Machine Learning Tools in Action

Gennady Pekhimenko, Assistant Professor

EcoSystem Group
Performance bottlenecks in DNN Training

Tools

Analysis & Optimizations

Key performance metrics

Diverse benchmark suite with state-of-the-art models
Performance bottlenecks in DNN Training

Tools

Diverse benchmark suite with state-of-the-art models

Key performance metrics

Analysis & Optimizations
Memory Profiler (BERT)

Feature maps are still more important than weights for memory consumption
New Generation of Debugging/Prediction Tools

• **Daydream**: Accurately Estimating the Efficacy of Performance Optimizations for DNN Training (**USENIX ATC’20**)

• **Skyline**: Interactive In-editor Performance Visualizations and Debugging for DNN Training (**UIST’20**)

• **Habitat**: Prediction-guided Hardware Selection for Deep Neural Network Training (**USENIX ATC’21**)

PyTorch
Interactive In-editor Performance Visualizations and Debugging for DNN Training

Geoffrey X. Yu, Tovi Grossman, Gennady Pekhimenko
Tired of not knowing why your model is slow and/or uses up so much memory?
Tired of not knowing why your model is slow and/or uses up so much memory?

**Intersperse torch.cuda.synchronize() liberally** when debugging cuda code, to see where the bottlenecks actually are.

Does anyone have any detailed tips, walkthrus, or tutorials on how to profile PyTorch code running on the GPU?

**Looking to speed it up.**

I have a lot of narrow/chunk/cat in the model. Could this be a factor?

A bit of code running very slow? I have a lot of training set, it is tiny code, I found the loss.backward, both score and target a dynamic attention...
Skyline: Interactive In-editor Performance Debugging

- Key performance metrics (throughput, memory usage)
- Iteration run time and memory footprint breakdowns
- Interactive visualizations linked to batch size predictions
- Live and proactive performance debugging during development
Skyline: Interactive In-editor Performance Debugging

- Key performance metrics (throughput, memory usage)
- Iteration run time and memory footprint breakdowns
- Interactive visualizations linked to batch size predictions
- Live and proactive performance debugging during development
Skyline: Interactive In-editor Performance Debugging

- Key performance metrics (throughput, memory usage)
- Iteration run time and memory footprint breakdowns
- Interactive visualizations linked to batch size predictions
- Live and proactive performance debugging during development
Skyline: Interactive In-editor Performance Debugging

- Key performance metrics (throughput, memory usage)
- Iteration run time and memory footprint breakdowns
- Interactive visualizations linked to batch size predictions
- Live and proactive performance debugging during development
Skyline: Interactive In-editor Performance Debugging

- Key performance metrics (throughput, memory usage)
- Iteration run time and memory footprint breakdowns
- Interactive visualizations linked to batch size predictions
- Live and proactive performance debugging during development
Interactive visualizations tied to the code!
Habitat
A Runtime-Based Computational Performance Predictor for Deep Neural Network Training

Geoffrey X. Yu, Yubo Gao, Pavel Golikov, Gennady Pekhimenko

Get started: github.com/geoffxy/habitat
What this work is about

The problem:

• Many GPUs available for deep neural network (DNN) training
  • Each has a different ✈️ cost and 🔥 performance
  • Which should a user choose for training?

Key observations:

• DNN training computation is 🔄 highly repetitive
• Predict a GPU’s training performance by 🌋 predicting the execution time of a single iteration

Habitat is open source: github.com/geoffxy/habitat
What this work is about

The problem:

• Many GPUs available for deep neural network (DNN) training
  • Each has a different 💰 cost and 🔥 performance
  • Which should a user choose for training?

Our work:

• Use an existing GPU to🔮 predict execution times on a different GPU using ≈ wave scaling and 📈 pre-trained multilayer perceptrons (MLPs)
• Implement ideas in a new tool called✨ Habitat (open source, supports PyTorch)
• Show🔍 two case studies where Habitat leads users to the✅ correct GPU choice
Deep neural networks (DNNs) are everywhere

Image Classification
- ResNet [CVPR’16]
- VGG [ICLR’15]
- AlexNet [NeurIPS’12]

Machine Translation
- Transformer [NeurIPS’17]
- Seq2Seq NMT [NeurIPS’14]

Object Detection
- YOLO [CVPR’16]
- SSD [ECCV’16]
- Fast R-CNN [ICCV’15]

Speech Recognition
- Deep Speech 2 [ICML’16]
- End-to-End w/ RNNs [ICML’14]

But they are often computationally expensive to train!
A Cambrian explosion in hardware for training

GPUs (workstation, cloud):
- A100
- V100
- P100
- T4
- 2080Ti
- 2070
- P4000
- TITAN V
- 3090

TPUs (v2, v3, v4):

Other Emerging Accelerators:
- Cerebras WSE
- Habana Gaudi
- AWS Trainium

Which accelerator should you use?
A Cambrian explosion in hardware for training

GPUs (workstation, cloud)
- A100
- V100
- P100
- T4
- 2080Ti
- 2070
- P4000
- TITAN V
- 3090

TPUs (v2, v3, v4)

Other Emerging Accelerators
- Cerebras WSE
- Habana Gaudi
- AWS Trainium

Which GPU should you use?
Choosing a GPU: The paradox of choice

No one-size-fits-all choice. The “correct” choice depends on the user’s needs!
Why not just...

- Measure directly?
  - 💰 Need to pay to access the GPU(s)
  - 😴 Tedious to repeat for many models

- Use existing benchmarking results?
  - ⚠ Not available for all models / GPUs

- Use simple heuristics?
  - ❌ Do not always work

Simple heuristics can lead to high (> 43%) prediction errors!
Key observations

- Deep learning users may already have an existing GPU
- DNN training is a repetitive process (short training iterations)
- Use existing GPU to make iteration execution time predictions for other GPUs

Make measurements on the 2080 Ti …

This work

… to predict execution time on the A100, V100, etc.
Habitat: A runtime-based performance predictor

1. **Profile** all operations in a training iteration on an **existing GPU**
2. Predict each operation using **wave scaling** or a **multilayer perceptron (MLP)**
3. Add predictions together to get an **iteration execution time prediction**

Habitat is an open source Python library; it supports PyTorch 1.4.0

Habitat is open source: [github.com/geoffxy/habitat](https://github.com/geoffxy/habitat)
How does Habitat work?
Background: GPU execution model

- **GPU kernels**: “work” divided into thread blocks (same code, different data)

- Streaming multiprocessors (SMs) run a **finite number** of blocks concurrently

- Blocks **round-robin scheduled** onto SMs

- GPU kernels execute in **“waves”** of thread blocks
Wave scaling

GPU A — 2 SMs

- GPU Kernel
- 8 Thread Blocks

GPU B — 4 SMs

- SM 1
- SM 2
- SM 3
- SM 4

Kernel execution time on GPU A: 200 µs

Kernel execution time on GPU B: ?

Kernel Execution Time
Wave scaling

GPU A — 2 SMs
- SM 1
- SM 2

GPU B — 4 SMs
- SM 1
- SM 2
- SM 3
- SM 4

Kernel execution time on GPU A: 200 µs

Kernel execution time on GPU B: ?
Wave scaling

GPU A — 2 SMs

SM 1

SM 2

GPU B — 4 SMs

SM 1

SM 2

SM 3

SM 4

Scaling Factors

- Memory bandwidth
- Wave size
- Clock frequency

Wave execution time on GPU A

100 µs

Wave execution time on GPU B

?
Habitat is open source: github.com/geoffxy/habitat

Wave scaling

GPU A — 2 SMs
SM 1
SM 2

GPU B — 4 SMs
SM 1
SM 2
SM 3
SM 4

Wave execution time on GPU A
8 Thread Blocks

100 µs

Scaling Factors
- Memory bandwidth
- Wave size
- Clock frequency

Wave execution time on GPU B

Wave Execution Time
Wave scaling

**GPU A — 2 SMs**
- SM 1
- SM 2

**GPU B — 4 SMs**
- SM 1
- SM 2
- SM 3
- SM 4

Wave execution time on GPU A
- 100 µs

**GPU Kernel**
- 8 Thread Blocks

**Scaling Factors**
- Memory bandwidth
- Wave size
- Clock frequency
Wave scaling

GPU A — 2 SMs

GPU B — 4 SMs

Wave execution time on GPU A: 100 µs

Scaling Factors
- Memory bandwidth
- Wave size
- Clock frequency

Habitat is open source: github.com/geoffxy/habitat
Wave scaling

GPU A — 2 SMs

SM 1

SM 2

GPU B — 4 SMs

SM 1

SM 2

SM 3

SM 4

Wave execution time on GPU A

100 µs

Wave execution time on GPU B

? 

Scaling Factors

- Memory bandwidth
- Wave size
- Clock frequency

Habitat is open source: [github.com/geoffxy/habitat](https://github.com/geoffxy/habitat)
Wave scaling

**Scaling Factors**
- Memory bandwidth
- Wave size
- Clock frequency

**GPU A — 2 SMs**
- SM 1
- SM 2

**GPU B — 4 SMs**
- SM 1
- SM 2
- SM 3
- SM 4

**Wave execution time on GPU A**
- 100 µs

**Wave Execution Time**

Habitat is open source: [github.com/geoffxy/habitat](http://github.com/geoffxy/habitat)
Wave scaling

GPU A — 2 SMs
- SM 1
- SM 2

GPU B — 4 SMs
- SM 1
- SM 2
- SM 3
- SM 4

Wave execution time on GPU A: 100 µs

Scaling Factors
- Memory bandwidth
- Wave size
- Clock frequency

Wave Execution Time

Habitat is open source: github.com/geoffxy/habitat
Wave scaling

GPU A — 2 SMs

SM 1

SM 2

GPU B — 4 SMs

SM 1

SM 2

SM 3

SM 4

Wave execution time on GPU A

100 µs

Wave execution time on GPU B

?

Scaling Factors

- Memory bandwidth
- Wave size
- Clock frequency

Wave Execution Time

Habitat is open source: github.com/geoffxy/habitat
Wave scaling

Scaling Factors
- Memory bandwidth
- Wave size
- Clock frequency

GPU A — 2 SMs
- SM 1
- SM 2

GPU B — 4 SMs
- SM 1
- SM 2
- SM 3
- SM 4

Wave execution time on GPU A
100 µs

Habitat is open source: github.com/geoffxy/habitat
Wave scaling

Wave execution time on GPU A: 100 µs

Scaling Factors:
- Memory bandwidth
- Wave size
- Clock frequency

Habitat is open source: github.com/geoffxy/habitat
Wave scaling

GPU A — 2 SMs
- SM 1
- SM 2

GPU B — 4 SMs
- SM 1
- SM 2
- SM 3
- SM 4

Wave execution time on GPU A: 100 µs

Wave execution time on GPU B: ?

Scaling Factors
- Memory bandwidth
- Wave size
- Clock frequency

Wave Execution Time

Habitat is open source: github.com/geoffxy/habitat
Wave scaling

GPU A — 2 SMs
- SM 1
- SM 2

GPU B — 4 SMs
- SM 1
- SM 2
- SM 3
- SM 4

Wave execution time on GPU A
- 100 μs

Scaling Factors
- Memory bandwidth
- Wave size
- Clock frequency

Habitat is open source: github.com/geoffxy/habitat
Wave scaling

Scaling Factors
- Memory bandwidth
- Wave size
- Clock frequency

Habitat is open source: github.com/geoffxy/habitat
Wave scaling

GPU A — 2 SMs
- SM 1
- SM 2

GPU B — 4 SMs
- SM 1
- SM 2
- SM 3
- SM 4

Wave execution time on GPU A: 100 µs
Wave execution time on GPU B: 100 µs

Wave Execution Time

Scaling Factors
- 🔄 Memory bandwidth
- 🌊 Wave size
- ⏰ Clock frequency

Habitat is open source: github.com/geoffxy/habitat
Wave scaling

GPU A — 2 SMs

SM 1

SM 2

GPU B — 4 SMs

SM 1

SM 2

SM 3

SM 4

Kernel execution time on GPU A

200 µs

Kernel execution time on GPU B

100 µs

Scaling Factors

- Memory bandwidth
- Wave size
- Clock frequency

Habitat is open source: github.com/geoffxy/habitat
One last wrinkle: Kernel-varying operations

- 🌊 Wave scaling assumes the same kernel is used across GPUs

- ⚠️ A few DNN operations use architecture-specific kernels (“kernel-varying”)
  - Convolutions, linear (dense) layers, LSTMs

-💡 Habitat uses pre-trained multilayer perceptrons (MLP) for these operations
Evaluation

• How **accurate** are Habitat’s predictions?

• Does using Habitat lead to making “correct” decisions?

• Six GPUs (spanning three generations):
  - P4000
  - P100
  - V100
  - 2070
  - 2080Ti
  - T4

• Five models:
  - ResNet-50
  - Inception v3
  - GNMT
  - Transformer
  - DCGAN

• PyTorch 1.4.0

Habitat is open source: [github.com/geoffxy/habitat](http://github.com/geoffxy/habitat)
How accurate is Habitat?

- Predict iteration execution time (GPU, model, batch size)
How accurate is Habitat?

• Predict iteration execution time (GPU, model, batch size)

2070  ➔  V100

ResNet-50
Batch Size: 32

ResNet-50
Batch Size: 32

🔮 Predicted: 96 ms
⏱ Actual: 94 ms
✨ Error: 1.9%
How accurate is Habitat?

Habitat makes **accurate** predictions, with an average error of **11.8%** across all configurations (30 GPU pairs x 5 models x 3 batch sizes).
Does Habitat lead to the “correct” decision?
Rent a GPU in the cloud?

**Scenario:** Want to train **GNMT**, have access to a **P4000**. Which cloud GPU to use, if any?
Rent a GPU in the cloud?

Scenario: Want to train GNMT, have access to a P4000. Which cloud GPU to use, if any?

GNMT Throughput Normalized to the P4000

Measured | Predicted
---|---
19.0% | 14.9%
8.0% | 13.3%
1.4% | 0.4%
2.0% |
33.8% |
3.5% |

GPU and Batch Size

Habitat is open source: github.com/geoffsy/habitat
Rent a GPU in the cloud?

Scenario: Want to train GNMT, have access to a P4000. Which cloud GPU to use, if any?

Habitat correctly predicts that the V100 is the best choice for performance.
Rent a GPU in the cloud?

Scenario: Want to train GNMT, have access to a P4000. Which cloud GPU to use, if any?

Habitat correctly predicts that the T4 (or P4000) is the best choice for cost.
Always use the “best” GPU?

**Scenario:** Want to train DCGAN, have access to a 2080 Ti. Use the V100?
Always use the “best” GPU?

Scenario: Want to train DCGAN, have access to a 2080 Ti. Use the V100?

Habitat correctly predicts that the V100 only offers a marginal improvement (1.1x) over the 2080 Ti.

Habitat is open source: github.com/geoffxy/habitat
More details and results in the paper

- **🔍 Prediction breakdowns**
- **👥 MLP sensitivity study**
- **🔮 Predictions onto additional GPUs**
- **…” Discussion about extensibility**
- **🖥 Distributed training**
- **🔍 Mixed precision training**
- **💬 Other types of hardware**

Habitat is open source: github.com/geoffxy/habitat

---

**Abstract**

Deep learning researchers and practitioners usually leverage GPUs to help train their deep neural networks (DNNs) fast. However, choosing which GPU to use is challenging because (i) there are many options, and (ii) users grapple with competing concerns: maximizing compute performance while minimizing costs. In this work, we present a new practical technique to help users make informed and cost-efficient GPU selections: make performance predictions with the help of a GPU that the user already owns. Our technique exploits the observation that, because DNN training consists of repetitive compute steps, predicting the execution time of a single iteration is usually enough to characterize the performance of an entire training process. We make predictions by scaling the execution time of each operation in a training iteration from one GPU to another using either (i) wave scaling, a technique based on a GPU’s execution model, or (ii) pre-trained multiplier perceptions. We implement our technique into a Python library called Habitat and find that it makes accurate time predictions (with an average error of 11.6%) on ResNet-50, Inception v3, the Transformer, GNMT, and DCGAN across six different GPU architectures. Habitat supports PyTorch, is easy to use, and is open source.1

1 Introduction

Over the past decade, deep neural networks (DNNs) have seen incredible success across many machine learning tasks [26, 37, 39, 51, 93, 96, 99]—leading them to become widely used throughout academia and industry. However, despite their popularity, DNNs are not always straightforward to use in practice because they can be extremely computationally expensive to train [23, 53, 95, 109]. This is why, over the past few years, there has been a significant and ongoing effort to bring hardware acceleration to DNN training [10, 16, 31, 36, 45, 78, 80].

Habitat: A Runtime-Based Computational Performance Predictor for Deep Neural Network Training

Geoffrey X. Yu  
University of Toronto  
Vector Institute

Yubo Gao  
University of Toronto  
Vector Institute

Pavel Golikov  
University of Toronto  
Vector Institute

Genady Pohhizensko  
University of Toronto  
Vector Institute

training. These options range from desktop and server-class GPUs (e.g., 2080 Ti [70] and A100 [73]) all the way to specialized accelerators such as the TPV [17], AWS Trainium [10], Gaudi [36], GPU [72], and Cerberus WSE [16]. Having all these options offers flexibility to users, but at the same time can also lead to a paradox of choice: which hardware option should a researcher or practitioner use to train their DNNs?

A natural way to start answering this question is to first consider CUDA-enabled GPUs. This is because (i) they are commonly used in deep learning; (ii) are supported by all major deep learning software frameworks (PyTorch [86], Tensorflow [1], and MXNet [19]); (iii) have mature tooling support (e.g., CUPiT [76]); and (iv) are readily available for rent and purchase. In particular, when considering GPUs, we find that that there are many situations where a deep learning user needs to choose a specific GPU to use for training:

- **Choosing between different hardware tiers**. In both academia and industry, deep learning users often have access to several tiers of hardware: (i) a workstation with a GPU used for development (e.g., 2080 Ti), (ii) a private GPU cluster that is shared within their organization (e.g., RTX6000 [84]), and (iii) GPUs that you can rent in the cloud (e.g., V100 [63]). Each tier offers a different cost, availability, and performance trade-off. For example, a private cluster might be “free” (in monetary costs) to use, but jobs may be queued because the cluster is also shared among other users. In contrast, cloud GPUs can be rented on-demand for exclusive use.

- **Deciding on which GPU to rent or purchase**. Cloud providers make many different GPUs available for rent (e.g., P100 [62], V100, T4 [71], and A100 [73]), each with different performance at different prices. Similarly, a wide variety of GPUs are available for purchase (e.g., 2080 Ti, 3090 [82]) both individually and as a part of pre-built work-
Key takeaways

- 🔁 DNN computation is special (*repetitive*), enabling new analysis opportunities.

- ⏰ Use *runtime-based information* to make iteration execution time predictions.

- ✅ Habitat leads to the *correct decision* in the case studies.

- 🏞 The hardware landscape is growing; users need help *choosing effectively*!

Habitat is open source: [github.com/geoffxy/habitat](https://github.com/geoffxy/habitat)
Potential Integration with Skyline

- Habitat can provide performance predictions for **different GPUs**
- **Proactive** hardware selection
- Opportunity to surface more content from an **educational** standpoint
Daydream: Accurately Estimating the Efficacy of Optimizations for DNN Training

Hongyu Zhu\textsubscript{1,2}, Amar Phanishayee\textsubscript{3}, Gennady Pekhimenko\textsubscript{1,2}
Executive Summary

• Benefits of proposed DNN optimizations are not fully exploited because
  • Efficacy varies for different HW/SW configurations
  • It is onerous to implement optimizations

• **Daydream** efficiently explores the efficacy of various DNN optimizations using **Dependency graph analysis**:
  • Tracking dependencies at the abstraction of GPU kernels
  • Kernel-to-layer mapping
  • Transformation rules to model a diverse set of optimizations

• **Evaluation**:
  • Low estimation error on 5 optimizations, 5 DNN models across 3 applications
  • Ability to predict distributed training runtime based on single-GPU profile
Advances in ML Full Stack Research

Hard for a ML programmer to identify the efficacy of new algorithms, optimizations, and hardware improvements in their deployments.

DNN compute requirements are growing exponentially

https://openai.com/blog/ai-and-compute/

Rapid advances in algorithms, systems optimizations & hardware architectures

https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=8259424&tag=1
What-if Questions

Why is my DNN training workload running slow? What is the bottleneck?

ML Programmers

Why is my DNN training workload running slow? What is the bottleneck?
Why Dependency Analysis

Answering what-if questions in non-ML contexts

DNN Computational Graph

Making Sense of Performance in Data Analytics Frameworks (Ousterhout et al., NSDI 15)

COZ: Finding Code that Counts with Causal Profiling (Curtis et al., SOSP 15)

What-If Analysis of Page Load Time in Web Browsers Using Causal Profiling (Pourghassemi et al., SIGMETRICS 19)

AlexNet (2012)

LSTM (2014)

TensorFlow’s computational graph (2016)
Outline

• Motivation and Background

• Challenges and Design

• Evaluation
Daydream Overview

**Input**: an DNN training implementation $X$, an optimization $Y$

**Output**: the estimation of runtime when applying $Y$ to $X$
Challenges for Dependency Graph Analysis in the ML context

**Challenge #1**: Thousands of tasks, and dependency needs to be tracked across CPU threads, GPU streams, and interconnects.

**Challenge #2**: Some optimizations operate on the kernel-level granularity. Others operate on layer-level granularity. How should one correlate low-level traces with DNN topology?

**Challenge #3**: Ability to easily model diverse DNN optimizations.
Challenge 1: Tracking Dependencies

Observation: GPU kernels are highly serialized for most DNN training workloads
Daydream’s Graph Construction

We identify the following types of dependencies:

1. Sequential CPU-CPU: two consecutive CPU calls on the same CPU thread
2. Sequential GPU-GPU: two consecutive GPU kernels on the same stream
3. CPU-GPU launching: A CPU call launching a GPU kernel/CUDA memory copies
4. GPU-CPU sync: A CPU synchronization call waiting for GPU kernel to finish
5. CPU-Communication
6. CPU-CPU: e.g., thread spawn, lock, join, ...
Challenge 2: Trace-Layer Correlation

• Optimizations might operate at kernel level, yet still requiring the knowledge of DNN topology
  • E.g., Fusing CONV and RELU layers
  • Low-level traces have NO domain knowledge

• Naïve approach that adds synchronization is not performant
Daydream’s Kernel-Layer Mapping

1. Get $L_0$’s Timestamps
2. Get $L_0$’s CPU tasks
3. Map $K_0$, $K_1$ to $L_0$ according to dependencies

$K_0$, $K_1$ belong to $L_0$

Little overhead (only need to instrument frameworks for per-layer timestamps)
No alternation to the dependency graph (synchronization-free)
## Challenge 3: Optimization Diversity

<table>
<thead>
<tr>
<th>Optimization Goals</th>
<th>Strategy</th>
<th>Technique Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Improving Hardware Utilization in Single-Worker Environment</td>
<td>Increasing Mini-batch Size by Reducing Memory Footprints</td>
<td><strong>vDNN</strong> (MICRO16), <strong>Gist</strong> (ISCA18), Echo (ISCA20)</td>
</tr>
<tr>
<td></td>
<td>Reducing Precision</td>
<td><strong>Automatic Mixed Precision</strong> (arxiv17)</td>
</tr>
<tr>
<td></td>
<td>Kernel/Layer Fusion</td>
<td><strong>FusedAdam</strong>, <strong>MetaFlow</strong> (MLSys19), TASO (SOSP19)</td>
</tr>
<tr>
<td></td>
<td>Improving Kernel Implementation</td>
<td><strong>Restructuring Batchnorm</strong> (MLSys19), TVM (OSDI18), Tensor Comprehensions (arxiv18)</td>
</tr>
<tr>
<td>Lowering Communication Overhead in Distributed Training</td>
<td>Reducing Communication Workloads</td>
<td><strong>Deep Gradient Compression</strong> (ICLR18), QSGD (NeurIPS17), AdaComm (MLSys19), Parallax (EuroSys19), TernGrad (NeurIPS17)</td>
</tr>
<tr>
<td></td>
<td>Improving Communication Efficiency/Overlap</td>
<td><strong>Wait-free Backprop</strong> (ATC17), <strong>P3</strong> (MLSys19), <strong>BlueConnect</strong> (MLSys19), TicTac (MLSys19), BytePS (SOSP19), Blink (MLSys19)</td>
</tr>
</tbody>
</table>

We evaluate “optimizations”, and show that we can easily model “optimizations”, using Daydream
Daydream’s Transformation Rules

Most DNN optimizations can be described as a combination of the following rules:

1. Select(expr): return tasks of interests for further process
2. Insert(s, task, t):
3. Delete(s, task, t):
4. Schedule(Q): --> task
   return one task from a queue of tasks that are ready to execute

```plaintext
Select(taskPtr(isCONV(t)))
```
Example – Automatic Mixed Precision

Using Daydream to estimate the efficacy of AMP (refer to our paper for more examples)

```python
def Estimate AMP(cupti_file, timestamps_file):
    records = read_cupti_trace(cupti_file)['traceEvents']
    timestamps = read_timestamps(timestamps_file)
    graph = Graph(records, timestamps)

    for node in graph.nodes:
        if node.kind == "KERNEL":
            # node is a GPU kernel
            if "volta" in node.name or "wgrad" in node.name or "sgemm" in node.name:
                # using TensorCore
                node.dur /= 3
            else:
                # using FP16 cores
                node.dur /= 2

    return graph.simulate()
```

CUPTI records

Timestamps of DNN layers

Loop over GPU kernels

If using TensorCore

Simulate the graph to predict runtime

10 optimization examples, each around 20 lines of code (refer to our paper)
Outline

• Motivation and Background

• Key Ideas and Design

• Evaluation
  • Automatic Mixed Precision, FusedAdam, Reconstructing Batchnorm
  • Distributed Training, P3
Experimental Setup

Hardware:
- RTX 2080 Ti

Software:
- PyTorch v1.0
- mxnet v1.1
- Caffe v1.0
- CUDA v10.0
- cuDNN v7.4.2
- NCCL v2.4.2
Methodology

We evaluate Daydream using the following workloads:

<table>
<thead>
<tr>
<th>Application</th>
<th>Model</th>
<th>Dataset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image Classification</td>
<td>VGG-19</td>
<td>Imagenet</td>
</tr>
<tr>
<td></td>
<td>DenseNet-121</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ResNet-50</td>
<td></td>
</tr>
<tr>
<td>Machine Translation</td>
<td>GNMT (Seq2Seq)</td>
<td>WMT</td>
</tr>
<tr>
<td>Language Modeling</td>
<td>BERT</td>
<td>SQuAD</td>
</tr>
</tbody>
</table>

We evaluate the followings for each optimization $X$ on benchmark $Y$:

- Baseline: the iteration time of running $Y$ without $X$
- Ground Truth: the iteration time of running $Y$ with $X$
- Prediction: Daydream’s prediction of modeling $X$ using dependency graph from $Y$
Runtime Estimation Accuracy

Estimating Automatic Mixed Precision (AMP), FusedAdam, and Restructuring Batchnorm (RB)

Daydream achieves 8% prediction error on average (15% maximum)
Estimating Distributed Training

Daydream can accurately estimate the performance for various system configurations.
Daydream can accurately estimate the performance for multiple DNN models.
Estimating Efficacy of P3

Prediction accuracy for Priority-Based Parameter Propagation (P3)

Using Daydream, we can successfully estimate whether P3 would provide significant or subtle improvement.
Conclusion

• To our knowledge, Daydream is the first system that aims at estimating efficacy of optimizations for DNN training

• Daydream uses:
  • Dependency graph analysis based on the kernel-level granularity
  • Sync-free kernel-to-layer mapping
  • Graph transformation rules

• Daydream is able to accurately estimate the efficacy of optimizations across a wide range of DNN optimizations
RL-Scope: Cross-Stack Profiling for Reinforcement Learning

RL-Scope is an open-source tool: https://github.com/UofT-EcoSystem/rlscope
Deep reinforcement learning (RL) progress

- The past decade has seen promising advancements in RL:

  - Atari from RGB pixels\(^1\)
  - Expert level play at Go\(^2\)
  - Robotic locomotion\(^3\)
  - Datacenter management\(^4\)

Deep reinforcement learning (RL) progress

**PROBLEM (1):** *training time* limits progress!

- E.g., 40 days to train AlphaGoZero (4 TPUs)
- Many RL algorithms to try!

**PROBLEM (2):** RL workloads ≠ supervised learning workloads

<table>
<thead>
<tr>
<th>Supervised Learning</th>
<th>Reinforcement Learning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Large labelled dataset</td>
<td>Simulator</td>
</tr>
</tbody>
</table>

**PROBLEM (3):** profiling tools are designed for *GPU-bound* workloads.

- Existing tools limited to *bottleneck layers*
- In RL, *CPU* time matters too!
- Profiling overhead must be handled; ≥ 1.9× inflation!

Why not just focus on supervised learning instead, and let RL indirectly benefit?

RL workloads have fundamentally different workload structure!
Supervised Learning, Reinforcement Learning...
What’s the difference?

The key differences in RL:
1. Training data
2. Training loop
3. Neural network size

Where does an RL workload spend its time?
High-level algorithmic operations

Inference

Simulation

Backpropagation

RL algorithm (e.g. DQN):

1. Initialize replay memory $D$ to capacity $N$.
2. Initialize action-value function $Q$ with random weights $\theta$.
3. Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$.

For episode $= 1, M$ do

1. Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$.
2. For $t = 1, T$ do
   3. With probability $\epsilon$ select a random action $a_t$; otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$.
   4. Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$.
   5. Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$.
   6. Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $D$.
   7. Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $D$.
   8. Set $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$.
   9. Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameters $\theta$.
   10. Every $C$ steps reset $\hat{Q} = Q$.

End For

End For
RL-Scope: Cross-stack RL Profiler

RL-Scope profiler features:
- Cross-stack scoping
  Cross-stack view of where CPU/GPU time is spent.
- Cross-framework
  Works with TensorFlow, PyTorch, and multiple simulators.
- Corrects for profiling overhead
  Correct CPU overhead for accurate insights.

Contributions:
- RL-Scope profiler
  Open-source tool; profile your own RL workloads!
- RL workload survey
  How CPU/GPU time varies across algorithms and simulators.
# RL-Scope profiler

(1) Developer annotations

```python
# ML script training loop
for t in range(num_timesteps):
    # simulation code
    with rls.operation("simulation"):
        # simulation code
    # inference code
    with rls.operation("inference"):
        # inference code
    # backprop code
    with rls.operation("backprop"):
        # backprop code
```

Understandable output
Developer annotations help users tie profiler output to high-level code

Simulation
Inference
(2) Transparent interception

Supports TensorFlow and PyTorch
Transparent interception ⇒ no recompilation.
Easy to support multiple DL frameworks/simulators.
**RL-Scope profiler**

(3) Overhead calibration and correction

RL-Scope corrects for CPU profiling overhead
Allows for accurate insights

Collection disabled

Collection enabled

= Average overhead

4 collection events
RL-Scope profiler

1. Resource overlap: CPU/GPU overlap

2. Scoping: scope CPU/GPU time to high-level algorithmic operations

(4) Cross-stack event overlap
RL Workload Survey

3 workload dimensions to survey:

(1) DL backend:
- PyTorch
- TensorFlow

(2) RL algorithm:
- On-policy: $\pi(a|s)$
- Off-policy: $V(S), Q(s,a)$

(3) Simulator:

- Simulator complexity
  - Low
    - Atari
    - Go
  - Medium
    - Grasping
    - Photo-realistic
  - High
    - Drones

Computer games
Robotics
How do different executions models (and implementations) affect RL training time?
Case study: DL backends

RL algorithm: TD3

RL algorithm: DDPG

Neither Autograph nor Graph outperforms the other and are within 19.7% of one another.

Eager execution is 1.9 × to 4.8 × slower than Autograph and Graph execution.

Autograph reduces Python time to 7.2% from 59.2% in Graph for inference/backpropagation.

RL-Scope explains AutoGraph speedup: reduced Python → Backend transitions.

RL-Scope explains PyTorch vs TensorFlow: PyTorch Eager is 2.3 × faster than TensorFlow Eager since it minimizes Python → Backend transitions more effectively.

CUDA API time dominates GPU kernel execution time, taking on average 3.6 × longer.

At most 14.1% is spent executing GPU kernels across all ML backends and execution models; the rest is entirely CPU-bound.

RL-Scope explains performance differences between ML backends by correlating transitions.

Transitions also help explain bottlenecks (e.g., over abstraction, hyperparameter settings); see paper for details!
**RL Workload Survey: Takeaway**

3 workload dimensions to survey:

1. **DL backend:**
   - PyTorch
   - TensorFlow

2. **RL algorithm:**
   - On-policy: $\pi(a|s)$
   - Off-policy: $V(S)$, $Q(s,a)$

3. **Simulator:**
   - Low
   - Medium: Grasping
   - High: Drones

All RL workloads suffer from poor GPU utilization due to RL workload structure.

- At most 14.1% executing GPU kernels.
- Backpropagation/Inference spend at most 14.1% executing GPU kernels.

At least 38.1% of training time in simulation.

Computer games: Atari, Go

Robotics: Grasping, Drones

Case studies: All RL workloads suffer from poor GPU utilization due to RL workload structure.
My Students: EcoSystem Research Group

- Hongyu Zhu (PhD)
- Bojian Zheng (PhD)
- Alexandra Tsvetkova (PhD)
- James Gleeson (PhD, co-advised)
- Anand Jayarajan (PhD)
- Mustafa Quraish (PhD)
- Shang (Sam) Wang (MSc)
- Jiacheng Yang (MASc)
- Pavel Golikov (MSc)
- Yaoyao Ding (MASc)
- Daniel Snider (MSc)
- Kevin Song (MASc)
- Yu Bo Gao (BSc)
- Kimberly Hau (BASc)
- Qingyuan Qie (BSc)
- Chenhao Jiang (BSc)
- Murali Andoorveedu (BASc)
Backup Slides
Wave scaling: In detail

\[ T_d = \left[ \frac{B}{W_d} \right] \left( \frac{C_o}{C_d} \right)^{1-\gamma} \left( \frac{D_o W_d}{D_d W_o} \right) \gamma \left[ \frac{B}{W_o} \right]^{-1} T_o \]

Gamma is set automatically using the Roofline model; see paper for more details.
How to set gamma?

Roofline model

- **Roofline**: estimates performance ceilings using arithmetic intensity
- Measure kernel’s arithmetic intensity
- Set gamma based on its **proximity** to the ridge point

\[
\gamma = \begin{cases} 
-0.5/R \times x + 1 & \text{if } x < R \\
0.5R/x & \text{otherwise}
\end{cases}
\]

\[ R = P/D \]

Ridge point

Measured intensity (Kernel-specific, GPU-agnostic)

Ridge point (GPU-specific, computable)

Habitat is open source: [github.com/geoffxy/habitat](https://github.com/geoffxy/habitat)
What are “kernel-varying” operations?

- Logical operations implemented using **different** physical kernels on **different** GPUs
  - Convolution, linear layers (matrix multiplication), LSTMs
- Exist for performance reasons, significant engineering effort
  - Corollary: Relatively few kernel-varying operations
- Habitat handles them using pre-trained MLPs
Wave scaling and MLPs

- Most operations predicted using wave scaling
  - MLPs: 4/77 (5%)
  - Wave scaling: 73/77 (95%)
- Both contribute non-trivially to predictions
- Highlights strength of Habitat:
  - Pre-trained MLPs for a few operations
  - Wave scaling handles the rest

Habitat is open source: github.com/geoffxy/habitat
Details about the MLPs?

- Treat prediction as regression problem, one MLP per operation
- **Input:** Operation configuration, GPU features (memory bandwidth, clock frequency)
- **Output:** Execution time on that GPU
- **Training data:** Collect (GPU, operation configuration, execution time) data points
- See paper for more details
Distributed training?

- Three parts: computation, communication, overlap
- Habitat can help with **computation** predictions
  - **Data parallel:** Use as-is
  - **Model parallel, pipeline parallel, others:** Manually partition model code before running Habitat
- Apply techniques from other works for communication, overlap
  - Daydream (USENIX ATC ’20)
  - Paleo (ICLR’17)
Mixed precision?

- Yes: use Habitat in conjunction with Daydream (USENIX ATC ’20)

- **Daydream**: Single precision, Same GPU ➡ Mixed precision, Same GPU

- **Habitat**: Single precision, Same GPU ➡ Single precision, Different GPU

- Run Habitat, send predictions into Daydream

- We tried ResNet-50 (2070, 2080Ti), average error: 16.1%

- **Habitat + Daydream should produce accurate cross-GPU mixed precision predictions**
Habitat vs. Paleo

**Habitat (this work)**

“Hybrid” prediction strategy: runtime-based plus MLPs

Pros:
- No need to derive a model for every single operation; can handle new operations
- Captures behavior of kernel-varying operations

Cons:
- Need to re-train MLPs for new GPUs

**Paleo [1]**

Analytical modeling of FLOPs plus heuristics for kernel-varying operations

Pros:
- Does not require an existing GPU

Cons:
- Derive new model for each new operation
- Heuristics cannot always capture kernel-varying behavior