Parallel and Distributed training of DNNs

From GPUs to clusters

Mark Silberstein
mark@ee.technion.ac.il
Plan

- Reminder: parallelization principles
- Parallelizing DNN training: high level overview
- Acceleration on GPUs: why it works
- Distributed training approaches
- Research opportunities
Parallelization → problem partitioning

- Domain decomposition
  - (SPMD)
  - Input domain
  - Output domain
  - Both

- Functional decomposition
  - (MPMD)
  - Independent tasks
  - Pipelining
Performance of parallel programs

• Intuitive measures
  – Wall clock time
  – Speedup = (Serial time)/(Parallel time)
  – GFLOPs/s = how well the hardware is exploited

• Scaling
  – Scalability: Speedup as a function of #CPUs
  – Efficiency: Speedup/#CPUs
Upper bound on speedup
Amdahl's law

• Sequential component limits the speedup
• Split program into
  – totally parallel: $A$ ($0 \leq A \leq 1$)
  – totally sequential: $1-A$

Scalability = Speedup($\#$CPUs)
= $1/(A/\#$CPUs + (1-A))
Bad news

Source: wikipedia
Strong scaling/weak scaling

• Strong scaling (Amdahl's law)
  - problem size is fixed: *solve same problems faster*

• Week scaling
  - Problem size grows: *make larger problems feasible*
  - Keep the amount of work per CPU when adding more CPUs
Main sources of inefficiency

- Wrong choice of granularity for a given platform
- Parallel program may need more work to do
- Data transfer overhead
- Control
  - Synchronization
Parallelizing DNN training
DNN training – birds view

For each batch

For each Layer L  //forward

For each Sample S in a batch

Compute  \( L(w,s) \)

Endfor (samples)

Endfor (forward)

Compute Error

For each Layer L  //backward

For each Sample S in a batch

Compute  \( \nabla L(w,s) \)

Compute gradient for output of prev. layer for sample

Endfor (samples)

\[ \nabla L = \frac{1}{|S|} \sum_s \nabla L(w,s) \]

Endfor (layers)

Update weights

Endfor
DNN training: hierarchical parallelism

For each batch

For each Layer L

//forward

For each Sample S in a batch

Compute $L(w, s)$

Endfor (samples)

Endfor (forward)

Compute Error

For each Layer L

//backward

For each Sample S in a batch

Compute $\nabla L(w, s)$

Compute gradient for output of prev. layer for sample

Endfor (samples)

$\nabla L = \frac{1}{|S|} \sum_s \nabla L(w, s)$

Endfor (layers)

Update weights

Endfor
DNN training: hierarchical parallelism

For each batch
   For each Layer L //forward
      For each Sample S in a batch
         Compute $L(w, s)$
      Endfor (samples)
   Endfor (forward)
   Compute Error
   For each Layer L //backward
      For each Sample S in a batch
         Compute $\nabla L(w, s)$
         Compute gradient for output of prev. layer for sample
      Endfor (samples)
      $\nabla L = 1/|S| \sum_s \nabla L(w, s)$
   Endfor (layers)
   Update weights
Endfor

Fine-grain parallelism

Coarse-grain parallelism
This lecture

- Fine grain parallelism - GPUs
- Data parallelism
- + Relaxation of SGD
  - Lockless updates (Hogwild!)
  - Asynchronous SGD (Downpour, Adam, Dogwild!)
- + Model Parallelism
- + Convolution-specific
Why GPUs

Linear algebra accelerators
Why GPUs

circa 2015

![CUDNN V2 - PERFORMANCE](image)

- Alexnet: 9, 17, 1
- Caffenet: 9, 17, 8
- GoogLeNet: 16

CPU is 16 core Haswell E5-2698 at 2.3 GHz, with 3.6 GHz Turbo
GPU is NVIDIA Titan X
Simplified GPU model
GPU 101

CPU

Memory

GPU

Memory
GPU is a co-processor

CPU
Computation

Memory

GPU

Memory
GPU is a co-processor
Co-processor model

CPU

GPU kernel

GPU

Memory
GPU is a co-processor
Simple GPU program

- Idea: **same set** of operations is applied to different data chunks *in parallel*

- Algorithmic challenge – identify independent tasks

- Implementation
  - Every *thread* runs the same code on different data chunks.
  - GPU concurrently runs many parallel threads
Vector sum $C = A + B$

- Sequential algorithm
  
  For every element $i$
  
  $C[i] = A[i] + B[i]$
Vector sum $C = A + B$

- **Sequential algorithm**
  
  For every element $i$
  
  $C[i] = A[i] + B[i]$

- **Parallel algorithm**
  
  *In parallel every $i*$
  
  $C[i] = A[i] + B[i]$
Implementation for a vector of length 1024

- GPU kernel (this program runs in every thread)
  \[ C[\text{threadId}] = A[\text{threadId}] + B[\text{threadId}] \]
Implementation for a vector of length 1024

- **GPU**

  \[ C[\text{threadId}] = A[\text{threadId}] + B[\text{threadId}] \]

- **CPU**

  1. Allocate three arrays in GPU memory
  2. Copy data CPU -> GPU
  3. Invoke kernel with 1024 threads
  4. Wait until complete and copy data GPU->CPU
GPUs are good for..

• Computations
  – Well-structured
  – Massive parallelism
  – Lots of FP/DP operations
  – High memory bandwidth requirements
  – Small-medium memory footprint (up to 12-16 GB)
GPU hardware
GPU hardware characteristics

- Massive parallelism
- Low serial performance
GPU hardware parallelism
1. Multi-core
GPU hardware parallelism

2. SIMD
GPU hardware parallelism

3. Parallelism for latency hiding

![Diagram showing GPU hardware parallelism with execution states and threads T1, T2, T3.](image-url)
GPU Hardware
3. Parallelism for latency hiding

![Diagram of GPU hardware with parallel execution paths T1, T2, T3 and memory access R 0x01]
GPU Hardware
3. Parallelism for latency hiding

![Diagram showing GPU hardware with parallelism for latency hiding]
GPU Hardware
3. Parallelism for latency hiding

![Diagram of GPU hardware showing parallel execution states and memory access]

- GPU memory
- Execution state
- R 0x01
- R 0x04
- R 0x08
- T1
- T2
- T3
3. Parallelism for latency hiding
Putting it all together: 3 levels of hardware parallelism

[Diagram showing 3 levels of hardware parallelism: GPU, GPU memory, Core, Core, Core, Core, SIMD vector, State 1, State k]
Software-Hardware mapping

GPU

Core

Core

Core

Core

GPU memory

State 1

... 

State k

SIMD vector

Thread n

Thread 1
Takeaway 1: 10,000-s of concurrent threads!

NVIDIA Pascal GPU: $64 \times 60 \times 32 = 122880$ concurrent threads
Takeaway 2: One thread is slow

~100x slower than a CPU thread
GPUs in DNN

• Benefits: huge boost in FP performance
  – Fast BLAS, FFT, Convolutions
  – cuDNN/cuBLAS/cuFFT/MaxDNN..
  – “Free” - no loss in precision

• Challenges
  – Data arrangement
  – Memory size limit
  – CPU-GPU data transfers
  – Development
Data parallelism: batch-level parallelization
DNN training

For each batch
  For each Layer L                               //forward
    For each Sample S in a batch
      Compute $L(w, s)$
    Endfor (samples)
  Endfor (forward)
  Compute Error
  For each Layer L                               //backward
    For each Sample S in a batch
      Compute $\nabla L(w, s)$
      Compute gradient for output of prev. layer for sample
    Endfor (samples)
    $\nabla L = \frac{1}{|S|} \sum_s \nabla L(w, s)$
  Endfor (layers)
  Update weights
Endfor
DNN training: data parallelism

For each batch

Node \( k \) chooses sub-batch

For each Layer \( L \) //forward
  For each Sample \( S \) in a batch
    Compute \( L(w,s) \)
  Endfor (samples)
Endfor (forward)

Compute Error

For each Layer \( L \) //backward
  For each Sample \( S \) in a batch
    Compute \( \nabla L(w,s) \)
    Compute gradient for output of prev. layer for sample
  Endfor (samples)
  \[ \nabla L = \frac{1}{|S|} \sum_s \nabla L(w,s) \]
Endfor (layers)

Update weights

Endfor
DNN training: data parallelism

For each batch

Node $k$ chooses sub-batch

For each Layer $L$ //forward

For each Sample $S$ in a batch

Compute $L(w,s)$

Endfor (samples)

Endfor (forward)

Compute Error

For each Layer $L$ //backward

For each Sample $S$ in a batch

Compute $\nabla L(w,s)$

Compute gradient for output of prev. layer for sample

Endfor (samples)

$\nabla L=1/|S|\sum_s \nabla L(w,s)$

Endfor (layers)

Send local $\nabla L(w,s_{local})$

Receive $\nabla L(w,s_{others})$ from remote nodes and update weights

Endfor
DNN training: data parallelism

For each batch

Node $k$ chooses sub-batch

For each Layer $L$ //forward
  For each Sample $S$ in a batch
    Compute $L(w,s)$
  Endfor (samples)
Endfor (forward)

Compute Error

For each Layer $L$ //backward
  For each Sample $S$ in a batch
    Compute $\nabla L$
    Compute gradient for output of prev. layer for sample
  Endfor (samples)
Endfor (layers)

$\nabla L = \frac{1}{|S|} \sum_s \nabla L(w,s)$

Send local $\nabla L(w, s_{local})$

Receive $\nabla L(w, s_{others})$ from remote nodes and update weights

Training different models, reconciling once per iteration
Parallelization via Master Worker

\[ \nabla L(w, s_0) \]
\[ \nabla L(w, s_1) \]

Parameter server

Worker

Sub-batch 0

Worker

Sub-batch 1

\[
\nabla L(w, s_0)
\]
\[
\nabla L(w, s_1)
\]

w

w
Problem 1: scalability limit

Time until convergence

Machines
Scalability vs. Convergence

- Increase batch size – improve scalability but affect convergence
- Decrease batch size – improve convergence but communications become the bottleneck
- The minimum batch size is dictated by the efficiency of the local implementation
Figure 5: Time to reach a fixed accuracy (16%) for different optimization strategies as a function of number of the machines (left) and cores (right).

Fully connected 5 layers
42*10^6 parameters
Problem 2: synchronization

For each batch

Node $k$ chooses sub-batch

For each Layer $L$ //forward
  For each Sample $S$ in a batch
    Compute $L(w, s)$
  Endfor (samples)
Endfor (forward)
Compute Error
For each Layer $L$ //backward
  For each Sample $S$ in a batch
    Compute \( \nabla L(w, s) \)
    Compute gradient for output of prev. layer for sample
  Endfor (samples)
Endfor (layers)
Send local \( \nabla L(w, s_{local}) \)
Receive \( \nabla L(w, s_{others}) \) from remote nodes and update weights

Synchronization point!
Main optimization goals

1. Minimize data transfer overhead
2. Minimize synchronization overhead
Efficient communications: 
Single parameter server bottleneck

- Use reduction tree instead of a single parameter server
- **FireCaffe**: GoogleNet on 128 GPUs – 47x faster
- Batch size = 1024
- Meta-parameter tuning necessary to compensate for huge batches
Asynchronous updates

• SGD iterations are dependent
• Updates to weights needs to be synchronized across nodes: $w += \Delta w$
• Idea: allow asynchronous updates/overwrites/losses
• When would it work?
  – SGD sparse models (look up in Hogwild!)
  – Practically works (look up Dogwild!, Adam MSR, Downpour Google, some others)
How it works

Large Scale Distributed Deep Networks

Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen,

Parameter Server \[ w' = w - \eta \Delta w \]

Model Replicas

Data Shards

Project Adam: Building an Efficient and Scalable Deep Learning Training System

Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, Karthik Kalyanaraman

Global Model Parameter Store

Model Replica

W \Delta W

Model Workers

Data Shards
Scalability vs. convergence

• More asynchrony – **better scalability** – faster iteration
• However – slower convergence!
• The benefits are problem dependent
• Asynchronous training adds noise
• Excellent analysis in

GeePS: Scalable deep learning on distributed GPUs with a GPU-specialized parameter server
Low level system tricks

- Dogwild!: use unreliable network transfers for efficiency
- Project Adam: optimize for NUMA and L3 cache, lockless local updates
- DistBelief: optimize load balancing and stragglers
Data parallelism: inherently not scalable

- Large batch size reduces convergence
- Larger models spill out of limited (GPU) memory
- Too much communications: in particular fully connected
  - Too many weights to exchange
Model parallelism: more complex

For each batch

For each Layer $L$  
  //forward
    For each Sample $S$ in a batch
      Compute $L(w, s)$
    Endfor (samples)
  Endfor (forward)

Compute Error

For each Layer $L$  
  //backward
    For each Sample $S$ in a batch
      Compute $\nabla L(w, s)$
      Compute gradient for output of prev. layer for sample
    Endfor (samples)
  Endfor (layers)

Update weights

Endfor
How it is done

Large Scale Distributed Deep Networks

Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen,
How it is done

Deep learning with COTS HPC systems

Adam Coates
acoates@cs.stanford.edu

Distributed array abstraction helps
What about convnets?

“One weird trick...” by A. Krizhevsky

- **Observation:**
  - Convolutional layers – 90-95% computations, 5% of parameters
  - Fully connected – the opposite

- **Conclusion:** data parallel where few weights, model parallel where many weights, smaller feature maps
How it works

Model parallelism:
all workers train on same batch;
workers communicate as frequently as
network allows.

Data parallelism:
each worker trains the same
convolutional layers on a different
data batch.
Hiding network overheads
How well does it work on GPUs?

One weird trick for parallelizing convolutional neural networks

Alex Krizhevsky

<table>
<thead>
<tr>
<th>GPUs</th>
<th>Batch size</th>
<th>Cross-entropy</th>
<th>Top-1 error</th>
<th>Time</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(128, 128)</td>
<td>2.611</td>
<td>42.33%</td>
<td>98.05h</td>
<td>1x</td>
</tr>
<tr>
<td>2</td>
<td>(256, 256)</td>
<td>2.624</td>
<td>42.63%</td>
<td>50.24h</td>
<td>1.95x</td>
</tr>
<tr>
<td>2</td>
<td>(256, 128)</td>
<td>2.614</td>
<td>42.27%</td>
<td>50.90h</td>
<td>1.93x</td>
</tr>
<tr>
<td>4</td>
<td>(512, 512)</td>
<td>2.637</td>
<td>42.59%</td>
<td>26.20h</td>
<td>3.74x</td>
</tr>
<tr>
<td>4</td>
<td>(512, 128)</td>
<td>2.625</td>
<td>42.44%</td>
<td>26.78h</td>
<td>3.66x</td>
</tr>
<tr>
<td>8</td>
<td>(1024, 1024)</td>
<td>2.678</td>
<td>43.28%</td>
<td>15.68h</td>
<td>6.25x</td>
</tr>
<tr>
<td>8</td>
<td>(1024, 128)</td>
<td>2.651</td>
<td>42.86%</td>
<td>15.91h</td>
<td>6.16x</td>
</tr>
</tbody>
</table>

Same trick used in Project Adam, and Baidu with 64 GPUs
Research question: scaling

- Truly distributed learning
- Efficient inter-GPU communications
- Exploiting sparsity in training
- Space efficiency
Accelerated Systems Lab

- Systems security
- Distributed DNN
- GPU OS and I/O abstractions
- Smart NICs

Interested?
Ask me!

mark@ee.technion.ac.il