration.
Training iteration complete!
All-gather over parameters is spread over entire forward pass, but needs to happen again for backward pass as parameters are discarded.
Results

ZeRO vs. Megatron-LM (MP) and PyTorch Distributed Data Parallel (Baseline without MP)

400 V100 GPUs
Super-Linear Scalability - 60B parameters
Trillion Parameters Possible?

Theoretically yes, when combined with MP and with 1024 GPUs

- 16-way model parallelism (intra DGX-2 node)
- 64-way data parallelism

Turing-NLP: 17B

GPT-3: 175B