Device Placement with Reinforcement Learning

Azalia Mirhoseini, Hieu Pham, Quoc V. Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, Jeff Dean
Several keywords

- Device Placement
- Reinforcement Learning
- Model Description
- Architecture Details
- Experiments
Device placement?
Heterogeneous distributed environment

*What?*

- Merging CPU and GPU environment
- Neural network training
- How to make the system works and improve performance
Heterogeneous distributed environment

Why?

• Different purposes

  • CPU: fewer cores, higher frequency, more complicated and faster cache mechanism;

  • GPU: much more cores, SIMD, harder to program.
Heterogeneous distributed environment

Why, again?

- Balanced workload

[Diagram showing matrix multiplication (SGEMM) with execution time in nanoseconds for different configurations. The best configuration is noted as (80,20).]

https://www.ntnu.edu/documents/139931/1275097249/NTNU_HetComp_toPublish.pdf/486588ee-23af-4104-8a04-bb18cd5a68c1
Heterogeneous distributed environment

Why, even more?

- Decrease data communication
- Increase flexibility and reliability
- Increase energy efficiency
Heterogeneous distributed environment

Why not?

- Programming difficulties - solutions:
  - Develop human-friendly libraries (tensorflow-gpu)
  - Hire more graduate students
- Workload balance?
  - Hard to implement and optimize
Heterogeneous distributed environment

How?

• Human specify:
• Questionable;
• Low reusability/generalizability
• Algorithm:
• Hard to tune the hyperparameters.
Reinforcement Learning?
Reinforcement Learning?

What?
- Agents (Problem)
- Exploration (Actions)
- An environment (Markov decision process)
- Cumulative reward (Feedback)
Reinforcement Learning?

Why?
A good strategy;
A good policy-finder.

Isn’t reinforcement learning just genetic algorithm?

• Mutation and evolution;
• (Kind of) metaheuristics optimization;
• Search and memory;
Reinforcement Learning?

Where?

Real-time strategy game

Atari game
Model description
Question description

Minimize: \( r(P) \)

Where:

\( M \) operations: \( \{o_1, o_2, \ldots, o_M\} \);
\( D \) available devices;
\( P \) is a placement: an assignment of an operation \( o_i \in G \) to a device \( p_i \), where \( p_i \in \{1, \ldots, D\} \).
\( r(P) \) is the complete execution time of a TensorFlow graph \( G \) under placement \( P \).

In plain word: under different placement \( P \), minimize the corresponding execution time.
Why can’t we just do the math, based on $r(P)$?

1. Bad placements sampled in the beginning is noisy: will lead to bad learning signals.
2. Near converged, different placement is hard to distinguish.

Empirically change a more robust optimization target:

$$R(P) = \sqrt{r(P)}$$
Markov assumption

Supposing a system consisting of several states $s_t$ and corresponding action between subsequent action $a_t$

$$p(s_{t+1}|s_t, a_t) = f(s_t, a_t)$$

Basically: next state is only determined by current state and the action.

*Remember dynamic programming?*

What do we want?

Supposing a policy/action/arrangement/placement \( \pi(\theta) \), which is parameterized by \( \theta \).

Our goal is to find:

\[
\theta^* = \arg \max_{\theta} J(\theta)
\]

Basically: Find the best parameters for our \( \pi \)

What is my reward?

We will need to assign a reward for our decision/policy:

\[ J(\theta) = \] 

Basically: Find the best parameters for our policy with shortest time.

What the heck does this mean!

\[ J(\theta) = E_{P \sim \pi(P | G; \theta)} [R(P)|G] \]

1. Why there's an E?
   R(P) is a stochastic process;
2. What is \( \pi(P | G; \theta) \)?
   It is defined by an attentional sequence-to-sequence model
3. What is \( P \sim \pi(P | G; \theta) \)?
   P follows the distribution of policy we gave, which is also predetermined by \( \theta \);
Policy gradient - REINFORCE equation

\[ \nabla_\theta J(\theta) = E_{P \sim \pi(P|G; \theta)} [R(P) \cdot \nabla_\theta \log p(P|G; \theta)] \]

- This is a famous equation!
- Explicit learning – not (or less) a black box.

\[ \nabla_\theta J(\theta) \approx \frac{1}{K} \sum_{i=1}^{k} (R(P_i) - B) \cdot \nabla_\theta \log p(P|G; \theta) \]

- It works!
- Scale-down using a baseline term B
- Easier

Difficulties in practice

- Graphs with large memory footprint: fail to execute. Instead, set a large failing signal, manually.
- Some sporicidal and unexpected fail: especially after higher steps. Instead, hard code the training process.
- Initialize the baseline B with the failing signal results in more exploration.
Architecture details
sequence-to-sequence model

- With LSTM (Long short-term memory);
- And a content-based attention mechanism;
Encoder

- Input: Sequence of operations
- Embed the operations by concatenating
- Store a tunable embedding vector for each layer
Decoder

- Attentional LSTM
- Same steps as operations in graph G
- Output: Each step outputs the device for the operation at the same encoder time step
Co-locating operations

- A lot of operations in a graph;
- Vanishing and exploding gradient;
- Large memory footprints;

Solution: manually/heuristically forcing several operations located on same device.

With collocate_with feature of TensorFlow
Distributed training

- Asynchronous distributed
- Several controllers;
- Not storing input graph parameters on server to avoid latency;
- Each controller control K workers.
  - Execute 10 steps for each benchmark
  - 20 controller; each with 4 – 8 workers
  - Takes 12 – 27 hours to find best placement
Experiments
Benchmark models

Three established deep learning models:

- Recurrent Neural Network Language Model (RNNLM) with 2 LSTM layers (size = 2048 and 1024): with Adam.
- Neural Machine Translation with attention mechanism (NMT) (size = 2048 and 1024): with Adam.
- Inception-V3 (299 × 299 × 3): with RMSProp

Co-location groups:

<table>
<thead>
<tr>
<th>Model</th>
<th>#operations</th>
<th>#groups</th>
</tr>
</thead>
<tbody>
<tr>
<td>RNNLM</td>
<td>8943</td>
<td>188</td>
</tr>
<tr>
<td>NMT</td>
<td>22097</td>
<td>280</td>
</tr>
<tr>
<td>Inception-V3</td>
<td>31180</td>
<td>83</td>
</tr>
</tbody>
</table>
Device

1 Intel Haswell 2300 CPU with 18 cores
OR
2 or 4 Nvidia Tesla K80 GPUs

50 GB of RAM
Baselines

Comparison with the trained placement:

• Single-CPU;
• Single-GPU;
• Scotch static mapper;
• MinCut
• Expert-design
## Single-Step Runtime Efficiency

<table>
<thead>
<tr>
<th>Tasks</th>
<th>Single-CPU</th>
<th>Single-GPU</th>
<th>#GPUs</th>
<th>Scotch</th>
<th>MinCut</th>
<th>Expert</th>
<th>RL-based</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>RNNLM (batch 64)</td>
<td>6.89</td>
<td>1.57</td>
<td>2</td>
<td>13.43</td>
<td>11.94</td>
<td>3.81</td>
<td>1.57</td>
<td>0.0%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td>11.52</td>
<td>10.44</td>
<td>4.46</td>
<td>1.57</td>
<td>0.0%</td>
</tr>
<tr>
<td>NMT (batch 64)</td>
<td>10.72</td>
<td>OOM</td>
<td>2</td>
<td>14.19</td>
<td>11.54</td>
<td>4.99</td>
<td>4.04</td>
<td>23.5%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td>11.23</td>
<td>11.78</td>
<td>4.73</td>
<td>3.92</td>
<td>20.6%</td>
</tr>
<tr>
<td>Inception-V3 (batch 32)</td>
<td>26.21</td>
<td>4.60</td>
<td>2</td>
<td>25.24</td>
<td>22.88</td>
<td>11.22</td>
<td>4.60</td>
<td>0.0%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td>23.41</td>
<td>24.52</td>
<td>10.65</td>
<td>3.85</td>
<td>19.0%</td>
</tr>
</tbody>
</table>

**Diagram:**

- **LSTM 2**
- **LSTM 1**
- **Embedding**
- **Softmax**
- **Attention**
- **LSTM 2**
- **LSTM 1**
- **Embedding**
For each model…

- **RNNLM**: detects that it is possible to fit the whole RNNLM graph into one GPU, to save communication latencies.
- **Neural MT**: Learns to put the less computationally expensive operations, such as embedding lookups, on the CPU.
- **Inception-V3**: when only 2 GPUs available, the degree of freedom for model parallelism is limited. It thus places all the operations on a single GPU (although it could use 2 GPUs).

Again, unexpected results through searching!
End-to-End Runtime Efficiency

• Not only the single-step-wise training performance.
• But also the entire training process.
• Strangely, the paper only presented NMT and Inception-V3
NMT

Expert-designed placement
229.57 hours

RL-based placement
165.73 hours
Inception-V3

Data parallelism, rather than model parallelism.

Asynchronous towers: puts one replica of the Inception-V3 network on each GPU.

Synchronous Tower: the same as Asynchronous towers, except that it waits for the gradients of all copies before making an update.
Computational load profiling of NMT model

RL-based placement balances the workload significantly better than does the expert-designed placement.

The imbalance is much more significant when back-propagation time is considered.
Computational load profiling of Inception-V3

RL-based placement does not seek to balance the computations between GPUs.

Inception-V3 has more dependencies than NMT, allowing less room for model parallelism across GPUs.
Memory copy profiling of Inception-V3

RL-based model parameters are on the same device as the operations that use them.

Synchronous tower model has to wait for all parameters have to be updated and sent to them.

Reduce the communication cost, leading to overall reduction in computing time.
vDNN: Virtualized Deep Neural Networks for Scalable, Memory-Efficient Neural Network Design

Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, Stephen W. Keckler
### Remember this?

**Out of memory!**

<table>
<thead>
<tr>
<th>Tasks</th>
<th>Single-CPU</th>
<th>Single-GPU</th>
<th>#GPUs</th>
<th>Scotch</th>
<th>MinCut</th>
<th>Expert</th>
<th>RL-based</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>RNNLM (batch 64)</td>
<td>6.89</td>
<td>1.57</td>
<td>2</td>
<td>13.43</td>
<td>11.94</td>
<td>3.81</td>
<td>1.57</td>
<td>0.0%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td>11.52</td>
<td>10.44</td>
<td>4.46</td>
<td>1.57</td>
<td>0.0%</td>
</tr>
<tr>
<td>NMT (batch 64)</td>
<td>10.72</td>
<td>OOM</td>
<td>2</td>
<td>14.19</td>
<td>11.54</td>
<td>4.99</td>
<td>4.04</td>
<td>23.5%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td>11.23</td>
<td>11.78</td>
<td>4.73</td>
<td>3.92</td>
<td>20.6%</td>
</tr>
<tr>
<td>Inception-V3 (batch 32)</td>
<td>26.21</td>
<td>4.60</td>
<td>2</td>
<td>25.24</td>
<td>22.88</td>
<td>11.22</td>
<td>4.60</td>
<td>0.0%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td>23.41</td>
<td>24.52</td>
<td>10.65</td>
<td>3.85</td>
<td>19.0%</td>
</tr>
</tbody>
</table>

Why GPU memory is so small compared to main memory?

- Ultra-high bandwidth
- Cannot just add more memory chips
- Increased electrical capacity & Decreased resistance
- Spoil the timing on the memory bus.
- Conspiracy: just another NVIDIA marketing strategy?
Memory management is significant

12 GB: the memory capacity of the state-of-the-art NVIDIA Titan X

Maximum fraction of baseline allocation actually utilized when traversing layer-wise.
Motivations

1. DNNs trained via stochastic gradient-descent (SGD) are designed and structured with multiple layers

2. The training involves a series of layer-wise computations, the order of which is statically fixed and repeated for numerous iterations throughout the entire training process
Common sense - Forward and backward propagation

- Forward propagation: serialized process, highly dependent. **Layer-wise.**
- Backward propagation

\[
\frac{\partial \text{Loss}}{\partial X_{(N)}} = \frac{\partial \text{Loss}}{\partial Y_{(N)}} \cdot \frac{\partial Y_{(N)}}{\partial X_{(N)}}
\]

input gradient maps (dY) output gradient maps (dX)

Deriving the value of dx for layer(n) generally requires memory for both its input/output gradient maps (dy and dx) but also the input/output feature maps (X and Y) for this layer: a lot of memory
Common sense - Batch size

• Training a network involves both forward and backward propagation, which are repeated for millions to billions of iterations

• Nature of SGD-based backward propagation, the network input is generally batched with hundreds of images

• Batch increases memory allocation size but helps the network model better converge to an optimal solution.
Motivation - again

Even though the GPU can, process a single layer’s computation, popular ML frameworks adopt a network-wide memory allocation policy

- Popular ML frameworks suffer from severe limitations in the way they allocate and manage memory.
- Sacrifice memory use to improve performance
• Existing schemes overprovision the memory allocations to network-wise requirements, even though the GPU only use part of the memory for the layer-wise requirements.

• 53% to 79% of memory is wasted.
Memory page?

- Memory is **messy**;
- Humans are **lazy**;
- Machines are **stupid**.

- **Page-migration** based virtualization solutions will cause **more I/O overhead**:
  - page transfer
  - page table update.
Memory mechanism

- To reduce the latency of memory access by moving pages near to the processor where the process accessing that memory is running.
- It’s slow! 80 – 200 MB/sec compared with DMA cudaMemcpy: 12.8 – 16 GB/sec.
- Could take 10s per GB.
Find the gaps

• First, the intermediate feature maps (blue) and workspace (red) incur an order of magnitude higher memory usage compared to the weights.

• Second, most of these intermediate data structures are concentrated on the feature extraction layers and are less significant in the later classifier layers.

• Third, the weights, smaller compared to these intermediate data, are concentrated on the classifier layers due to their full connectivity.
How did we save memory traditionally?

• Memory-efficient convolutional algorithm:
• Get rid of the workspace: no-WS incurred algorithm.
• E.g.: Implicit GEMM in cuDNN

• Implicit GEMM operates natively on the convolution input tensors, converting the computation into a matrix multiply on the fly. It is important to note that corresponding matrices are never created in memory. Thus, to calculate arithmetic intensity, one can use the original tensor sizes.
• But, it is bad for the performance.
VDNN’s goals

1. Virtualize the memory usage of DNNs
2. Using both GPU and CPU memory
3. Minimizing its impact on performance
4. Transparent to programmer
5. Concentrate on the feature extraction layers
Design principles

- Sliding-window based, layer-wise memory management strategy
- Runtime memory manager *conservatively* allocates memory from its memory pool.
- Useless intermediate data structures will be released.

Memory-wise, to store all the input data in the system is not economic, although they can be reused.
Forward propagation

Be careful! Do not kill useful distant data!
Back propagation

- vDNN immediately frees up a layer’s Y and dY once this layer’s backward computation is complete
- X and dX are not released as the preceding layer’s backward propagation will be needing these values for gradient derivation
Implementation

- Based upon cuDNN.
- Use two separate CUDA streams:
  - $stream_{\text{compute}}$
  - $stream_{\text{memory}}$

Three key components:
1. Allocation/release
2. Offload
3. Prefetch
Memory allocation/release

- CUDA only has synchronous de/allocations
- Will increase communication overhead

Instead, use an asynchronous API provided by NVIDIA
Non-blocking memory transfer of X to the pinned memory via PCIe
• Much faster than disk searching; slower than direct memory reading
• Purpose: save GPU memory use.

Pinned memory: faster than unpinned memory.
Offload – for each layer

- Convolutional and pool layers’ input feature maps are read-only data structures
- Overlapping a layer’s offload operation with the same layers’ forward propagation is okay.
- Activation layers only use Y and dY, so do not need to offload
Prefetch

- Prefetching offloaded input feature maps (Xs) back to GPU memory
- Unlike offload, need to fetch the input data of layer $n$ when the backward propagation of layer $m$: $n < m$
- If $n$ and $m$'s difference is too large, benefit will be less;
- Data of layer $n-1$ needs to be ready before layer $n$’s backward propagation is over: **force synchronization**
Simultaneously
vDNN Memory Transfer Policy

Optimization problem
Subject to:
1. GPU memory capacity;
2. Overall layer-wise memory usage;

The network should be able to be trained while saving memory, but the performance should not be too bad.
Performance-wise consideration

Where did the performance loss come from?
1. Additional latency from offload and prefetch;
2. The memory-efficient algorithm per se, i.e. implicit GEMM in cuDNN for this paper.

So, need to optimize the problem using a heuristic-based memory transfer policies.
Static vDNN

Fact: 70 – 80% time is spent on the convolutional layers.

1. Offload all input features of all layers: $vDNN_{all}$
2. Only offload all convolutional layers. $vDNN_{conv}$

Simple, but maybe too simple.
Dynamic vDNN

NVIDIA’s cuDNN runtime API: initial profiling stage;

Dynamic vDNN: additional profiling process:

1. Test $vDN N_{all}$ with least memory usage to determine the feasibility
2. Test “normal” $vDN N_{all}$ and $vDN N_{conv}$, if successful, proceed training with the setting.
3. Or else, use a greedy algorithm to locally reduce memory usage. Basically: try the fastest algorithm to see whether that will overflow.

$vDN N_{conv} => vDN N_{all} => vDN N_{all}$ with memory-optimal, no-workspace algorithm
GPU node setting

NVIDIA’s Titan X
- highest math throughput (single precision throughput of 7 TFLOPS)
- memory bandwidth (max 336 GB/sec)
- memory capacity (12 GB)

Intel i7- 5930K (containing 64 GB of DDR4 memory) via a PCIe switch (gen3), which provides a maximum 16 GB/sec data transfer bandwidth
Benchmarks

• AlexNet
• OverFeat
• GoogLeNet
• Three different batch sizes for VGG-16 (the deepest network with 16 CONV and 3 FC layers)

Very deep networks:
• Series of VGG: 16 CONV layers to 416 CONV layers (16/116/216/316/416)
Legend:

(m): memory-optimal
(p): performance-optimal

all: $vDNN_{all}$
conv: $vDNN_{conv}$
dyn: dynamic vDNN
base: baseline
The smaller the average memory usage becomes, the more likely vDNN will have headroom to improve performance, by:

1. Employing performance-efficient convolutional algorithms that require larger workspace
2. Reducing the total number of offload layers and prevent potential performance drops due to offloading
GPU Memory Usage – Quick facts

• Baseline: do not optimize, so cannot train VGG16 with batch size = 128 and 256

• $vDNN_{all}$: smallest memory usage, higher traffic

• $vDNN_{conv}$: more memory usage saving, less traffic, even with performance-optimal setting.

• $vDNN_{dyn}$: balance memory and performance, but reduce memory less: actually better.
Impact on memory system
Performance (normalized to baseline)

Legend:

(m): memory-optimal
(p): performance-optimal

all: $vDNN_{all}$
conv: $vDNN_{conv}$
dyn: dynamic vDNN
base: baseline
Power

• Effect of $vDNN_{dyn}$ on overall GPU power consumption:
  • Incurs 1% to 7% maximum power overheads
  • Average power consumption (energy/time) is rarely affected because of the following two factors:

1) $vDNN_{dyn}$ does not incur any noticeable performance overhead for these five networks

2) The studied DNNs rarely saturate the peak DRAM bandwidth
On very deep networks

vDNN’s scalability

- $vDNN_{dyn}$ also did not incur any noticeable performance degradations
- the offload and prefetch latency is completely hidden inside the layer’s DNN computations
- still being able to employ the performance-optimal algorithms across the network.
Conclusion

• A scalable, memory-efficient runtime memory manager
• Reduces the average GPU memory usage
• Balance memory saving and performance