S61453 – Optimizing Large-Scale Distributed GPU Training with Holistic Trace Analysis

Xizhou Feng
Software Engineer
Meta
fengx@meta.com

Yuzhen Huang
Research Scientist
Meta
yuzhenhuang@meta.com

Meta

Speakers

Xizhou Feng, PhD

Software Engineer
Meta Platforms
Monetization ML Infra Team
20+ years of experience in developing innovative HPC systems and software for AI and computational sciences applications.
fengx@meta.com

Yuzhen Huang, PhD

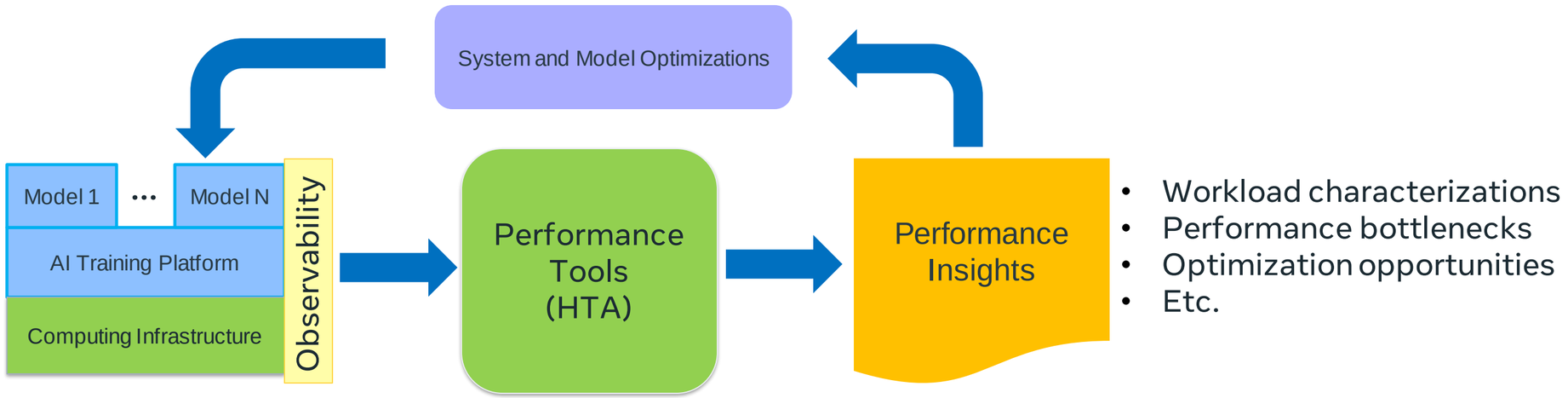
Research Scientist
Meta Platforms
Monetization ML Infra Team
TL & expert in large-scale recommendation model training systems, optimization, and performance tools.
yuzhenhuang@meta.com

Agenda

1. Introduction: AI Model Training Efficiency
2. AI Model Performance Analysis using HTA
3. Case Studies and Success Stories
4. Summary
5. Q&A

A High Level Summary

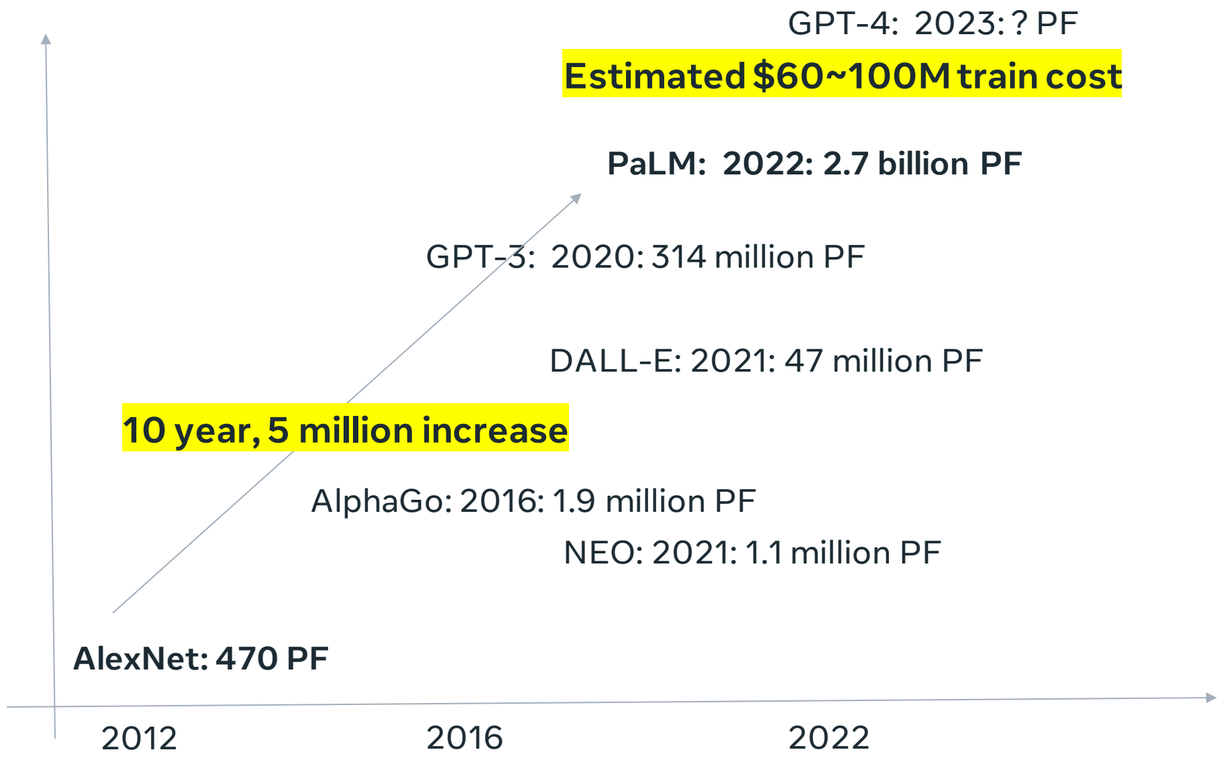
1. **Efficient training** is crucial for developing large AI models.
2. **Performance tools** enable and speed up the model optimization.
3. **HTA** aims to provide performance insights on distributed GPU training.



Why Efficient Training Matters - I?

Two primary drivers of today's AI revolution:

1. **Scale Large Neural Network**
   - Model performance depends strongly on scale
     - # of model parameters
     - training data size
     - the amount of computing spent on training
2. **Increase Computing Power**
   - 2023 H100 SXM:
     - .067 PF/2 for FP32;
     - 2.0 PF/s for BF16 Tensor Core (~29x when use lower precision)
   - Top #1 system by LINPACK performance
     - Sequoia: 2012 : 16 PF/s
     - Frontier: 2023: 1,194 PF/s (~74x in 10 years)



PF: PetaFLOP=10^{15}FLOP

Sources: Max Roser (2022) - "The brief history of artificial intelligence: The world has changed fast – what might be next?" Retrieved from: 'https://ourworldindata.org/brief-history-of-ai'.

The training cost per model grows by 3~4x every year!

Why Efficient Training Matters - II?

- **Business Impact**
  - Improved scalability
  - Reduced computational cost
  - Faster development cycle
- **Environmental Impact**
  - Lower energy consumption
  - Smaller carbon footprint
- **Other actors**
  - Capacity crunch
  - Model Accessibility

Challenges to Improve Large AI Model Training Efficiency

- **Model complexity**
  - Billions to tens of trillions of model parameters
  - Hundreds or thoughts of neural network modules
  - One platform to support many model types
- **Computing system complexity**
  - Multiple computing paradigms: parallel heterogenous systems
  - Large scale systems: 100s~10,000s CPU and GPUs
  - Efficiently moving large amount of data across and between devices
- **Complex SW/HW interactions**
  - Construct and optimize the software component to effectively utilize the hardware’s capacity
  - Cope with additional ML Ops complexity: data loading, checkpointing, observability

Effective Performance Tools Are Essential

To utilize hardware's raw capacity and unblock AI model training efficiency.

The peak performance of a cluster of NVIDIA H100 GPUs is extraordinary.

Form Factor	H100 SXM	H100 PCIe	H100 NVL[1]
FP64	34 teraFLOPS	26 teraFLOPS	68 teraFLOPs
FP64 Tensor Core	67 teraFLOPS	51 teraFLOPS	134 teraFLOPs
FP32	67 teraFLOPS	51 teraFLOPS	134 teraFLOPs
TF32 Tensor Core	989 teraFLOPS[2]	756 teraFLOPS[2]	1,979 teraFLOPs[2]
BFLOAT16 Tensor Core	1,979 teraFLOPS[2]	1,513 teraFLOPS[2]	3,958 teraFLOPs[2]
FP16 Tensor Core	1,979 teraFLOPS[2]	1,513 teraFLOPS[2]	3,958 teraFLOPs[2]
FP8 Tensor Core	3,958 teraFLOPS[2]	3,026 teraFLOPS[2]	7,916 teraFLOPs[2]
INT8 Tensor Core	3,958 TOPS[2]	3,026 TOPS[2]	7,916 TOPS[2]
GPU memory	80GB	80GB	188GB
GPU memory bandwidth	3.35TB/s	2TB/s	7.8TB/s[3]

Source: https://www.nvidia.com/en-us/data-center/h100/

However, most large AI model training workloads only achieves a fraction of the system's peak performance.

➔ We need tools to identify where the performance bottlenecks are and recommend how to overcome them.

The craftsman who wishes to do his work well must first sharpen his tools.
(工欲善其事，必先利其器)

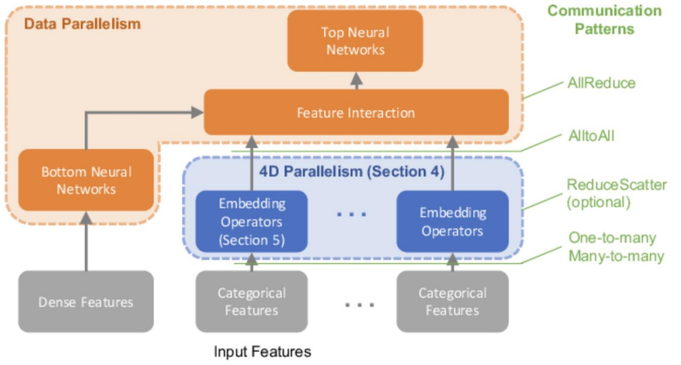
- Analects of Confucius, Book 15, Chapter 9

02 AI Model Performance Analysis using HTA

Typical Workloads of Large Distributed GPU Models Training

Many model types, each with different architecture and workload characteristics

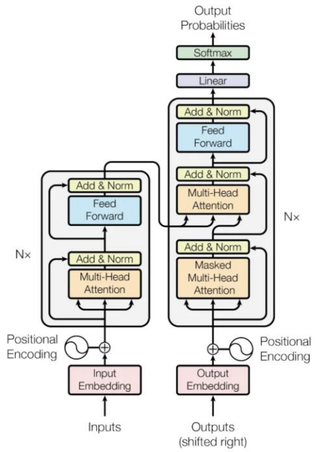
- Deep Learning based Recommendation Systems (DLRMs)
- Large Language Models (LLMs)
- Computer Visions
- Diffusion Models



The DLRM Model Architecture.
Credit: Mudigere, Dheevatsa et al. "Software-hardware co-design for fast and scalable training of deep learning recommendation models." Proceedings of the 49th Annual International Symposium on Computer Architecture (2021)

Workload Characteristics

- PyTorch-based: dev productivity v. computing efficiency
- Model scales: 100s→ 1000s → 10,000s GPUs
- Multiple types of parallelisms
- Data movement & synchronizations across devices.
  - CPU-CPU, GPU-CPU, GPU-GPU, …



The Transformer Model Architecture
Credit: Vaswani, Ashish et al. "Attention is All you Need." Neural Information Processing Systems (2017).

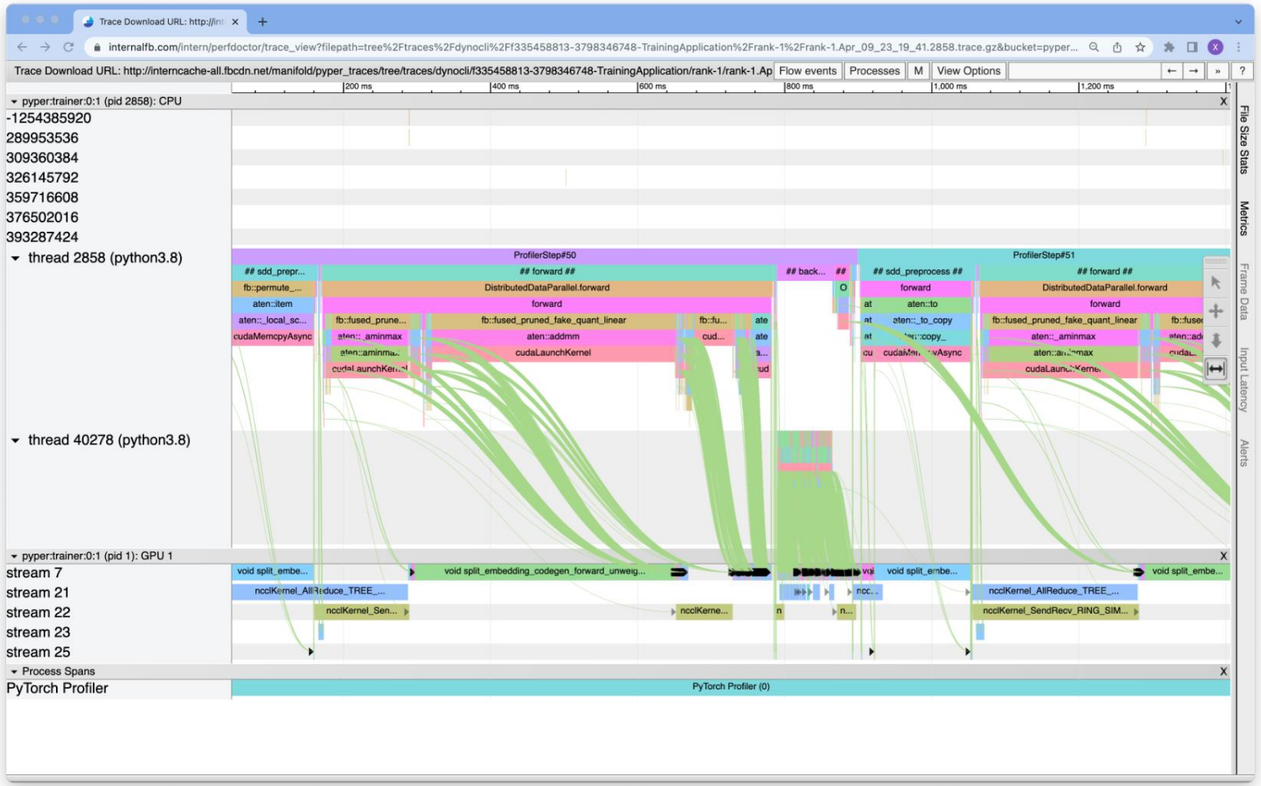
PyTorch Profiler - Profiling PyTorch Model Execution

- **PyTorch Profiler**
  - A powerful tool for collecting the execution time and memory footprint of a PyTorch model during its training and inference.
  - Reference: https://pytorch.org/docs/stable/profiler.html

```
from torch.profiler import profile, schedule, tensorboard_trace_handler

tracing_schedule = schedule(skip_first=5, wait=5, warmup=5, active=5, repeat=1)
trace_handler = tensorboard_trace_handler(dir_name=/output/folder, use_gzip=True)

with profile(
    activities = [ProfilerActivity.CPU, ProfilerActivity.GPU],
    schedule = tracing_schedule,
    on_trace_ready = trace_handler,
    record_shapes = True,
    with_stack=True
) as prof:

    for step, batch_data in enumerate(data_loader):
        train(batch_data)
        prof.step()
```

Visualize Single PyTorch Trace Using Google Trace Viewer

- Users can view collected PyTorch traces using Google's Trace Viewer plugin or the Perfetto tool.
- A trace captures the annotated and timed activities on the CPU and GPU devices.



Vertically

- Device (CPU | GPU)
- CPU Processes & Threads, and CUDA Stream

Horizontally

- Timeline
- Duration of CPU and CUDA operators

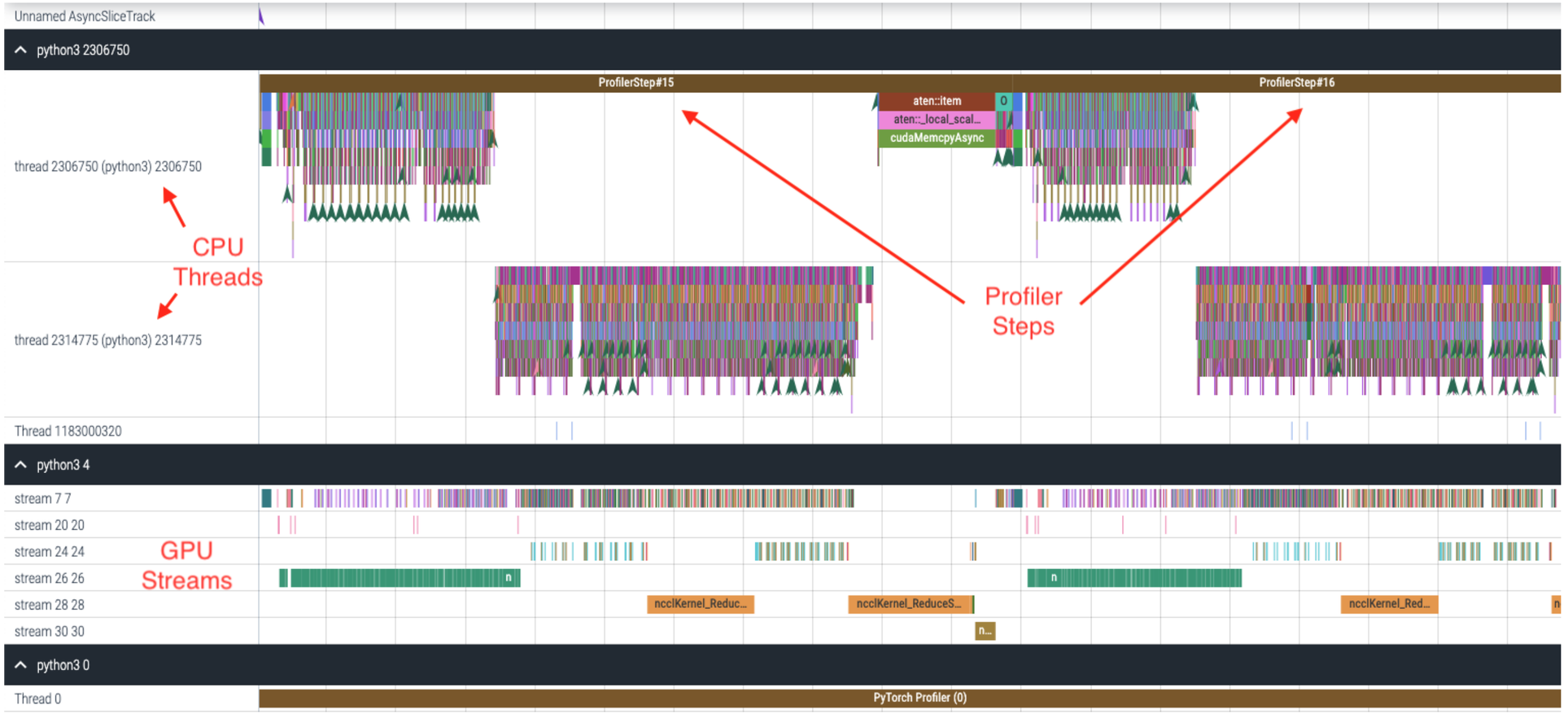
Links

- Call stack (implicitly, inferred from time intervals)
- CPU-GPU linkage (annotation like correlation IDs)

References:

- Trace Viewer: https://github.com/catapult-project/catapult/blob/master/tracing/docs/getting-started.md
- Perfetto: https://perfetto.dev/

Perfetto Is Another Tool For Analyzing PyTorch Traces



References:

- Trace Viewer: https://github.com/catapult-project/catapult/blob/master/tracing/docs/getting-started.md
- Perfetto: https://perfetto.dev/

Comparing Nsight Systems and PyTorch (Kineto) Profiler

NVIDIA Nsight Systems

- A performance analysis tool offered by NVIDIA for profiling various GPU applications.
- Provides detailed view of system resource usage like CPU, GPU, and memory usages.
- Comprehensive and powerful; steeper learning curve for AI developers.
- More heavyweight; need infra support for large-scale deployment.
- Proprietary, which can limit tooling customization and integration with our internal training stack and performance tools.

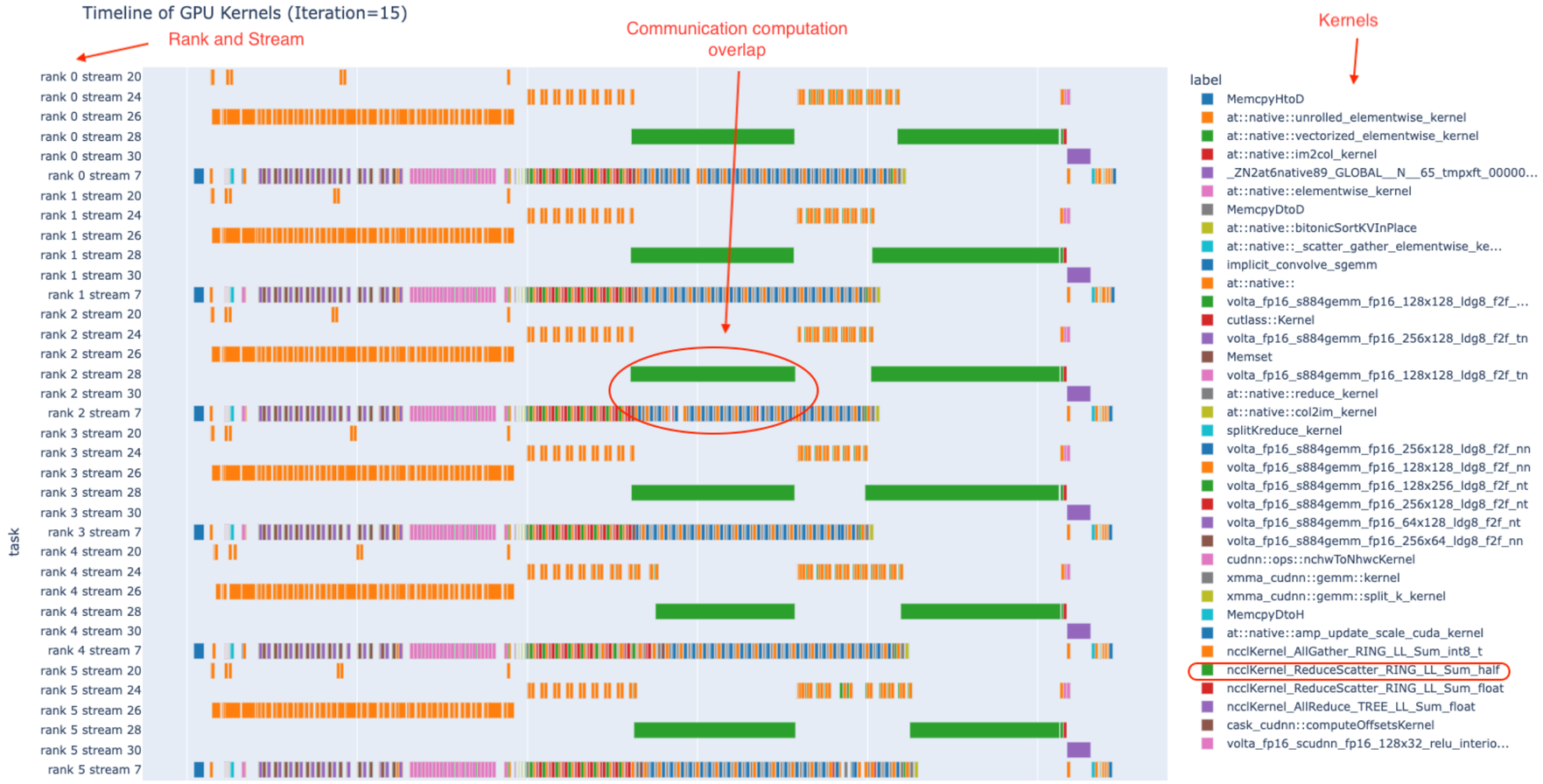
PyTorch (Kineto) Profiler

- A built-in tool specifically designed for PyTorch applications.
- Focuses on perf. events and metrics related to AI model training.
- Easy to integrate with any PyTorch models; efforts to enable advanced CUPTI features.
- Very lightweight; no additional deployment efforts are required.
- Open-source, having been integrated with existing training stacks; allowing us focus on developing trace analysis to guide model optimization.

Both Nsight and Kineto use NVIDIA CUPTI library for profiling and tracing. More advanced, low-level performance analysis and optimization features are provided by Nsight but are missing in the PyTorch profiler.

Sources: https://developer.nvidia.com/nsight-systems; https://developer.nvidia.com/cupti

Single Trace Analysis Is Inadequate for Distribute GPU Models

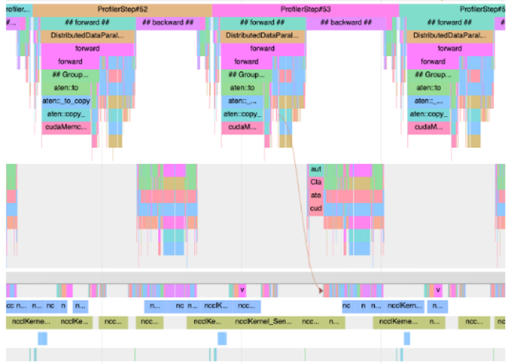


Holistic Trace Analysis (HTA)

A Tool For Analyzing Distributed AI Model Training Performance

Trace

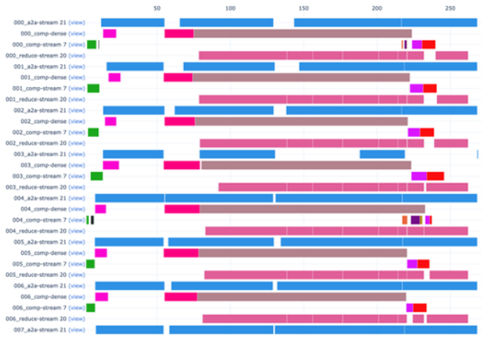

```
{
    "ph": "X",
    "cat": "cpu_op",
    "name": "aten::zeros",
    "pid": 3602,
    "tid": 3602,
    "ts": 1657898338931755,
    "dur": 9,
    "args": {
        "Trace name": "PyTorch Profiler",
        "Trace iteration": 0,
        "External id": 1,
        "Input Dims": [
            [],
            [],
            [],
            []
```

Holistic

Look at the complete system rather than its parts

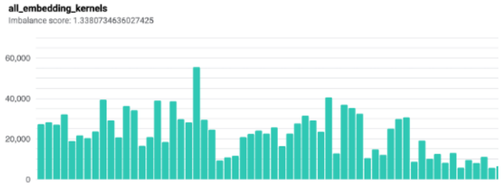
- All trainers
- The whole model
- Timing + resource usage
- Detailed model information

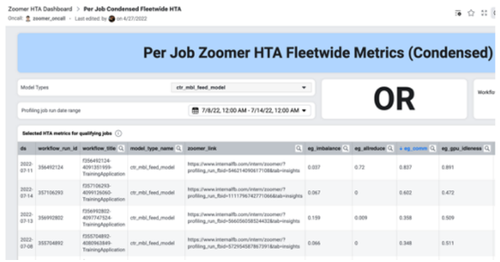


Analysis

Examine a complex thing to find its essential feature

- Is this job efficient?
- What is the bottleneck?
- Where can I optimize?
- How much gain can I get?





HTA is open sourced in 2022

- Reference: https://pytorch.org/blog/trace-analysis-for-masses/
- GitHub: https://github.com/facebookresearch/HolisticTraceAnalysis

HTA Facilitates Trace Analysis Of Distributed GPU Training

Challenges & missing pieces of analyzing large distributed GPU models

- Hard to analyze a massive number of events, CPU-GPU interactions and model's bottlenecks.
- Singe trace analysis doesn't provide the full context for distributed GPU models.

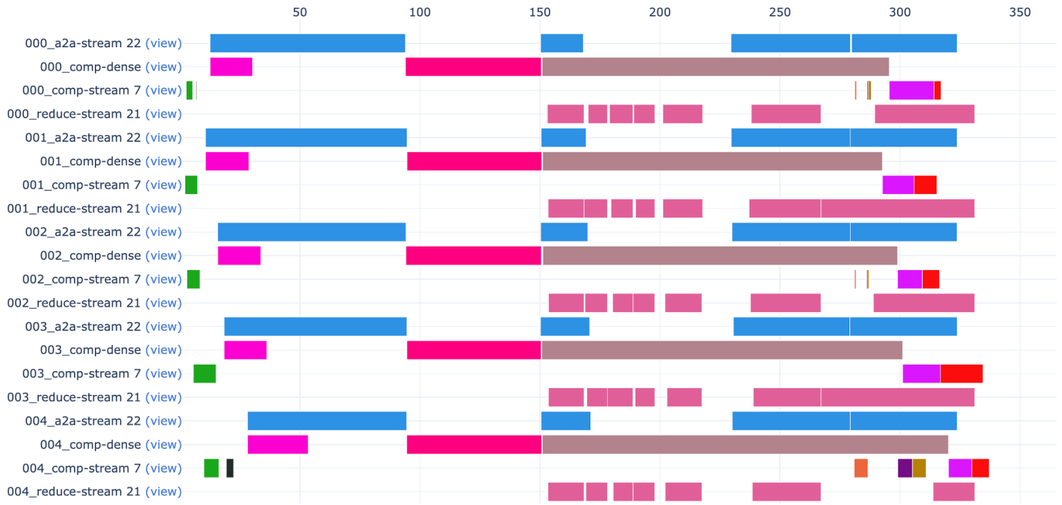
HTA solves these challenges through

- Analyzing multiple trainers of a distributed training job
  - Provide a holistic view of the model performance.
- Simplifying the analysis
  - Efficient trace data representation, perf metrics, & analysis routines.
- Supporting advanced analysis functionalities
  - Anti-patterns, trace diff/comparison, and critical path analysis.

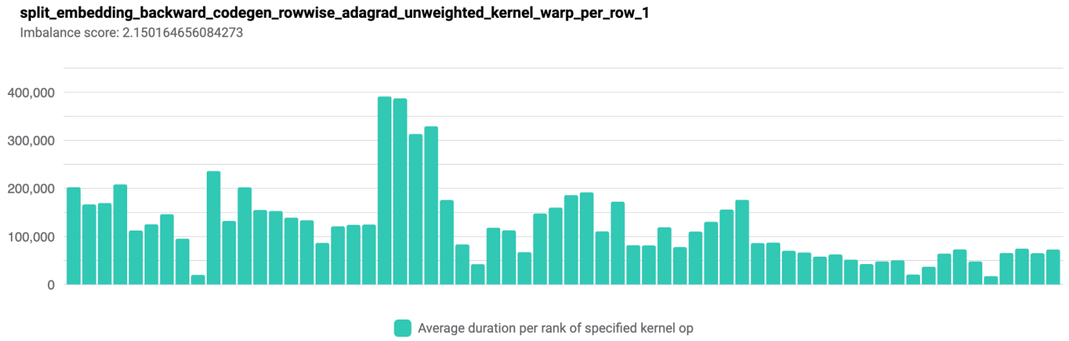
Cross-rank Analysis with HTA

For a distributed AI model job, the behavior of a single trainer doesn't paint the whole picture, e.g.:

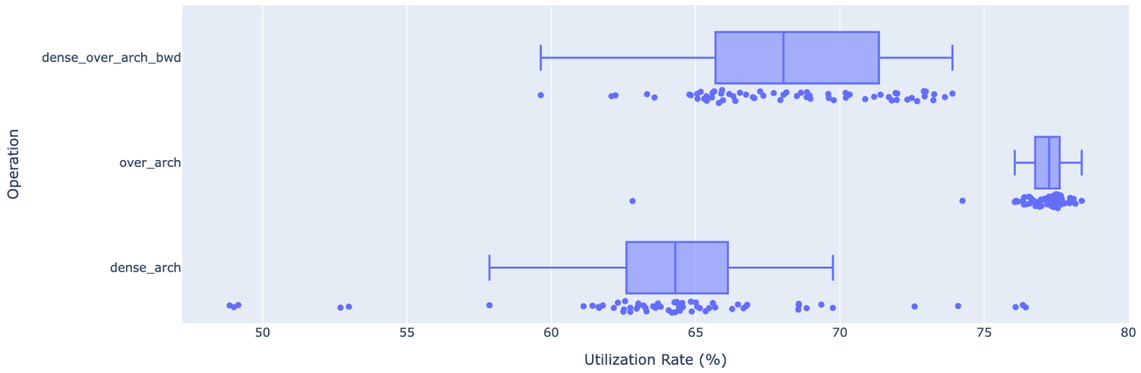
- Load imbalance due to imperfect sharding (i.e., partition) plan
- Insufficient communication-computation overlapping
- A few stragglers slowdown the entire execution.



Feature A: A multi-rank GPU kernel timeline highlights how different model parts use the GPU resources across different ranks and CUDA streams.

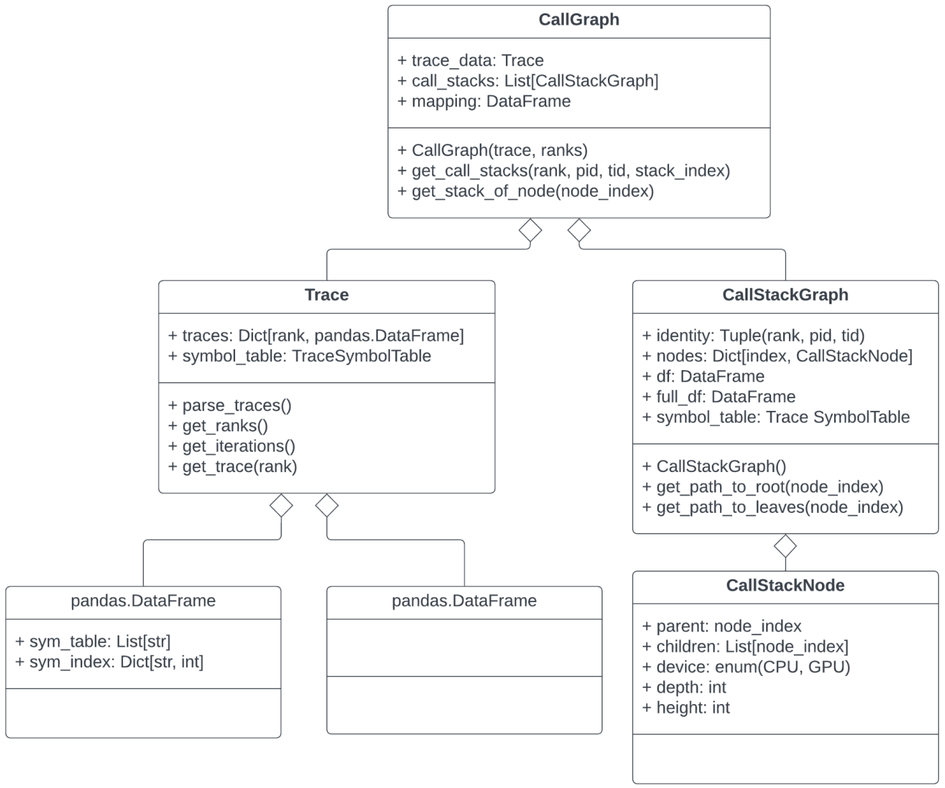


Feature B: The distribution of perf metrics like average kernel latency and imbalance scores reveal a potential performance issue in the model.



Feature C: The distribution in performance metrics, such as GPU utilization by different model components, enables engineers to strategically prioritize their optimization efforts.

HTA Organizes Trace Analysis Around Two Key Data Structures



Trace Data Frame

- Each trace is a time series of events with attributes.
- A trace is parsed into a compressed Pandas Data Frame for memory efficiency and data science tools support.
- This representation simplifies the interfaces between different analyses.

Trace Call Graph

- Captures the hierarchical, semantic relationships between trace events.
- Supports stack-based CPU/GPU operator query, statistics, and analysis.
- Simplifies more advanced graph-based trace analysis.

HTA Derives Performance Insights Using Various Analyzers

Core Data Structures

- **HTA Trace**
  - Trace
- **Trace Data Frame**
  - Trace Symbol Tables
- **Trace Call Stack Graph**
- **Trace Filter**
- **Timeline**

Single Trace Analysis

- **Trace Search and Event Statistics**
  - Basic descriptive statistics
  - Call-stack based statistics
- **Trace Events Summary**
  - CPU Operator Summary
  - GPU Kernel Summary
- **Time Breakdown Analysis**
  - Communication Kernels
  - Compute Kernels
  - GPU Idleness Analysis
- **Execution Overlapping Analysis**
- **Resource Utilization Analysis**
- **CPU Op/Annotation Highlights**
- **Performance Variation Analysis**
- **Critical Path Analysis**

Single Job Multiple Trace Analysis

- **Support all per-trace based analysis features**
- **Load Balance/Imbalance Analysis**
- **Communication Analysis**
- **Straggler Analysis**

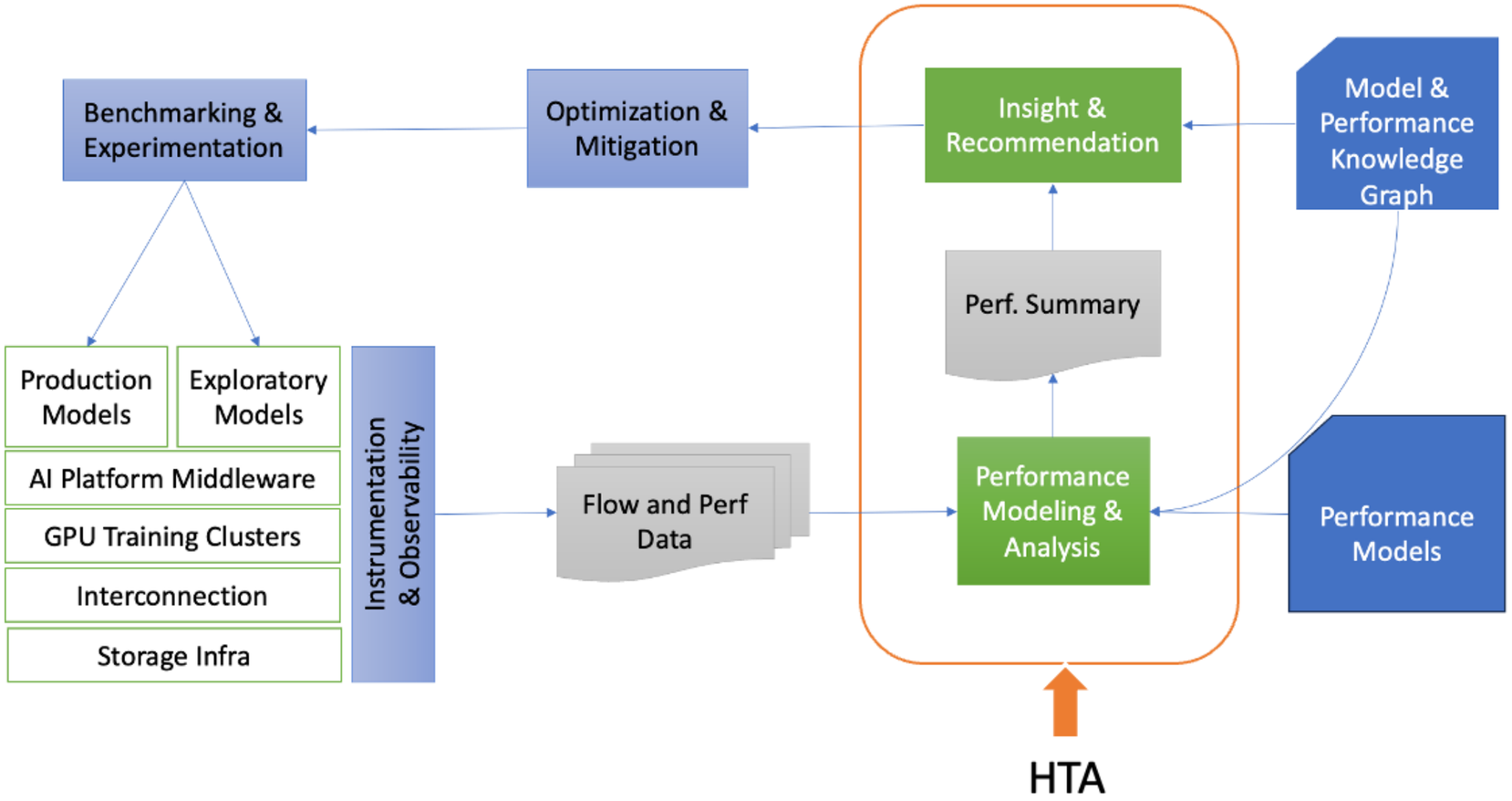
Comparative Trace Analysis

- **Trace Diff**
- **Top GPU Operators/CPU Kernels**
- **Combined Timeline Analysis**

Fleetwide Trace Analysis

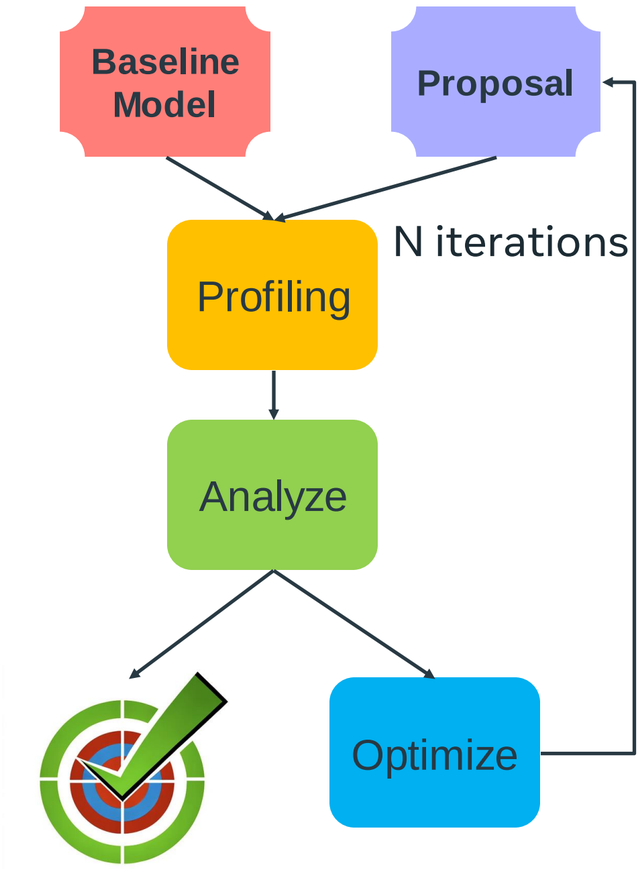
- **Estimate Gain Metrics**
- **Model-Oriented Analysis**

HTA As A Key Component In The AI Performance Ecosystem



03 Use Cases

Common Process & Techniques for Training Efficiency Optimization



Common Optimization Techniques

- Choose efficient model architecture
- Tune hyperparameters like batch sizes
- Preprocess training samples
- Utilize multiple parallelism to accelerate the execution
- Reduce CPU-GPU data copy cost & synchronization overhead
- Transform the model graph through various fusion methods
- Compress model for fast computation and communication
- Overlap communication and commutation
- Use mixed precision training like FP16
- Prefetch data batches
- Avoid all known efficiency antipatterns
- Choose proper computer system types and configurations
- Etc.

Usually, the optimization/tuning phase are conducted within a short period within constrained resources. Therefore, engineers want to get accurate and actionable insights on which set of optimization techniques should be prioritized.

Use Case #1: Identify Optimization Candidates Using Trace Statistics

- **Common Trace Statistics/Metrics**
  - Execution time distribution
  - Compute throughput
  - Data movement throughput
  - Resource utilization (CPU, GPU, Tensor Core, memory, and network)
  - Communication-computation overlap ratio
  - Event count and frequency
  - Workload distribution and imbalance
  - Kernel queue lengths
  - Call stack analysis
  - …

Example: simple kernel statistics and ordering

```
1  top_10_kernels_by_total_duration = (
2      kernel_summary.sort_values("normalized_duration", ascending=False).round(2).head(10)
3  )
4  top_10_kernels_by_total_duration
```

Kernels with the longest GPU duration should be investigated first.

	name	count	total_duration	average_duration	normalized_duration	s_name
0	64	68	777655	11436.10	0.42	ncclKernel_AllReduce_TREE_LL_Sum_float(ncclWor...
1	304	34	554297	16302.85	0.30	ncclKernel_SendRecv_RING_SIMPLE_Sum_int8_t(ncc...
2	356	4	192792	48198.00	0.10	void split_embedding_backward_codegen_rowwise_...
3	71	5	121443	24288.60	0.07	void split_embedding_codegen_forward_unweighte...
4	81	75	100326	1337.68	0.05	void at::native::(anonymous namespace)::CatArr...
5	417	4	77760	19440.00	0.04	void split_embedding_backward_codegen_rowwise_...
6	384	1859	70593	37.97	0.04	void at::native::elementwise_kernel<128, 2, at...
7	311	155	59021	380.78	0.03	Memcpy HtoD (Pinned -> Device)
8	432	160	58107	363.17	0.03	void at::native::elementwise_kernel<128, 2, at...
9	337	670	46691	69.69	0.02	void at::native::vectorized_elementwise_kernel...

```
1  top_10_kernels_by_count = (
2      kernel_summary.sort_values("count", ascending=False).round(2).head(10)
3  )
4  top_10_kernels_by_count
```

Kernels with many occurrences are candidate for kernel fusion.

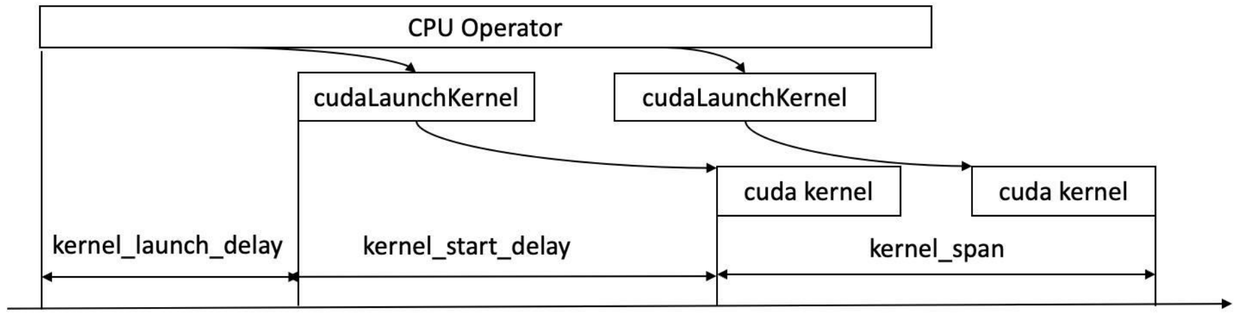
	name	count	total_duration	average_duration	normalized_duration	s_name
6	384	1859	70593	37.97	0.04	void at::native::elementwise_kernel<128, 2, at...
38	63	1191	8707	7.31	0.00	void at::native::vectorized_elementwise_kernel...
81	280	1112	2588	2.33	0.00	Memset (Device)
18	141	953	16863	17.69	0.01	void at::native::vectorized_elementwise_kernel...
15	33	760	25385	33.40	0.01	void at::native::reduce_kernel<128, 4, at::nat...
9	337	670	46691	69.69	0.02	void at::native::vectorized_elementwise_kernel...
70	242	600	3676	6.13	0.00	void at::native::vectorized_elementwise_kernel...
32	107	547	10960	20.04	0.01	Memcpy DtoD (Device -> Device)
74	274	369	3437	9.31	0.00	void splitKreduce_kernel<float, float, float, ...
62	364	340	4220	12.41	0.00	void at::native::vectorized_elementwise_kernel...

These metrics and statistics serve as indicators which optimization techniques might be required and which part of the model should be optimized.

Use Case #1: Identify Optimization Candidates Using Trace Statistics

Model performance analysis often needs to correlate the CPU operators and their corresponding GPU kernels to identify where an optimization opportunity comes from.

```
cpu_operators = df.loc[
    df.stream.eq(-1)
    & df.iteration.gt(0)
    & df.num_kernels.ge(1)
    & df.s_name.str.match("aten::")
].copy()
cpu_operators["gpu_utilization"] = (
    cpu_operators["kernel_dur_sum"] / cpu_operators["kernel_span"]
)
cpu_operators[
    [
        "index",
        "s_name",
        "iteration",
        "num_kernels",
        "kernel_span",
        "kernel_dur_sum",
        "gpu_utilization",
    ]
].sort_values("gpu_utilization").head(10)
```



	index	s_name	iteration	num_kernels	kernel_span	kernel_dur_sum	gpu_utilization
165	165	aten::cumsum	550	2	1808.0	9.0	0.004978
36482	36482	aten::cumsum	553	3	2533.0	19.0	0.007501
106656	106656	aten::sum	553	2	1332.0	16.0	0.012012
96262	96262	aten::mm	552	2	788.0	33.0	0.041878
107658	107658	aten::native_layer_norm_backward	553	2	818.0	43.0	0.052567
96144	96144	aten::sum	552	2	105.0	18.0	0.171429
159	159	aten::sum	550	2	178.0	43.0	0.241573
70579	70579	aten::sum	550	2	100.0	26.0	0.260000
31773	31773	aten::sum	552	2	221.0	62.0	0.280543
171	171	aten::cumsum	550	3	38.0	12.0	0.315789

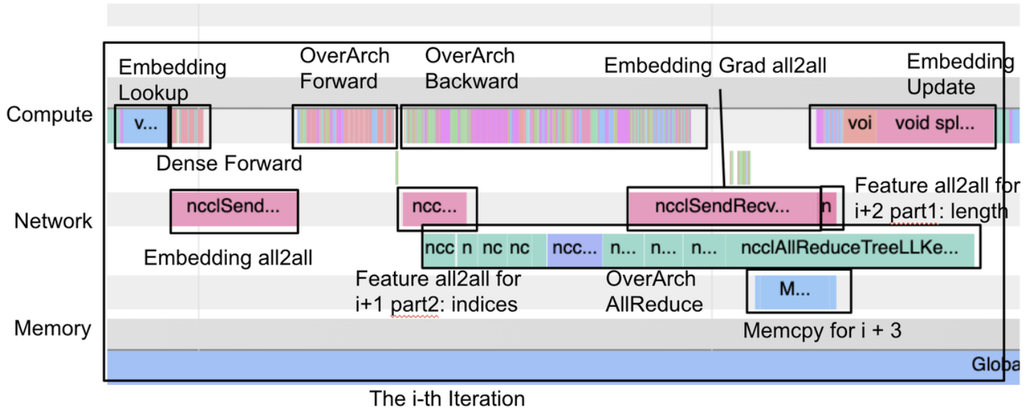
Many optimization starts with identifying key CPU operators for optimizations.

By combining the operator input size, we can calculate the FLOPS and memory b/w of specific operators to locate inefficient model components.

Use Case #2: Debug Performance Using Interactive Trace Analysis

- A large training job may use 100~10,000s GPU workers and each worker involves multiple CPU threads and multiple GPU streams.
- To handle such level of complexity, we need to apply an effective process to systematically filter the traces and focus on crucial events.

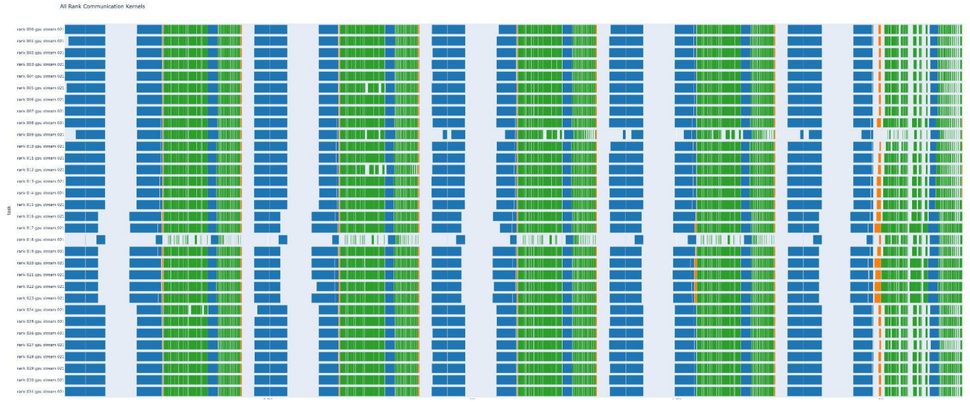
A conceptual model of a DLRM Model's workload



Identify the stragglers across all ranks

a. Visualizing the key training stages to identify the outliers.

b. Visualizing signature events (e.g., nccl and embedding kernels) to confirm the findings and get more details.

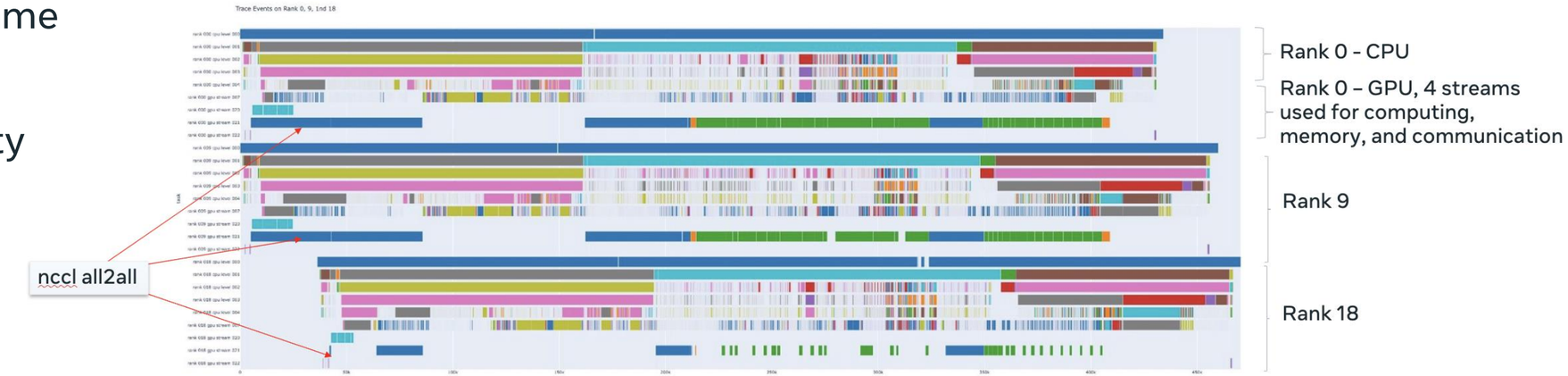


Use Case #3: Trace Comparison – Root Cause Analysis and Pattern Discovery

- Compare several different training jobs or different ranks of the same job.
- Compare at different granularity

  (1) Trace timeline
  (2) Training stages
  (3) Module level statistics
  (4) Operator/kernel statistics
  (5) Op-by-op comparison

Combined timeline visualization



Trace statistics comparison

Phase Level Comparsion | Operator Level Comparsion | Kernel Level Comparsion | Timeline View Comparison

Search Operator Name

Show 10 entries Search: fbgemm

	name	input_dims	input_type	count_baseline	count_comparison	count_delta	kernel_span_sum_baseline	kernel_span_sum_comparison	k
601	fbgemm::asynchronous_complete_cumsum	[[3072]]	['int']	0	20	20	0	494	4
7435	fbgemm::permute_2D_sparse_data	[[2560], [2560, 3072], [71544384], [], []]	['int', 'int', 'float', '', 'Scalar']	0	2	2	0	5671	5
7436	fbgemm::permute_2D_sparse_data	[[2560], [2560, 3072], [71697088], [], []]	['int', 'int', 'float', '', 'Scalar']	0	2	2	0	5686	5
7437	fbgemm::permute_2D_sparse_data	[[2560], [2560, 3072], [72085440], [], []]	['int', 'int', 'float', '', 'Scalar']	0	2	2	0	6063	6
603	fbgemm::asynchronous_complete_cumsum	[[4325376]]	['int']	0	1	1	0	61	6
604	fbgemm::asynchronous_complete_cumsum	[[7864320]]	['int']	0	1	1	0	76	7
1627	fbgemm::FloatToFP8RowwiseQuantized	[[1603584, 256], []]	['float', 'Scalar']	0	1	1	0	3280	3

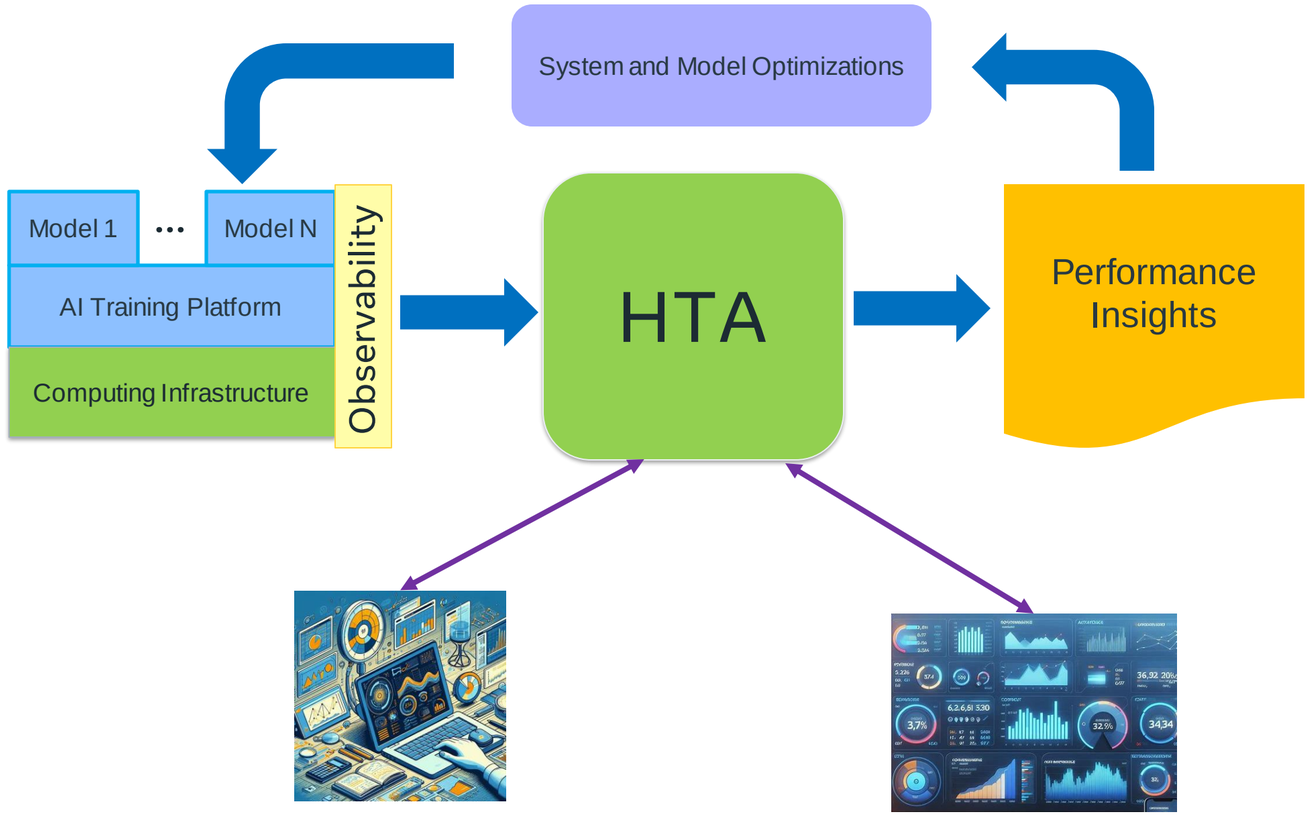
Use Case #4: Automation & Scale-out: Opportunity Detection

Inefficient Patterns	Trace Indicator	Fix
Many short GPU kernels.	Kernel duration and count.	Fuse the kernels.
Too many GPU-to-CPU sync points.	CPU operators (e.g., to, item) count.	Replace CPU ops with GPU counter parts; remove unnecessary CPU ops
Unbalanced embedding kernels.	Kernel duration distribution across ranks for sharding imbalance.	Switch to a better sharding algorithm.
Long unhidden nccl kernels.	Compute-comm overlap ratio.	Prefetch, pipeline, and reorganize operator orders to increase overlap.
Long GPU idle time.	GPU idle time percentage.	Move operations from CPU to GPU
GPU memory threshing.	cudaMalloc/cudaFree events statistics.	Reduce GPU memory usage or schedule job to nodes with large GPU memory size.
Missed optimized opportunities such as using FP16/INT8 to compress the model, utilizing tensor core, large batch size, etc.	Varies according to what the missed opportunity is.	Enable the missed opportunity.

This table highlights a sample set of inefficient patterns; new ones emerge as engineers encounter them. Once they are discovered, it is easy to map them into one or several performance indicators for automation.

04 Summary and Future Works

HTA, together with PyTorch profilers and internal tools, provide AI developer **effective performance tools** for identifying performance bottlenecks and enhancing the performance and training efficiency of large AI models.



As libraries for interactive trace analysis in Jupyter notebooks

As the backend of an automated trace analysis service for fleet-wide workloads

Optimization efforts have yielded significant efficiency improvements.

The exact gains vary with different models and corresponding baselines.

For some models, users have reported up to 7x speedup over the baseline.

Key Lessons from HTA Development

1. Align with Business Impact

Improve efficiency: Effective tools help to unlock the performance potentials of large models, which is crucial for reducing operational cost & accelerating model iterations.

2. Collaborate with End Users

Focus on Needs: Closely work with end users to understand their specific requirements and pain points, and then creating the tools

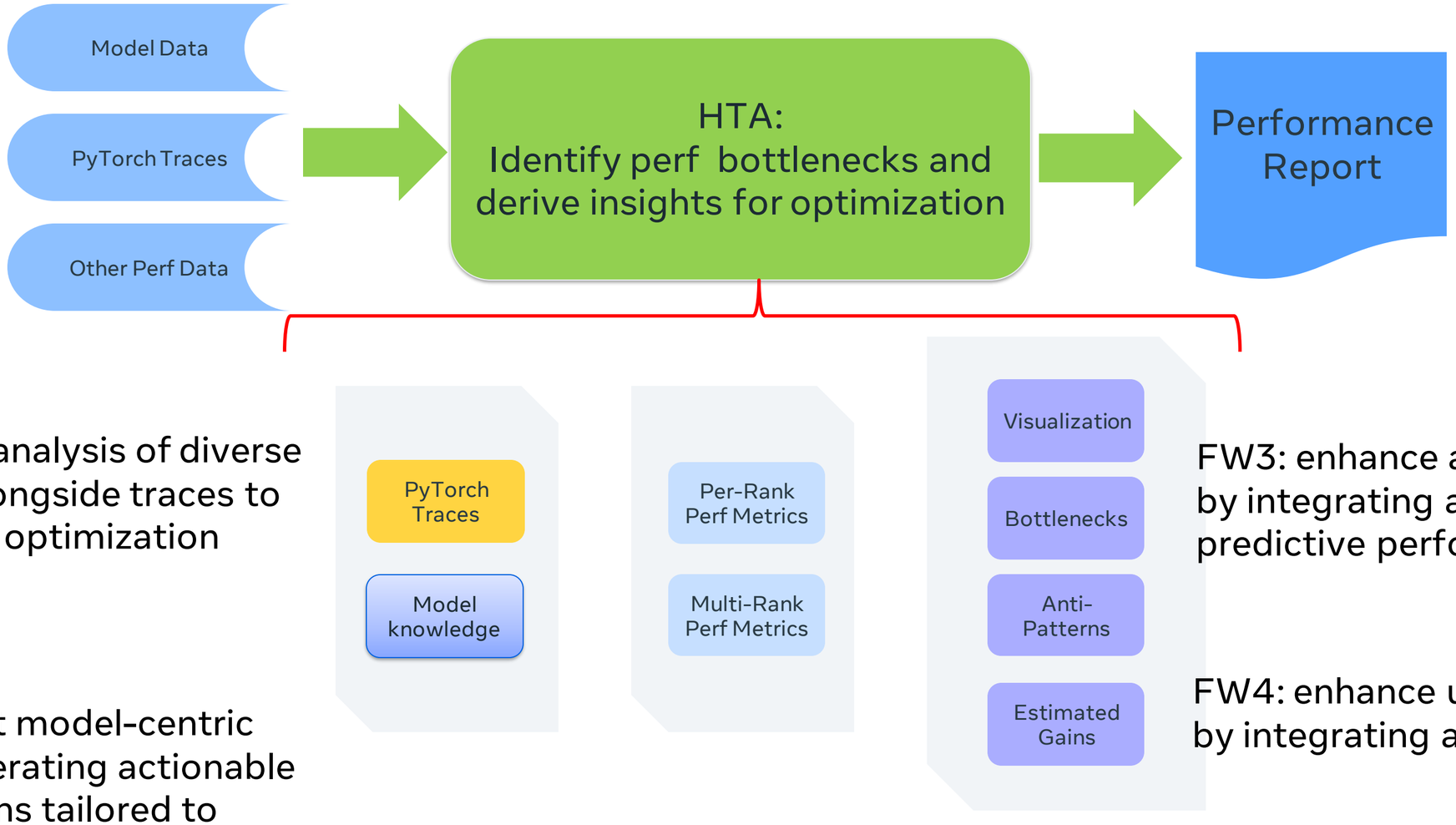
3. Automate Workflows

Lower the Barrier to Entry: Automate the trace analysis pipeline to make performance analysis more accessible for users of all levels.

4. Streamline Tool Integration

Develop Effective Framework: Build a formal but flexible framework that simplifies the integration of various trace analyzers by developers.

04 Future Work



FW1: Integrate analysis of diverse data sources alongside traces to uncover deeper optimization opportunities.

FW2: Implement model-centric analysis for generating actionable recommendations tailored to critical AI models.

FW3: enhance analytical capability by integrating analytical and predictive performance models.

FW4: enhance user experiences by integrating advances in LLMs.

Acknowledgments

HTA started with an internal project at Meta and open-sourced in 2022. We would like to express our sincere gratitude to the current and former developers, users, and managers who have contributed to the incubation, development, and open-source efforts.

Special thanks to Yifan Liu, Shuai Yang, Yusuo Hu, Jian He, Lei Tian, Wei Sun, Max Leung, Rocky Liu, Musharaf Sultan, John Bocharov, Jade Nie, Anupam Bhatnagar, Aaron Shi, Jay Chae, Louis Feng, Sung-Han Lin, Brian Coutinho, Zhongdao Ren, Xiaodong Wang, Huaqing Xiong, Robert Luo, Dima Ivashchenko, Daohang Shi, Valentin Andrei, Shi Feng, Adnan Aziz, and many others for their contributing, funding, and supporting this project.

Thanks for the NVIDIA team for offering us the opportunity to share this work and provide valuable feedback to this presentation.

05 Questions and Discussions

Contact

Xizhou Feng (fengx@meta.com)
Yuzhen Huang (yuzhenhuang@meta.com)