Research Statement

Deepak Narayanan

The end of Moore's Law has led to the rapid adoption of a number of parallel architectures, such as multicore CPUs (with SIMD), GPUs, FPGAs, and domain-specific accelerators like the TPU, each with different programming models and performance characteristics (e.g., number of cores, SIMD lane width, cache sizes). Achieving high performance on these architectures is challenging for non-expert programmers like Machine Learning engineers and data analysts, who are increasingly using computer systems to perform expensive computations (e.g., a modern ML model has trillions of floating point operations). These computations will only become more expensive going forward as ML models become larger and the amount of available data to perform analyses over increases at unprecedented rates.

My research makes it easier for programmers to achieve high performance on parallel hardware for emerging workloads such as deep neural network model training and data analytics. My main research focus has involved building systems to execute deep learning computations in a more resource-efficient and scalable way. I have examined this problem in two main contexts: a) At a microscale granularity, how should operators in a deep neural network (DNN) model be partitioned among multiple workers to maximize throughput? I built a distributed deep learning engine called PipeDream that adapts *pipelining*, an optimization used in conventional compilers and runtime systems, to accelerate DNN training performance with no reduction in the final accuracy of the model. PipeDream introduced pipeline-parallel training, a principled combination of pipelining with data and inter-layer model parallel training. Pipeline-parallel training is now used at Microsoft, Facebook, and Nvidia. b) At a macroscale granularity, how should heterogeneous resources in a shared cluster be allocated to ML training jobs to optimize scheduling objectives specified over one or more jobs (e.g., fairness, cost)? I introduced a general way to convert a wide range of scheduling policies into heterogeneity-aware policies, improving diverse objectives by up to 3.5× in a system called Gavel. I was also part of the founding team of the DAWNBench and MLPerf deep learning benchmark suites, which have been widely adopted in industry. These benchmarks popularized the time-to-accuracy performance metric for ML training. Subsequent analysis of DAWNBench and MLPerf submissions helped provide motivation for much of my other work. Additionally, I have worked on accelerating various data analytics workloads (e.g., Weld for data science and Willump for model serving) transparently, using ideas from other areas of CS like compilers and databases, to provide order-of-magnitude speedups on real applications.

I like to build real systems and design new algorithms with provable guarantees. I have worked across the stack, from low-level performance optimization, to high-level scheduling and ML training algorithms. I also like to deploy my research ideas to production users when possible: in addition to PipeDream, our Weld compiler and runtime that accelerates data analytics is being used by research groups in industry (NEC) and academia (CWI, KTH). I will continue to use my relationships with industry to identify exciting high-impact research problems.

Accelerating Distributed DNN Model Training using Pipelining

As DNN models and training datasets become larger, many organizations are adopting distributed DNN training to either decrease training time or train very large models that do not fit on a single accelerator (e.g., language models like OpenAI's GPT-3 [2]). Today, distributed training is largely performed using *intra-batch parallelism* techniques (data parallelism, model parallelism, and hybrid parallelism that combines the two), where training for a single batch of input samples is parallelized over multiple workers. These techniques, however, all hit fundamental scaling limits, either by introducing expensive all-to-all communication into the computation graph, or by lowering compute resource utilization by forcing workers to wait for intermediate outputs from other workers (in model parallelism). We showed how to use *pipelining* as a new parallelization dimension for DNN training: a batch is broken into smaller microbatches and workers process *different* microbatches concurrently. Pipelining enables new distributed training strategies that can outperform previous intra-batch methods, achieving low communication overhead and high resource utilization.

Pipelining is a common performance optimization used in various systems, such as for instruction-level parallelism in processors. However, pipelining in distributed model training presents one key difference over previous computer systems that use pipelining: training is bidirectional and stateful. A forward pass through the model is followed by a backward pass for the same set of samples which updates weight parameters, and intermediate outputs and weight

parameters used in the forward pass are needed in the backward pass. Naive pipelining can lead to weight version mismatches across forward and backward passes that compromise the accuracy of the final trained model.

I built PipeDream [5, 8], which versions state (weight parameters and intermediate activations) to ensure clean weight update semantics. In steady state, each worker in PipeDream processes a forward pass for one microbatch followed by a backward pass for a potentially different microbatch (called a 1F1B schedule). PipeDream supports different ways of stashing weight versions to trade off between memory footprint, throughput, and the number of samples over which weight gradients are averaged before updating model parameters. I also devised a memory-efficient mode in followup work called PipeDream-2BW [9], which offers a way to train large models (e.g., GPT-3 [2]) that do not fit on a single worker by stashing fewer weight versions on each worker; vanilla PipeDream cannot support the training of such large models. PipeDream automatically determines how best to partition operators across workers by reasoning about the computation times of each operator and the sizes of the tensors communicated across workers. Instead of using the same parallelization strategy for all models, PipeDream ensures that the partitioning is model- and hardware-aware.

PipeDream is able to train models to the same accuracy target up to 5× faster than data parallelism. PipeDream, when optimizing for lower memory footprint (using the 2BW memory-efficient scheme), can train large language models with 3.5 billion parameters up to 6.9× faster than model parallelism (data parallelism cannot be deployed in settings where models are too large to fit on a single worker). PipeDream and PipeDream-2BW train models with similar convergence trajectories to existing widely-used approaches like data parallelism, indicating that weight stashing provides data parallelism-like weight update semantics. Since its introduction in PipeDream, pipeline-parallel training has been adopted by Facebook to train large recommendation models and by Microsoft in its DeepSpeed training engine. I am also currently collaborating with Nvidia to integrate memory-efficient pipeline parallelism (PipeDream-2BW) in their Megatron framework to train state-of-the-art language models with thousands of GPUs.

Heterogeneous Resource Allocation for Deep Learning in Shared Clusters and Clouds

Due to the end of Moore's Law, specialized accelerators such as GPUs, TPUs, FPGAs, and other ASICs are now widely used for DNN training and inference. However, different DNN models display highly heterogeneous performance behavior across accelerator types, e.g., a ResNet-50 image classification model is about 10× faster on a later-generation Nvidia V100 GPU compared to an older-generation K80 GPU, whereas a Transformer model is only about 3.3× faster. We expect heterogeneity to increase as newer accelerator generations and domain-specific accelerators are released. This raises a difficult question for ML users: how should an organization allocate accelerators, which usually span multiple generations, among its workloads in either a private cluster or in the public cloud? This is especially challenging since organizations typically wish to optimize for a wide range of objectives, such as inter-user fairness or total dollar cost. Prior resource allocation algorithms that optimize these objectives generally do not consider device heterogeneity.

I built Gavel [11], a scheduling system that determines how heterogeneous resources in on-premise and cloud deployments should be shared among training jobs from multiple users to optimize a wide range of classical resource allocation objectives. In Gavel, we observed that existing policy objectives can be expressed as a function of a job's observed throughput. Consequently, policies can be formulated as optimization problems over the allocation. We showed how to extend these optimization problems to consider heterogeneity by extending allocations to represent the fractions of time each job should spend on each resource type, and using effective throughput, i.e., the time-weighted average of throughputs jobs observe on each resource type, in the policy objectives. Gavel's heterogeneity-aware policies can also consider performance optimizations such as space sharing (concurrent execution of applications to improve utilization), by changing the allocation representation. We showed that commonly used policies can be expressed as linear problems, which can be solved efficiently using off-the-shelf solvers. Gavel's heterogeneity-aware policies reduce objectives like average job completion time by 3.5× compared to previous schedulers that are heterogeneity-agnostic, and sustain up to 1.5× higher load using the *same* cluster by more efficiently using resources.

In subsequent work [10], we also studied the implications of using heterogeneity-aware policy formulations to pick spot instances in the public cloud, where prices and availability change with time. Heterogeneity-aware scheduling can lead to significant cost savings (up to 3.5×) by moving ML workloads across instances during training.

Understanding the Performance of Deep Learning Workloads

I also co-led our efforts to build DAWNBench [4], a benchmark suite for deep learning training. Prior to DAWNBench, benchmarks compared ML systems on the basis of throughput (operations / second). Unfortunately, throughput is a

poor metric for ML training performance since a number of ML training optimizations like reduced precision can increase throughput but reduce convergence speed or even final model accuracy. DAWNBench was the first public benchmark to compare ML systems on a simple metric, *time-to-accuracy*, that takes into account both throughput and convergence speed, and drew submissions from a number of leading vendors like Google, Alibaba, and Intel. Our analysis [3] of DAWNBench entries found that the time-to-accuracy metric was robust, and that highly-optimized systems were still bottlenecked by communication and memory-bound operators. Our experience with DAWNBench led to the time-to-accuracy metric being adopted to a more diverse set of tasks in MLPerf [7], the current industry standard for benchmarking software and hardware ML solutions with 70+ participating companies. I was a part of the founding team of MLPerf. Time-to-accuracy is now a standard evaluation metric for ML training performance in research papers and in industry.

Accelerating Data Science Applications

I have also worked on ways to accelerate data science applications through new runtime systems and algorithms.

Weld. Applications written using high-level libraries (e.g., those used in the Python data analytics stack) are composed of multiple function invocations; moving data between these function calls is often a significant bottleneck on multi-core CPUs. This is exacerbated as the memory-compute bandwidth gap grows. As part of the Weld project [12, 13], we performed end-to-end program optimization by expressing computations in an intermediate representation composed of loops and builders, a declarative construct used to collect parallel results. Optimization passes (e.g., loop fusion and vectorization) can be run over this intermediate representation to make programs less memory-bound. This is particularly useful when optimizing programs written by non-experts (e.g., scientists trying to run simulations, data scientists analyzing large datasets), allowing a way for them to write programs using higher-level primitives with good performance. Weld was able to accelerate Python programs that used Pandas and NumPy, as well as other applications such as TPC-H, by as much as 23×. Weld is being used in research projects at CWI, KTH, and NEC.

Offload Annotations. A number of accelerator-focused kernel libraries such as PyTorch and cuDF have facilitated the use of accelerators such as GPUs for data science computations. However, porting CPU-targeted applications to use accelerators is still a challenge – not all operators are supported in the kernel library, and transferring intermediate results from the CPU to the accelerator often requires repetitive boilerplate code. In Bach [14], developers can merely add annotations to existing functions to specify how functions can be offloaded; a runtime will then automatically schedule offloads efficiently. Bach achieves a median speedup of 6.3× across a variety of data science workloads.

Willump. For a number of ML models such as linear regression, feature computation is often a significant computational bottleneck for model serving. In order to reduce this bottleneck, Willump [6] leverages the fact that the majority of features are not necessary to correctly categorize the majority of inputs. Using this insight, we propose two techniques that increase throughput: automatic cascading and top-K query approximation. Willump achieves throughput improvements of up to 5× with no statistical loss of accuracy for real models from Kaggle competitions.

MacroBase. As the amount of auto-generated data increases, generating actionable data-driven insights becomes more challenging. MacroBase [1] is a data engine we built that can detect anomalies in a data stream according to some prescribed metric, and then generate explanations for these anomalies (some combination of attributes for these anomalous data records that explain the anomaly). For example, user downloads of a mobile application could be down for a particular version due to a newly introduced software bug. MacroBase was able to find anomalies and corresponding explanations in real-world deployments at Microsoft and other companies efficiently.

Future Research

I want to continue to work on systems for machine learning and faster data analytics. I have worked closely with researchers at Microsoft and Nvidia, and I hope to continue collaborating closely with users in industry and academia going forward. The following are some broad ideas I am interested in:

ML Model Management. As Machine Learning models are increasingly used in various safety- and performance-critical applications, it is becoming important to think about how ML models should be managed through their life cycle. For example, a real model deployment needs a training data collection pipeline, efficient software and hardware for model training, algorithms to prune the model to make it more compute- and memory-efficient for inference, and

model serving infrastructure. There are a number of open questions here: How do we detect model drift in a model serving system? Can we devise smarter ways to trade off accuracy for extra performance (e.g., lower precision, lossy compression while communication, etc.) than manually trying every combination of optimizations for both training and inference? On the model serving side, can we extend the idea of profile-driven optimization to determine how to parallelize model inference graphs, while being cognizant of queuing delays and average or tail latency objectives? Effectively tackling these problems in many cases involves combining theoretical insights with system building, which I love to do. Some of these problems are related to projects I have already worked on (e.g., detecting model drift is similar to finding and explaining anomalies in a data stream, parallelizing model inference is similar to parallelizing model training). I believe my experience with ML systems thus far makes me uniquely qualified to tackle the big picture questions in this area.

Unified Scheduling and Optimization. As the demand for compute resources grows, deciding how to share (possibly heterogeneous) resources efficiently among many users is a pressing problem. Current approaches to resource scheduling typically decouple resource allocation from optimization decisions. For example, deciding how to parallelize a distributed job is typically made *after* the job has been granted a set of resources from the cluster scheduler. What happens if we can make these decisions jointly instead? Could we distribute a computation using heterogeneous resources when the cluster is busy, reducing demand on faster resource types? Could we optionally decide to use architecture-specific optimizations depending on the allocated hardware (e.g., older hardware might not efficiently support irregular access patterns)? Computing allocations in this unified problem setting is more computationally expensive, and hence allocation computation would need to be accelerated to be truly practical.

Automatic Discovery of Optimizations for Data Science Programs. High-level languages like Python have become the lingua franca for scientists and data analysts. Common libraries in these languages often hide functionality behind a declarative API, making it harder to write high-performance code. Can enabling high-level optimizations akin to those performed by a query optimizer in a database help bridge this performance-usability gap? For example, can the optimal data layout for a data structure (e.g., sparse vs. dense representations for a vector or matrix) or implementation of a high-level operator (e.g., join algorithm) be adaptively determined? Systems like Weld use an adaptive optimizer to make simple optimization decisions based on input data properties. While this is a promising direction, automated optimization remains a challenge: How do we discover equivalent programs? How do we model the performance of potentially complicated sub-routines that have irregular memory access patterns? I am hopeful that I can apply ideas from my work on optimizing ML computations (which are regular and iterative) to these other domains.

I am excited to collaborate with researchers in other fields to better understand how high-performance at-scale computation can be made more easily accessible.

References

[1] P. Bailis, E. Gan, S. Madden, D. Narayanan, K. Rong, and S. Suri. Macrobase: Prioritizing Attention in Fast Data. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD)*, 2017. https://cs.stanford.edu/~deepakn/assets/papers/macrobase-sigmod17.pdf.

[2] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, et al. Language Models are Few-Shot Learners. *arXiv preprint 2005.14165*, 2020.

[3] C. Coleman, D. Kang, D. Narayanan, L. Nardi, T. Zhao, J. Zhang, P. Bailis, K. Olukotun, C. Ré, and M. Zaharia. Analysis of DAWNBench, a Time-to-Accuracy Machine Learning Performance Benchmark. *ACM SIGOPS Operating Systems Review*, 2019. https://cs.stanford.edu/~deepakn/assets/papers/dawnbench-sigops19.pdf.

[4] C. Coleman, D. Narayanan, D. Kang, T. Zhao, J. Zhang, L. Nardi, P. Bailis, K. Olukotun, C. Ré, and M. Zaharia. DAWNBench: An End-to-End Deep Learning Benchmark and Competition. In *NeurIPS Workshop on Systems for Machine Learning*, 2017. https://cs.stanford.edu/~deepakn/assets/papers/dawnbench-neurips17.pdf.

[5] A. Harlap, D. Narayanan, A. Phanishayee, V. Seshadri, N. Devanur, G. Ganger, and P. Gibbons. PipeDream: Fast and Efficient Pipeline Parallel DNN Training. *arXiv preprint 1806.03377*, 2018. https://arxiv.org/pdf/1806.03377.pdf.

[6] P. Kraft, D. Kang, D. Narayanan, S. Palkar, P. Bailis, and M. Zaharia. Willump: A Statistically-Aware End-to-end Optimizer for Machine Learning Inference. In *Third Conference on Machine Learning and Systems (MLSys)*, 2020. https://cs.stanford.edu/~deepakn/assets/papers/willump-mlsys20.pdf.

[7] P. Mattson, C. Cheng, C. Coleman, G. Diamos, P. Micikevicius, D. Patterson, H. Tang, G.-Y. Wei, P. Bailis, V. Bittorf, et al. MLPerf Training Benchmark. In *Third Conference on Machine Learning and Systems (MLSys)*, 2020. https://cs.stanford.edu/~deepakn/assets/papers/mlperf-mlsys20.pdf.

[8] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia. PipeDream: Generalized Pipeline Parallelism for DNN Training. In *27th ACM Symposium on Operating Systems Principles (SOSP)*, 2019. https://cs.stanford.edu/~deepakn/assets/papers/pipedream-sosp19.pdf.

[9] D. Narayanan, A. Phanishayee, K. Shi, X. Chen, and M. Zaharia. Memory-Efficient Pipeline-Parallel DNN Training. *arXiv preprint 2006.09503*, 2020. https://arxiv.org/pdf/2006.09503.pdf.

[10] D. Narayanan, K. Santhanam, F. Kazhamiaka, A. Phanishayee, and M. Zaharia. Analysis and Exploitation of Dynamic Pricing in the Public Cloud for ML Training. In *Workshop on Distributed Infrastructure, Systems, Programming and AI (DISPA)*, 2020. https://cs.stanford.edu/~deepakn/assets/papers/trainingonadime-dispa20.pdf.

[11] D. Narayanan, K. Santhanam, F. Kazhamiaka, A. Phanishayee, and M. Zaharia. Heterogeneity-Aware Cluster Scheduling Policies for Deep Learning Workloads. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020. https://cs.stanford.edu/~deepakn/assets/papers/gavel-osdi20.pdf.

[12] S. Palkar, J. Thomas, D. Narayanan, P. Thaker, R. Palamuttam, P. Negi, A. Shanbhag, M. Schwarzkopf, H. Pirk, S. Amarasinghe, et al. Evaluating End-to-End Optimization for Data Analytics Applications in Weld. *Proceedings of the VLDB Endowment*, 2018. https://cs.stanford.edu/~deepakn/assets/papers/weld-vldb18.pdf.

[13] S. Palkar, J. J. Thomas, A. Shanbhag, D. Narayanan, H. Pirk, M. Schwarzkopf, S. Amarasinghe, and M. Zaharia. Weld: A Common Runtime for High Performance Data Analytics. In *Conference on Innovative Data Systems Research (CIDR)*, 2017. https://cs.stanford.edu/~deepakn/assets/papers/weld-cidr17.pdf.

[14] G. Yuan, S. Palkar, D. Narayanan, and M. Zaharia. Offload Annotations: Bringing Heterogeneous Computing to Existing Libraries and Workloads. In *2020 USENIX Annual Technical Conference (ATC)*, 2020. https://cs.stanford.edu/~deepakn/assets/papers/offloadannotations-atc20.pdf.