

**PERFORMANCE ANALYSIS AND OPTIMIZATION FOR EXTREME
SCALE SYSTEMS**

by

Thomas St. John

A dissertation submitted to the Faculty of the University of Delaware in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Electrical and Computer Engineering

Summer 2021

© 2021 Thomas St. John
All Rights Reserved

PERFORMANCE ANALYSIS AND OPTIMIZATION FOR EXTREME
SCALE SYSTEMS

by

Thomas St. John

Approved: _____
Jamie Phillips, Ph.D.
Chair of the Department of Electrical and Computer Engineering

Approved: _____
Levi T. Thompson, Ph.D.
Dean of the College of Engineering

Approved: _____
Louis F. Rossi, Ph.D.
Vice Provost for Graduate and Professional Education and
Dean of the Graduate College

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____
Xiaoming Li, Ph.D.
Professor in charge of dissertation

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____
Hui Fang, Ph.D.
Member of dissertation committee

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____
Chase Cotton, Ph.D.
Member of dissertation committee

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____
Sunita Chandrasekaran, Ph.D.
Member of dissertation committee

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____
Romain Cledat, Ph.D.
Member of dissertation committee

ACKNOWLEDGEMENTS

This dissertation is the culmination of several years of work during which I had the privilege to collaborate with many excellent researchers. First of all, I'd like to acknowledge Professor Xiaoming Li. His guidance during the course of this effort has proven invaluable. I also need to thank Romain Cledat, Benoit Meister, and Andres Marquez for their assistance during the early stages of this work which was essential to the formulation of my topic. I also need to thank my wife, Xiaoyuan Han, who provided motivation during the challenges of completing my dissertation while working full-time. Finally, I'd like to thank the other contributors to the MLPerf and Parallel Research Kernels projects.

TABLE OF CONTENTS

LIST OF FIGURES	x
ABSTRACT	xiii
Chapter	
1 INTRODUCTION	1
2 PARALLEL RESEARCH KERNELS	3
2.1 Background and Motivation	4
2.1.1 Parallel Research Kernels	4
2.1.2 Programming Models	7
2.2 Implementation and Performance Results	10
2.2.1 Stencil Implementation Details and Performance	12
2.2.2 Synchron_p2p Implementation Details and Performance	16
2.2.3 Transpose Implementation Details and Performance	19
3 MLPERF TRAINING BENCHMARK	23
3.1 Background	25
3.1.1 Unique Challenges of Benchmark Training	25
3.1.1.1 Effect of Optimizations on Quality	26
3.1.1.2 Effect of Scale on Time to Train	26
3.1.1.3 Run-to-Run Variation	27

3.1.1.4	Diverse Software	27
3.2	MLPerf Training Benchmark	28
3.2.1	Benchmark Suite	29
3.2.1.1	Image Classification	29
3.2.1.2	Object Detection and Segmentation	30
3.2.1.3	Translation	30
3.2.1.4	Reinforcement Learning	31
3.2.1.5	Recommendation	32
3.2.2	Time-to-Train Performance Metric	32
3.2.2.1	Timing Rules	33
3.2.2.2	Number of Timing Runs	34
3.2.3	Choice of Quality Thresholds	34
3.2.4	References and Hyperparameters	35
3.3	Benchmarking Process	36
3.3.1	Submission and Review	36
3.3.2	Reporting Results	37
3.3.2.1	Submission Divisions	37
3.3.2.2	System Categories	37
3.3.2.3	Reporting Scale	38
3.3.2.4	Reporting Scores	39
3.4	Results	39
4	MLPERF INFERENCE BENCHMARK	46
4.1	Inference Benchmarking Challenges	50
4.1.1	Diversity of Models	50
4.1.2	Deployment Scenario Diversity	50
4.1.3	Inference System Diversity	52
4.2	Benchmark Design	53
4.2.1	Representative, Broadly Accessible Workloads	53

4.2.2	Robust Quality Targets	56
4.2.3	Realistic End-User Scenarios	57
4.2.4	Statistically Confident Tail-Latency Bounds	59
4.3	Inference Submission System	62
4.3.1	System Under Test	62
4.3.2	Load Generator	65
4.3.3	Data Set	67
4.3.4	Accuracy Checker	67
4.4	Submission System Evaluation	67
4.4.1	Result Submissions, Divisions, and Categories	68
4.4.2	Result Review	69
4.4.3	Result Reporting	70
4.5	Benchmark Assessment	70
4.5.1	Task Coverage	71
4.5.2	Scenario Usage	71
4.5.3	Processor Types and Software Frameworks	73
4.5.4	System Diversity	74
4.5.5	Open Division	75
4.6	Lessons Learned	76
4.6.1	Models: Breadth vs. Use-Case Depth	76
4.6.2	Metrics: Latency vs. Throughput	77
4.6.3	Data Sets: Public vs. Private	78
4.6.4	Performance: Modeled vs. Measured	78
4.6.5	Process: Audits and Auditability	79
5	A SCHEDULER-DRIVEN ADAPTIVE FRAMEWORK FOR EXTREME SCALE SOFTWARE STACKS	80
5.1	Adaptive Data Compression Engine	82
5.2	Experimental Results	83
6	RELATED WORK	84
6.1	Related Work	84
6.2	MLPerf Training Benchmark	85

6.3	MLPerf Inference Benchmark	86
7	CONCLUSION	88
7.1	Parallel Research Kernels	88
7.2	MLPerf Training Benchmark	89
7.3	MLPerf Inference Benchmark	90
7.4	ASAFESSS	92
	BIBLIOGRAPHY	93

LIST OF FIGURES

2.1	Hierarchical domain decomposition for MPISHM	14
2.2	Absolute performance of Stencil application, divided by performance of single-node MPI1 and number of nodes used.	16
2.3	Absolute Performance of Synchron_p2p kernel	19
2.4	Absolute performance of Transpose kernel	22
3.1	Training epochs to reach the target quality for the MLPerf v0.5 NCF (a) and MiniGo (b) benchmarks. Each experiment uses identical hyperparameters except for the random seed. For MiniGo, we observed considerable variability across runs even when fixing the random seed (same color).	40
3.2	41
3.3	MLPerf Training v0.5 benchmarks	41
3.4	Top-1 accuracy of MLPerf v0.5 ResNet-50 benchmark over 100 epochs for five runs (denoted by color) with identical hyperparameters but different random seeds. The dashed line indicates the quality target of 74.9% Top-1 accuracy. The early training phase exhibits much more variability than later phases. . .	42
3.5	MLPerf modifiable hyperparameters	43
3.6	Speedup in the fastest 16-chip entry from MLPerf version v0.5 to v0.6 for various benchmarks common to both (a) along with quality target increases (b).	44
3.7	Number of chips necessary to produce the fastest time to solution for MLPerf versions v0.5 to v0.6. This number increased by as much as 5.5x	45

4.1	An example of ML model diversity for image classification (plot from [4]). No single model is optimal; each one presents a design tradeoff between accuracy, memory requirements, and computational complexity.	51
4.2	The diversity of options at every level of the stack, along with combinations across the layers, make benchmarking inference systems difficult.	52
4.3	ML tasks in MLPerf Inference v0.5. Each one reflects critical commercial and research use cases for a large class of submitters, and together they cover a broad set of computing motifs (e.g., CNNs and RNNs).	54
4.4	Scenario description and metrics. Each scenario targets a real-world use case based on customer and vendor input.	58
4.5	Latency constraints in the multi-stream and server scenarios.	60
4.6	60
4.7	Query requirements for statistical confidence. All results must meet the minimum LoadGen scenario requirements.	61
4.8	Number of queries and samples per query for each task.	61
4.9	MLPerf Inference system under test (SUT) and associated components. First, the LoadGen requests that the SUT load samples (1). The SUT then loads samples into memory (2-3) and signals the LoadGen when it is ready (4). Next, the LoadGen issues requests to the SUT (5). The benchmark processes the results and returns them to the LoadGen (6), which then outputs logs for the accuracy script to read and verify (7).	63
4.10	Timing and number of queries from the LoadGen.	66
4.11	Results from the closed division. The distribution indicates we selected representative workloads for the benchmark’s initial release.	72
4.12	High coverage of models and scenarios.	73

4.13	Throughput degradation from server scenario (which has a latency constraint) for 11 arbitrarily chosen systems from the closed division. The server scenario performance for each model is normalized to the performance of that same model in the offline scenario. A score of 1 corresponds to a model delivering the same throughput for the offline and server scenarios. Some systems, including C, D, I, and J, lack results for certain models because participants need not submit results for all models.	74
4.14	Results from the closed division. They cover almost every kind of processor architecture - CPUs, GPUs, DSPs, FPGAs, and ASICs. .	75
4.15	Framework versus hardware architecture.	76
4.16	Performance for different models in the single-stream (SS), multi-stream (MS), server (S), and offline (O) scenarios. Scores are relative to the performance of the slowest system for the particular scenario.	77
5.1	Flow of the scheduler-driven adaptive optimization framework . . .	81

ABSTRACT

Thanks to the death of Dennard scaling and the slowing of Moore’s Law, future compute platforms are becoming increasingly complex as they attempt to continue to scale either overall performance or performance efficiency. Consequently, analyzing the performance of such systems has also become an ever-more complex problem. Gone are the days where peak FLOPS numbers were a reliable indicator of real-world performance, and they have been replaced by a field where performance bottlenecks can occur in any component of these platforms. The reality of the situation requires that performance analysis be conducted at many levels throughout the lifecycle of application development, focusing on both individual elements of computation and also end-to-end system performance of the application at scale, as well as many additional points within that spectrum. In this work, we present both component-level and system-level performance analysis tools, as well as a non-traditional application of using online performance analysis to enable dynamic performance optimization.

Chapter 1

INTRODUCTION

Thanks to the death of Dennard scaling and the slowing of Moore’s Law[19], future compute platforms are becoming increasingly complex as they attempt to continue to scale either overall performance or performance efficiency. Consequently, analyzing the performance of such systems has also become an ever-more complex problem. Gone are the days where peak FLOPS numbers were a reliable indicator of real-world performance, and they have been replaced by a field where performance bottlenecks can occur in any component of these platforms. The reality of the situation requires that performance analysis be conducted at many levels throughout the lifecycle of application development, focusing on both individual elements of computation and also end-to-end system performance of the application at scale, as well as many additional points within that spectrum. In this work, we present both component-level and system-level performance analysis tools, as well as a non-traditional application of using online performance analysis to enable dynamic performance optimization.

Chapter 2 covers the Parallel Research Kernels[138], a set of building blocks commonly found in high performance computing applications (stencil computation, sparse matrix-vector multiplication, matrix transposition, etc.), which can be used to evaluate parallel programming models for future systems. Chapter 3 covers the MLPerfTM training benchmark suite[97], an end-to-end system performance benchmark for machine learning training workloads. Chapter 4 covers the MLPerf inference benchmark suite[118], a similar machine learning system performance benchmark focused on inference workloads. Chapter 5 covers ASAFESSS[126], a scheduler-driven adaptive framework for extreme scale software stacks, which is a framework which

supports online workload characterization and can be used to dynamically drive online performance optimizations for large-scale systems where static analysis can be insufficient to determine which optimizations techniques are beneficial ahead of time. Chapter 6 provides a description of prior work related to these efforts and Chapter 7 provides conclusions made based on this work.

Chapter 2

PARALLEL RESEARCH KERNELS

With the stagnation of frequency scaling in modern processors and the emergence of power as a critical constraint in the design and deployment of supercomputers, hardware parallelism has been increasing dramatically. Managing expanded concurrency while maintaining voluminous validated legacy codes is a challenge, compounded by the expected increase in faults, and performance non-determinism. These challenges have inspired the development of *dynamic runtime systems* (DRTS). While DRTS offer much promise for exascale, it is vitally important that they meet the needs of HPC applications at all relevant scales of parallelism. Even if DRTS prove more suitable for asynchronous and/or task-oriented algorithms, they must be able to support the regular/structured algorithms and more synchronous/procedural styles common in HPC.

Our goal is to study many relevant HPC programming models, evaluating them on their ability to support ubiquitous application patterns. The m by n complexity of this problem - m programming models and n application patterns - constrains the scope greatly. We have learned from the community that even mini-applications are difficult to port to new models and that many algorithms in different domains map to the same parallel programming patterns, meaning that porting many mini-applications may not produce a proportional understanding. On the other hand, studying a small number of programming models, or even treating MPI as a single programming model, does not sufficiently answer the question of how to implement HPC applications for peta- and exascale.

We evaluate a range of different programming models using the Parallel Research Kernels (PRK)[138]. The PRK comprise about a dozen different application *patterns*;

we focus on the three that are the most relevant to scientific computing applications running on modern HPC systems: Synchron_p2p, Stencil, and Transpose. The models evaluated include:

- MPI¹ and MPI+X models in the form of MPI1+OpenMP (MPIOPENMP) and MPISHM²
- established PGAS models SHMEM and UPC (Unified Parallel C), and the relatively new but semantically similar MPIRMA (MPI with direct Remote Memory Access)
- the non-evolutionary models Charm++ and Grappa, both of which are oriented at irregular and unstructured computations, but which are still capable of implementing regular, structured computations

2.1 Background and Motivation

In this section, we provide context for our workloads, core programming models, and specific combinations of programming models used in our study.

2.1.1 Parallel Research Kernels

The PRK[137] are a suite of elementary operations (*kernels*) that can be used to study the suitability of parallel systems for parallel application programming. The full suite includes about a dozen kernels designed to expose various performance bottlenecks of a parallel system. In many cases, programmers can relate the observed performance of the PRK to the expected bottlenecks in their applications, allowing the PRK to serve as proxies for full applications.

The PRK are defined as paper-and-pencil operations independent of any particular implementation, although they do prescribe certain rules regarding data distribution on distributed memory systems. In addition to a canonical serial version,

¹ MPI refers to any model based upon the Message Passing Interface standard, while MPI1 refers to the usage of MPI two-sided communication.

² MPISHM refers to an MPI1-like program that uses shared memory within the node of explicit communication. It is similar to MPIOPENMP, but with significant differences related to locality and runtime overheads, among others. We use MPI-3 as our implementation of shared memory.

we provide reference implementations³ using various parallelization techniques. The PRK implementations are self-contained, are arbitrarily scalable (problem size, compute resources), synthesize any data they need, and validate the results. They have been used to study new hardware architectures using simulators, as well as to evaluate new features in programming models[43].

For our study of programming systems for exascale computers, we use three kernels, in order of nominally increasing granularity: `Synch_p2p`, `Stencil`, and `Transpose`. In our description of the reference implementations of the kernels we will refer to the units of execution generally as *ranks*. This should be replaced *threads*, *chares*, or *PEs* as needed, depending on the particular programming model. For grid-based kernels, we always opt for 2D over 3D. The reason is two-fold. First, 2D problems typically have fewer options to exploit concurrency and more overhead than 3D (worse surface-to-volume ratios in domain decomposition problems, especially in the strong scaling case), which creates extra stress for the runtime that we want to study. Second, 2D codes are more compact than 3D codes - a PRK design goal to maximize portability - but without sacrificing the realism that 1D codes do.

Synch_p2p involves a simple stencil-based problem. A two-dimensional array A of size $n \times m$, representing scalar values at grid points, is distributed among the ranks in vertical strips. We apply the following stencil: $A_{i,j} = A_{i-1,j} + A_{i,j-1} - A_{i-1,j-1}$, with the condition that only updated array values or (fixed) boundary values may be used. The 2D data dependencies are resolved using a 1D software pipeline. `Synch_p2p` models the algorithmic structure of well-known benchmarks such as the LU-SGS NAS Parallel Benchmark[10], and SNAP[148], but is even more stressful w.r.t. communication as the kernel only has two spatial dimensions. It is not data-parallel, and must synchronize strictly on a point-to-point basis. Such synchronizations can be implemented using shared flags plus the flush directive in OpenMP, point-to-point messages in MPI1 and Charm++, put-wait in SHMEM and shared pointer access in UPC. In Grappa, we

³ PRK open source repository: <https://github.com/ParRes/Kernels>

use Full-Empty Bits (FEB) to implement producer/consumer synchronization. These techniques incur a forced write-to and read-from shared memory in OpenMP, and an inter-process communication latency for MPI1, SHMEM, UPC, Charm++, and Grappa. Performance is constrained by the overhead of frequent synchronization between ranks. Note: it is possible to increase the algorithm’s granularity by grouping grid lines together explicitly, and our PRK implementation does indeed offer that option, but we do not consider that conforming, as we are interested in fine-grain application behavior.

The second kernel is **Stencil**, which applies a scalar, data-parallel stencil operator to the interior of a two-dimensional discretization grid of size $n \times n$. The stencil, which represents a discrete divergence operator, has radius r . In operator notation: $A = S(r)B$, where A and B are two-dimensional arrays and $S(r)$ is the stencil operator of radius r . The distributed memory versions used in this paper employ two-dimensional domain decompositions to minimize the surface to volume ratio, and hence the communication volume. Communication is required to obtain ghost point values from logically nearest neighbors.

The third kernel is **Transpose**, in which a square matrix B of order n is divided into strips (columns) among the ranks, which store its transpose B^T in matrix A . The matrices are distributed identically, necessitating a *global* rearrangement of the data (all-to-all communication), as well as a *local* rearrangement (per-rank transpose of matrix tiles). Canonical execution of Transpose may lead to high numbers of cache and TLB misses, due to the strided nature of the data access of either A or B . To reduce this effect, we employ blocking to implement the local transpose operation. For all programming models, we use the same block size (32 x 32).

Each of the kernels’ main operations is executed a number of times, and performance is computed by timing the entire sequence of operations, but skipping the first to reduce the effects of implicit initialization (e.g., network connections).

2.1.2 Programming Models

MPI1

The first version of the MPI standard[51] focused on message-passing functionality, defining point-to-point and collective operations, as well as the critical infrastructure that makes MPI1 portable, extensible, and composable. The set of functions in MPI-1 is quite complete with respect to practical parallel programming within Hoare’s Communicating Sequential Processes (CSP) model.

Contrary to popular belief, MPI-1 did not implement Valiant’s Bulk Synchronous Parallel (BSP) model (this pattern was introduced in MPI-2 via `MPI_Win_fence`). Much of the success of MPI has been built upon the extensibility of the CSP model.

MPIOOPENMP

MPI-2[52] introduced the awareness of threads to MPI. Support of threads is not holistic; communication routines merely became thread-safe according to the user’s request during initialization, but no changes to the communication routines were made that might make multithreaded MPI programs more efficient. For example, while the MPI standard does not explicitly require locks, most implementations use coarse-grain mutual exclusion for `MPI_THREAD_MULTIPLE`, which can have significant (negative) performance effects.

Because of the performance issues associated with concurrent access to MPI, many users only perform MPI calls outside of threaded regions. The fork-join model limits scalability, but the practical consequences of this are often relatively small if an MPI process is associated with each NUMA domain and threads are used only within that. For small numbers of threads, it is often the case that the overhead of mutual exclusion, which persists throughout the application, regardless of whether or not it is actually needed⁴ is worse than the effect of joining (or merely serializing) all the threads to make MPI calls.

MPISHM: MPI plus interprocess shared memory

⁴ MPI calls outside of threaded regions do not require mutual exclusion, but MPI has no way to know this.

While MPI+Threads is the most commonly applied solution to the multicore problem, an alternative model employs interprocess shared memory. Heroux and coworkers proposed to use interprocess shared memory instead of threads in the context of MPI applications[68]. Members of the MPI Forum have elaborated upon this model[70] and it became part of MPI-3[53]. This model is also referred to as MPI+MPI, as a way of connecting to MPI+X, where X equals MPI-3 shared memory.

SHMEM

Pioneered by Cray Inc. in the 1990s[50], SHMEM is a library specification that supports programming for a logically shared but physically distributed address space. All processes see variables in the global address space (symmetric variables), but also have their own local view in the partitioned global address space (PGAS). The SHMEM API provides a set of communication operations, similar to the MPI one-sided communication routines. SHMEM emphasizes one-sided communication features that can be mapped directly to hardware, which offers the potential for a more efficient implementation compared to MPI two-sided communication. Historically, SHMEM implementations have varied between vendor implementations, making it difficult to write portable SHMEM programs. Recently, OpenSHMEM[110] has emerged as a vendor-independent community standard and associated reference implementation, which has increased interest in SHMEM.

MPIRMA: MPI one-sided communication

MPI-2[52] introduced remote memory access (RMA), or one-sided communication, to MPI. This was not a successful effort, due to semantic constraints and implementation issues. With MPI-3[53], the semantic issues were largely addressed, and MPIRMA is able to support the semantics of important PGAS models like SHMEM, Global Arrays (ARMCI), and Fortran coarrays, as demonstrated by OSHMPI⁵,

⁵ <https://github.com/jeffhammond/oshmpi>

ARMCI-MPI⁶, and OpenCoarrays⁷, respectively, in addition to BSP and message-passing synchronization. We employ both passive- and active-target synchronization in the PRK. In all kernels, we could use multiple styles of MPI-3 RMA, but we chose only one each for `Synch_p2p` and `Stencil`, and two for `Transpose`. The reason is that, to the extent that MPI-3 RMA can be *semantically* equivalent to SHMEM, the SHMEM implementation tells us what is possible with the passive-target motif, up to implementation differences (which may be substantial today).

UPC

UPC[134] is a PGAS extension to the ISO C99 language. It handles shared and distributed memory with a uniform programming interface. Like SHMEM, UPC presents the abstraction of a global, shared address space. That space is partitioned among threads in a well-defined, user-prescribed way, allowing the programmer to exploit thread-to-data affinity and hence improve locality. Explicit put and get operations can also be used to increase granularity.

Charm++

The Charm++ programming framework[82] provides asynchronous remote method invocation (RMI) on persistent but migratable objects (*chares*) organized in multi-dimensional arrays. Communication between chares takes place by specifying the parameters of the remote method (marshalling) or by sending the remote chare a message. While similar to MPI messaging, there are important differences. Chares are virtualized and may migrate between compute units transparently to the programmer. The flow of a program is not statically defined in terms of two-sided message passing; RMIs (with their data) are placed in task queues at the recipient, and are scheduled by the runtime, subject to an execution policy. Hence, messages may arrive in different order and well before they are needed, potentially providing substantial asynchrony. In addition, multiple chares may be assigned to the same compute unit

⁶ <http://git.mpich.org/armci-mpi.git>

⁷ <https://github.com/sourceryinstitute/opencoarrays>

(overdecomposition), which may provide more asynchrony and latency hiding. RMIs are nominally non-blocking, but may wait for certain events to occur if they contain *Structured Dagger* (SDAG) code. Messages and SDAG constructs generally offer the best performance and control. We use these for the Charm++ versions of the PRK.

Grappa

The Grappa runtime[108] is loosely based on innovations explored in the Tera MTA and Cray XMT architectures, which allow very fast, hardware supported context switching between multiple instruction streams per core. This provides substantial latency hiding, but requires special hardware support. Grappa emulates these features in software using fast, user-level context switching, in addition to fine-grained communication and synchronization using FEBs across a partitioned global address space. The runtime watches for multiple asynchronous messages with a common destination from independent instruction streams, or from asynchronous communications in the same instruction stream, and batches them into a single transfer, transparently to the user. This can greatly improve the performance of fine-grain workloads.

2.2 Implementation and Performance Results

We performed three scaling experiments wherein problem size remains constant as more compute and memory resources are added for each kernel. For Stencil, we select a star-shaped stencil of radius four (i.e. a 17-point star), and a grid of 49152 x 49152 points. We use the same dimensions for the grid of Synch_p2p and the matrix in Transpose. Problem sizes, numbers of ranks, and overdecomposition factors are chosen to allow for an even load across all processing elements when using all the cores on a node. An exception is made for Charm++, which performs better if one core per node is reserved for communication. This core is counted in the resource consumption, but does no computational work. To balance the load among the remaining cores on a node, we change the problem size for Charm++ to 47104 for all kernels. While strong scaling from 1 to 512 nodes is not common in real applications, it exaggerates runtime overheads and allows us to draw conclusions about performance at much higher

scale with a large overall problem size, but a per-node problem size within the range considered.

Our experiments were conducted on NERSC Edison, a Cray XC30 supercomputer with two 12-core Intel Xeon E5-2695 processors per node with the Aries interconnect in a Dragonfly topology. We used Intel 15.0.1.133 C/C++ compiler for all codes, except that Cray Compiler Environment (CCE) 8.4.0.219 was used for Cray UPC, and GCC 4.9.2 was used for Grappa. Berkeley UPC compiler 2.20.2 was used with the same Intel C/C++ compiler. System library versions were Cray MPT (MPI and SHMEM) 7.2.1, uGNI 6.0, and DMAPP 7.0.1. All MPIRMA transpose experiments enabled asynchronous progress; passive target RMA employed the DMAPP implementation⁸. While asynchronous progress in MPI may introduce overheads, it is the natural way to use one-sided communication and compares fairly to SHMEM and UPC, which makes asynchronous progression on this platform.

All implementations of the PRK are derived from the same original MPI1 source code, and are functionally as similar as feasible, except for scheduling by the runtime, orchestration of communication, and data sharing. Although the implementations were written in a portable way, we consider them of high quality, since they were co-written and/or reviewed by the developers of the runtimes (Grappa, Charm++), or by other PRK contributors, who are experts in the other runtimes. All kernels are compiled with the same optimization flag (-O3). The use of Cray UPC and GCC C++ compilers (for Grappa) introduces a discrepancy that cannot be reconciled. However, sample trials using Gnu and Intel compilers for the same kernels show negligible performance differences, so we posit that this discrepancy does not substantively alter our results or conclusions with respect to Grappa.

Our methodology for presenting performance results is as follows. When a kernel is expected to show good scalability, we show normalized performance, in which absolute performance using P nodes is divided by P , as well as by the performance of

⁸ The environment variables are `MPICH_RMA_OVER_DMAPP`, `MPICH_MAX_THREAD_SAFETY`, and `MPICH_NEMESIS_ASYNC_PROGRESS`.

the MPI1 code using a single node. This is the method we used for Stencil. When a kernel is expected to experience moderate to severe scalability problems, we show absolute performance on a log (cores)-linear (performance) scale, since this best depicts performance differences between the different models. This is the method we used for Synchron_p2p and Transpose. For all kernels, we report the *maximum* performance observed over a large number of experiments. This is because, at the time of our experiments, NERSC Edison experienced intermittent network issues, leading to a number of outliers that would unfairly penalize models that happened to run at an unfavorable time. Selecting the maximum over a large number of attempts produced the most consistent, fairest data.

2.2.1 Stencil Implementation Details and Performance

For the Stencil kernel, we employ a 2D domain decomposition, where each MPI rank running on a processor core is assigned a tile within the overall grid that it updates in each iteration, based on the values computed in the previous iterations, as well as on values generated by logically nearest neighbor ranks. We employ the same method in Charm++, except that we allow an overdecomposition, which means that we divide the grid into more tiles - each assigned to a chore - than the number of processor cores employed. This technique is often used to provide overlap of computation and communication, since a core can work on updating the tile associated with one of its chores while the communication needed for another of its chores is in progress. Overdecomposition could be applied in MPI as well, in principle, but would require explicit management of multiple tiles per rank, constituting a significant programming complication.

In all cases, the tiles are chosen as close to square as possible to minimize their surface to volume ratio, which in turn minimizes the number of tiles beyond the required minimum value, and hence increases the communication volume.

MPI1

Within each iteration, an MPI rank determines whether it has a topological neighbor in each of four coordinate directions (+x, -x, +y, -y) and posts asynchronous receive calls (`MPI_Irecv`) for data from each of these neighbors. Subsequently, it copies data from its grid boundaries into communication buffers and posts asynchronous send calls (`MPI_Isend`) to each of its neighbors. Finally, it waits for all asynchronous communications belonging to the current iteration to finish (`MPI_Wait`) before copying the received data into ghost point locations and updating its tile values. It should be noted that the primary purpose of using asynchronous calls is to avoid deadlock. Since the load is fully balanced, all ranks engage in communication at effectively the same time, and negligible overlap of computation and communication occurs.

MPIOOPENMP

We spawn one MPI rank per socket, and use OpenMP to parallelize the performance-critical loop nest in the code among the 12 cores (our experiments show that one MPI rank per node performs more poorly, due to NUMA effects). MPI communications only take place outside the threaded code regions. In a separate OpenMP-only experiment, we determined that, for the granularities involved in this problem, OpenMP’s fork-join overhead is negligible, justifying the use of `omp parallel for`.

MPISHM

The MPISHM implementation uses a hybrid communication method and a hierarchical domain decomposition. Ranks are grouped into a shared memory domain (spanning one socket in the case of MPISHM12 and two sockets for MPISHM24). As in the case of flat MPI, we use a two-dimensional domain decomposition that minimizes the surface to volume ratio and that produces one tile per rank. The tiles in this decomposition are grouped into logical rectangles (*super-tiles*), each of which is assigned to a group of ranks within the same sub-communicator. See Figure 2.1a.

One rank per sub-communicator allocates shared memory to store the data belonging to a super-tile. This memory includes ghost point values at the perimeter of the super-tile for communication with other super-tiles only. There is no explicit communication within a super-tile, only loads and stores, with proper synchronization.

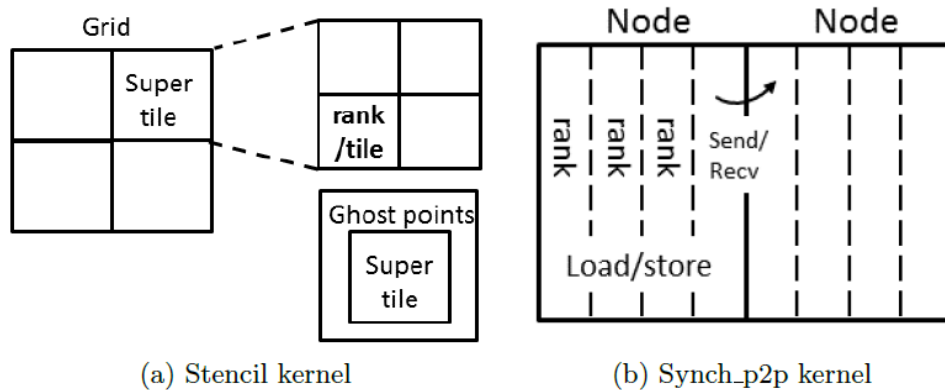


Figure 2.1: Hierarchical domain decomposition for MPISHM

We synchronize via empty messages exchanged between logically nearest neighbors within the super-tile. If neighboring tiles are not in the same shared memory group, standard MPI1 communication is used to exchange ghost point values. This happens on a peer-to-peer basis, in which individual ranks at the boundary of their super-tile exchange messages with their topological neighbors in other super-tiles, unlike in the MPIOPENMP case, where individual threads do not communicate.

MPIRMA

For MPIRMA, we replaced non-blocking Send-Recv pairs with MPI_Put and added MPI_Win_fence synchronization where required (same location as calls to MPI_Wait). We could have instead used the Post-Start-Complete-Wait pattern (PSCW)[53] as in Synchron_p2p, or passive target RMA with atomic counter synchronization as the SHMEM implementation does. These different versions could be implemented and compared in the future.

Charm++

The Charm++ implementation follows effectively the same strategy as MPI1. Virtually the entire program executed by a chare is cast as a single method invocation on the chare, comprising all iterations, including all communications with logically neighboring chares. Data is sent to neighbors asynchronously using messages, a special

kind of RMI, and is received in an SDAG when construct. Such constructs are allowed to block until remote data has arrived. In that case, the runtime can switch to work on another chare, if available, and execute its methods.

Grappa

The Grappa implementation also follows the MPI1 strategy, except that a core responsible for providing ghost point values to its neighbors writes those directly into those neighbors' communication buffers, to be scattered into actual ghost point locations by the receiving core. This bypasses the creation of communication buffers on the sending side, but at the expense of many small remote write operations (the runtime is capable of bundling these transparently, in principle). A single FEB word per buffer is used for synchronization.

SHMEM

The SHMEM implementation also closely follows the MPI1 version. We employ `shmem_putmem` for communication. For synchronization, we use `shmem` atomic increments and `shmem_wait_until` until the required number of ghost point updates has been received.

UPC

The UPC implementation also closely follows the MPI1 version. We create shared ghost zones for each UPC thread. These are filled with neighbor data via `upc_memget`. Initial shared arrays are privatized for optimization. For synchronization, we use `upc_barrier`. We refer to the Berkeley and Cray implementations of UPC as BUPC and CRAYUPC, respectively, in the text and figures.

Performance of Stencil

In Figure 2.2, we show the performance of all runtimes for the Stencil application, divided by the performance of single-node MPI1 and the number of nodes used. The numerals on Charm++ (1, 4, 16) indicate the overdecomposition factor. Those on MPISHM (12, 24), indicate the number of cores in a shared memory group. The graph mostly separates the performance of implementations that rely on global barriers for synchronization and hence do not scale well (MPIRMA, BUPC, CRAYUPC),

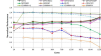


Figure 2.2: Absolute performance of Stencil application, divided by performance of single-node MPI1 and number of nodes used.

and those that use only point to point synchronization between nodes. Except for the highest core counts, performance of the latter is dominated by bandwidth to memory, forcing most of the performance curves together. Grappa has a lower absolute level because of its numerous small direct remote memory accesses. Charm++ uses a somewhat different problem size and hence a different data layout, which affects its absolute level of performance. Its scalability is good, except for an overdecomposition factor of 16, in which case runtime management and communication overhead of the wide ghost zones (4 points on all side of a grid tile) limit performance.

2.2.2 Synch_p2p Implementation Details and Performance

The implementation of Synch_p2p follows the same principles as that of Stencil. Here the data dependencies dictate a 1D domain decomposition, where each rank is responsible for a vertical slice of the grid, see Figure 2.1b. As mentioned in Section 2.1.1, the data dependency is resolved by updating values on only a single grid line segment before communicating one word with the logically neighboring rank. Implementation in MPI1 and Charm++ is obvious and will not be detailed.

MPIOOPENMP

This kernel has no data parallelism, and hence we cannot use the simple fork-join mechanism of `omp parallel for` to parallelize the code among the threads within a rank. Instead we use the method employed, for example, in the OpenMP version of the LU NPB, and documented in [21], for point-to-point synchronization between threads. Because there is no fork-join parallelism, the entire sweep over the grid is within a parallel OpenMP section (`omp parallel`), and threads within different ranks are required to communicate directly. A further complication is that the method for creating a dependence between successive sweeps over the grid involves a communication between

the last thread on the last node, and the first thread on the first node. This necessitates use of `MPI_THREAD_MULTIPLE`, which typically hurts performance.

Grappa

In `Synch_p2p`, all memory is private except for a vector of ghost point values containing, for each core, all the rightmost grid point values that were produced by its predecessor in the pipelined solution process. These are FEB words for efficient synchronization. Their values are stored remotely using the `async write` method, which allows multiple individual writes to be combined automatically by the runtime into a single message, thus reducing latency costs. The programmer is responsible for setting an aggregation target for the runtime when building the code. Our best results were obtained for a bundling factor of 64 words, displayed in Figure 2.3. Larger values of the aggregation factor can reduce latency more, but also increase pipeline fill time, so a non-trivial optimum usually obtains, which is not the same for all numbers of cores. Our optimum value of 64 was chosen to give the best performance at the highest core count for which we ran the Grappa version of the code.

MPISHM

The implementation of MPISHM is very similar to that of MPI1 and Grappa combined; communication between topologically neighboring ranks on different nodes uses blocking `Send-Recv`. Data exchange between neighboring ranks on the same node takes place via reads from the neighbor's shared memory window, supplemented with point-to-point synchronization via empty messages.

MPIRMA

The MPIRMA implementation of `Synch_p2p` is identical in form to MPI1, wherein `Send-Recv` pairs are replaced with the `PSCW` pattern. Data is transferred with `MPI_Put`. As there is essentially no semantic difference between MPI1 and MPIRMA for this kernel, we attribute any performance differences to implementation issues.

SHMEM

The SHMEM Synchron_p2p implementation features two synchronization protocols, termed *handshake* and *no handshake*. The latter allows for relaxed synchronization, established via one fine-grain semaphore per grid line segment[139]. We only show results for this protocol, since it performs best.

UPC

The UPC Synchron_p2p implementation has two variants. One is portable, using a shared flag for communication between neighboring ranks. The other is more optimized, but non-portable, using extensions in the Berkeley UPC compiler[18]. That version, denoted by *BUPCsem*, utilizes semaphores for synchronization within an iteration. However, both versions synchronize via a global barrier after each iteration.

Performance of Synchron_p2p

Figure 2.3 shows performance of Synchron_p2p. The results are dominated by two performance artifacts: efficiency with which runtimes can handle very frequent point-to-point synchronization, and the cost of doing global synchronization after each iteration. MPIRMA and both versions of UPC have barriers after each iteration, and clearly suffer the performance consequences at higher core counts. MPI1, SHMEM, Grappa, and MPISHM all function effectively the same, and their relative performances are close together (Grappa’s scalability suffers beyond 64 nodes due to the current policy in the aggregator: it uses atomic operations to poll for messages to aggregate between cores in a node, which wastes significant inter-socket bandwidth at scale for problems with communication locality; new policies need to be added to serve sparse workloads like Synchron_p2p). The Charm++ runtime is designed for workloads that require flexibility, resilience, and dynamic load balancing. The overhead of such functionality results in higher latencies than for the other runtimes, and that effect is exaggerated by Synchron_p2p. Finally, MPIOPENMP suffers from communication serialization within a rank due to the need for different threads within a rank to be able to communicate safely.

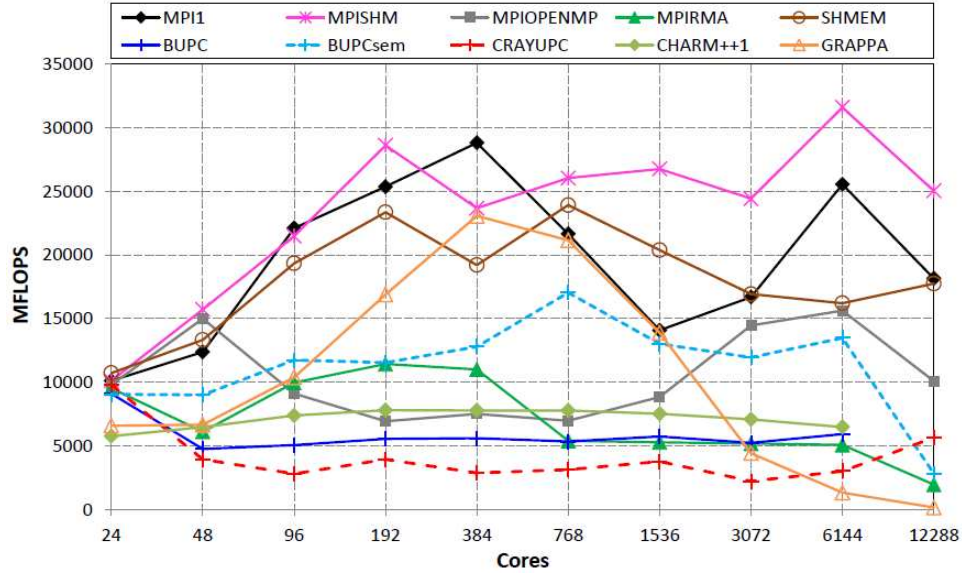


Figure 2.3: Absolute Performance of Synchron_p2p kernel

2.2.3 Transpose Implementation Details and Performance

Transpose requires all-to-all communication, wherein each processing element scatters pieces of its matrix columns to all other processing elements.

MPI1

This global exchange could be accomplished using `MPI_Alltoall`, but since the tiles of B are not contiguous in memory, and also need to be transposed locally, this would require a tripling of the memory footprint for matrix B to fit message buffers. Instead, we perform the transpose on N ranks in $N - 1$ communication phases. In each phase, each rank sends a different tile to a different recipient rank, using a conflict-free schedule that accomplishes a pairwise exchange. Message buffers are reused in each phase, which keeps the memory pressure small and improves locality. Local transposition of B tiles takes place as matrix elements are read from B and placed in the contiguous send buffer. Upon receipt of a tile, its data is scattered into the columns of A owned by the receiving rank, but no more transposition is required.

Charm++

This implementation exactly follows MPI1’s.

MPIOOPENMP and MPISHM

In the MPIOOPENMP implementation, we parallelize the loops that transpose, pack, and unpack the tile values using `#pragma omp parallel for`, but the MPI communications are carried out only by the master thread.

The MPISHM implementation closely follows MPIOOPENMP. All ranks inside a node collaborate to transpose, pack, and unpack the tile values, and only a single rank within each node carries out the communication. We could have chosen to let all ranks participate in the communication, but then MPISHM would be conceptually identical to MPI1. Instead, ranks within a node synchronize before send and after receive so that a single rank can perform inter-node communication.

Grappa

Grappa communicates at the granularity of block rows directly into the destination matrix, so does not need a receive buffer, but issues many small asynchronous communication requests for the transport of column segments within each matrix tile. Synchronization at the end of iterations is via barriers.

UPC

The UPC implementation follows the MPI1 approach. However, tiles are communicated directly into the private space of each thread using `upc_memget`. For synchronization, we use `upc_barrier`.

Performance of Transpose

Figure 2.4 shows performance of the Transpose kernel. At smaller core counts, it constitutes a coarse-grain workload. The message sizes decrease quadratically with the number of cores used; on 1536 cores, these are 4.5 MB for 1 communicating rank per node (CRPN), 1.25 MB for 2 CRPN, and 8 KB for the “flat” models (i.e. those with 24 CRPN), which is in the bandwidth-limited regime of the network. At 12288 cores, the message sizes associated with these models decrease to 72 KB, 16 KB, and 128 bytes, respectively. While 1 and 2 CRPN still use large messages, the other cases have become fine-grain. While Cray XC30 was designed to support fine-grain communication, it is

still difficult to provide the same level of efficiency, and overheads are noticeable. This fact alone explains the most obvious trend in the data: only MPI+X models scale past 1536 cores. The figure shows data for MPISHM2 (2 CRPN, one on each socket), MPISHM24 (1 CRPN), and MPIOPENMP (2 CRPN for all cases except 12288 cores, which used 1 CRPN). All the flat models taper off. Of the flat models, SHMEM is clearly the best at scale, in addition to being the best overall at 1536 cores and below. We attribute this to the low overhead of the SHMEM interface and the excellent support for SHMEM one-sided semantics in the Cray XC30 network. MPI1 is slightly worse than SHMEM, but follows a similar trend. Cray and Berkeley UPC also share a trend and flatten around 1536, reaching performance similar to the other flat models at high core counts. MPIRMA with flush synchronization, which is a similar semantic to the PGAS models, performs roughly on par with Cray UPC. MPIRMA with fence synchronization (collective) shows an anomaly at 12288 cores, which is hard to explain without detailed information about Cray MPI (we do not have it). It is possible that message aggregation permitted by the semantics of this synchronization motif becomes active at 128 bytes. Charm++ is competitive up to 768 cores, at which point it falls off and performs worst at scale, due to runtime overhead at fine granularities (cf. `Synch_p2p`). We evaluated Charm++ *with* overdecomposition, but it was no better than *without*, as expected, so we excluded the former from the figure for brevity. The performance of Grappa is poor, because its Transpose implementation is even finer grain than the other flat models.

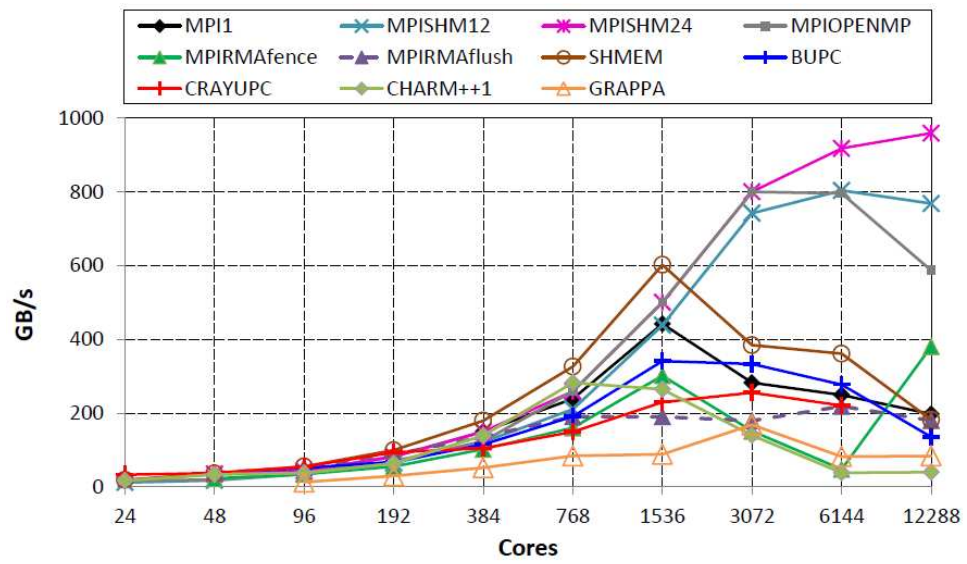


Figure 2.4: Absolute performance of Transpose kernel

Chapter 3

MLPERF TRAINING BENCHMARK

Machine learning (ML) has revolutionized numerous domains, including computer vision[89], language processing[42, 116], speech recognition[69], and gaming[124, 104, 25]. Much of this progress owes to deep learning (DL), which involves training of large deep neural network (DNN) models on massive data sets. To keep up with this growing computational demand, hardware and software systems have garnered sizable investments[4].

As the number of hardware and software systems for DL training increases[112, 1, 27, 79, 80, 29, 96, 77], so does the need for a comprehensive benchmark. History shows that benchmarks accelerate progress[67]. For example, breakthroughs in microprocessor and relational database systems in the 1980s inspired industry consortiums to create *Standard Performance Evaluation Corporation* (SPEC) for Unix servers[44] and the *Transaction Processing Performance Council* (TPC) for transaction processing and databases[40]. These organizations helped develop and maintain benchmarks that their respective communities then embraced. Their success inspired the formation of MLPerf, a consortium of commercial and academic organizations, to design a comprehensive benchmark suite for deep learning.

Unlike other computational workloads, deep learning allows a range of statistical, hardware, and software optimizations that can change the mathematical semantics of the underlying operators. Although these optimizations can boost *performance* (i.e. training speed), some change the learning dynamics and affect the final model’s *quality* (i.e. accuracy). Even accommodating different system scales (e.g., varying the number of chips) requires changing the hyperparameters, potentially affecting the amount of

computation necessary to reach a particular quality target. By contrast, other compute benchmarks can evaluate systems through targeted benchmarks.

Deep learning is also intrinsically approximate and stochastic, allowing multiple equally correct solutions - unlike conventional computing, which tends to allow just one correct solution. As a result, implementations and training times can vary while the final quality remains the same. Since it is approximate, deep learning requires careful definition of equally valid solution classes and the appropriate degrees of freedom.

Prior work has varied in granularity, but has either left the above challenges unaddressed or lacked critical workloads representative of modern machine learning. Microbenchmarks such DeepBench[9] are affordable to run and enable a fair comparison of competing systems by isolating hardware and software from statistical optimizations, but they fail to reflect the complexity of real workloads and have limited utility. Although throughput benchmarks like Fathom and TBD[2, 151, 56] evaluate full model architectures across a broad range of tasks to better reflect the diversity and complexity of real workloads, they limit model architecture and training innovations that advance the state-of-the-art. DAWNbench[37] measures end-to-end training time, subject to a quality threshold (i.e. time to train), and it accommodates innovative solutions (i.e. new model architectures and training techniques, such as progressive resizing and cyclic learning rates). It additionally collects source code to promote reproducibility. DAWNbench’s flexibility, however, also made it difficult to draw fair comparisons between hardware and software platforms. MLPerf builds on the strengths of prior work; it combines a broad set of benchmarks like Fathom or TBD, an end-to-end training metric like DAWNbench, and the backing of a broad consortium like SPEC.

MLPerf aims to create a representative benchmark suite for machine learning that fairly evaluates system performance to meet five high level goals.

- Enable fair comparison of competing systems while still encouraging machine learning innovation.
- Accelerate machine learning progress through fair and useful measurement.
- Enforce reproducibility to ensure reliable results.

- Serve both the commercial and research communities.
- Keep benchmarking effort affordable so all can participate.

Although prior machine learning benchmarking efforts[37, 2, 56, 9, 151] each contributed to meeting one or more of the above goals, we created MLPerf to address *all* of them holistically, building on the lessons learned from these efforts. To this end, MLPerf Training[97] does the following:

- Establish a comprehensive benchmark suite that covers diverse applications, DNN models, and optimizers.
- Create reference implementations of each benchmark to precisely define models and training procedures.
- Establish rules that ensure submissions are equivalent to these reference implementations and use equivalent hyperparameters.
- Establish timing rules to minimize the effects of stochasticity when comparing results.
- Make submission code open source so that the machine learning and systems communities can study and replicate the results.
- Form working groups to keep the benchmark suite up to date.

3.1 Background

3.1.1 Unique Challenges of Benchmark Training

ML benchmarking faces unique challenges relative to other compute benchmarks, such as LINPACK[45] and SPEC[44], that necessitate an end-to-end approach. After an ML practitioner selects a data set, optimizer, and DNN model, the system trains the model to its state-of-the-art *quality* (e.g., Top-1 accuracy for image classification). Provided the system meets this requirement, the practitioner can make different operation, implementation, and numerical representation choices to maximize system *performance* - that is, how fast the training executes. Thus an ML performance benchmark must ensure that systems under test achieve state-of-the-art quality while providing sufficient flexibility to accommodate different implementations. This tradeoff

between quality and performance is challenging because multiple factors affect both the final quality and the time to achieve it.

3.1.1.1 Effect of Optimizations on Quality

Although many optimizations immediately improve traditional performance metrics such as throughput, some can decrease the final model quality, an effect that is only observable by running an entire training session. For example, the accuracy difference between single-precision training and lower-precision training only emerges in later epochs[150]. Across several representation and training choices, the validation-error curves may only separate after tens of epochs, and some numerical representations never match the final validation error of full-precision training (lower validation error directly corresponds to higher accuracy: $accuracy = 1 - error_{validation}$). Thus, even though microbenchmarks[9, 31] can assess an optimization’s performance impact, a complete training session is necessary to determine the quality impact and whether the model achieves the desired accuracy. Owing to the introduction of systems with varying numerics[1, 11, 87, 100] and performance optimizations, ML benchmarks must include accuracy metrics.

3.1.1.2 Effect of Scale on Time to Train

ML training on large distributed systems with many processors typically involves data parallelism and large minibatches to maximize system utilization and minimize training time. In turn, these large minibatches require adjustments to optimizer parameters, such as the learning rate[88, 58]. Together, these changes affect the learning dynamics and can alter the number of iterations required to achieve the target accuracy. For example, MLPerf v0.5 ResNet-50 takes about 64 epochs to reach the target Top-1 accuracy of 74.9% at a minibatch size of 4K¹, whereas a minibatch size of 16K can require more than 80 epochs to reach the same accuracy, increasing computation by 30%. Larger minibatches, however, permit efficient scaling to larger distributed

¹ MLPerf v0.5 results <https://mlperf.org/training-results-0-5>

systems, reducing the time to train the model. The tradeoffs between system size, minibatch size, and learning dynamics present another challenge for a DL-focused performance benchmark.

3.1.1.3 Run-to-Run Variation

DNN training involves many stochastic influences that manifest in substantial run-to-run variation [33, 57, 6, 36]. Different training sessions for the same model using the same hyperparameters can yield slightly different accuracies after a fixed number of epochs. Alternatively, different training sessions can take a different number of epochs to reach a given target accuracy. For example, Figure 3.1 shows the number of epochs needed to reach target accuracy for two MLPerf v0.5 benchmarks using reference implementations and default batch sizes. Several factors contribute to this variation, such as application behavior (e.g., random weight initialization and random data traversal) and system characteristics (e.g., profile-driven algorithm selection and the non-commutative nature of floating point addition). Large distributed training tasks can involve asynchronous updates, altering the gradient accumulation order. These variations make it hard to reliably compare system performance.

3.1.1.4 Diverse Software

Multiple ML software frameworks have emerged, each of which executes similar but distinct computations owing to various implementations and constraints [1, 112, 27, 79]. Software frameworks and the underlying math libraries employ different algorithms to implement the same operation. For example, convolutional and fully connected layers - two compute-intensive operators prevalent in modern DNN models - typically use cache blocking to exploit processor memory hierarchies. Different block sizes and processing orders (which optimize for different hardware), although algebraically equivalent, yield slightly divergent results. In addition, operators can execute using various algorithms. For example, convolution layers can be executed using a variety of algorithms, including GEMM-based and transform-based (e.g., FFT or

Winograd) variants. In fact, the cuDNN v7.6 library provides roughly 10 algorithms for the forward pass of a convolutional layer², some of which vary in tiling or blocking choices depending on the hardware. Although mathematically equivalent, different implementations will produce different numerical results, as floating point representations have finite precision.

Additionally, frameworks occasionally implement the same function in mathematically different ways. For example, modern training frameworks implement stochastic gradient descent with momentum in two ways.

The Caffe framework[79] implements the first approach, whereas PyTorch[112] and TensorFlow[1] implement the second. These approaches differ mathematically if the learning rate η changes during training - a common technique. Although this difference is tiny in many cases, it can hinder training convergence for larger minibatches.

Variations also arise owing to the frameworks' programming interface. For example, PyTorch and TensorFlow interpret asymmetric padding differently, complicating the task of porting model weights between them. Data augmentation pipelines across frameworks can also apply image augmentations (e.g., crop, zoom, and rotation) in different orders.

Although ONNX[8], TVM[29], and similar emerging tools enable interoperability of model architectures across frameworks, their support remains limited. Moreover, ML systems involve a range of optimizations that extend beyond the model architecture, such as pre-processing, precision, and communication methods. Benchmarks must accommodate the wide diversity of deployed systems despite this lack of a standard way to specify every training aspect.

3.2 MLPerf Training Benchmark

We now present the MLPerf Training benchmark, detailing the workloads3.2.1, timing rules3.2.2, quality-threshold choices 3.2.3, and reference implementations and hyperparameters3.2.4.

² cuDNN <https://docs.nvidia.com/deeplearning/sdk/cudnn-developer-guide>

3.2.1 Benchmark Suite

To create a fair and useful benchmark suite for modern ML workloads, we curated a representative set of tasks from several major ML areas, including vision, language, recommendation, and reinforcement learning. Our selection of benchmarks was primarily based on commercial and research relevance, representing diverse compute motifs. To establish relevance, we relied on feedback from the tens of commercial and academic organizations that support MLPerf. To keep the suite affordable, we selected a compact but representative set of seven benchmarks, which we describe below and summarize in Figure 3.3. Although these benchmarks already cover a wide range of research and industrial tasks, we are continuously exploring additional ones to keep the suite relevant to the ML community.

3.2.1.1 Image Classification

Image classification is the most important task for evaluating ML system performance [37, 2, 151, 58, 78, 101, 145, 56, 106]. A classifier selects a class that best describes the contents of a given image. Classification model architectures also serve as feature extractors for many other computer vision workloads, including object detection, captioning, and style transfer. We use the ILSVRC 2012 ImageNet classification data set, consisting of 1.28 million training images and 50,000 validation images [41]. Our model quality metric is the Top-1 accuracy on the validation set.

ResNet-50 is a residual network [64, 65]; such networks and their derivatives remain the state of the art in image classification, and system studies commonly use them [58, 78, 101, 145, 127]. Several slightly different ResNet-50 implementations appear in training framework repositories, preventing comparison of earlier system performance claims because of model differences. To ensure meaningful system comparison, MLPerf uses the ResNet-50 v1.5 model, which performs addition after batch normalization, omits 1 x 1 convolution from the skip connection of the first residual block, and applies downsampling by the 3 x 3 convolutions. MLPerf also specifies the appropriate parameter initialization, optimizer schedule, and data augmentation.

3.2.1.2 Object Detection and Segmentation

Object detection and segmentation are crucial components of many industrial systems for robotics, autonomous driving, video analytics, and social networks. Object detection is a regression task as opposed to a classification task: it returns bounding-box coordinates for objects in a given image. Segmentation assigns an object class to each input-image pixel. Although pre-trained image classification models commonly serve as the backbone (feature extractor) for DNN object detectors and segmenters, these DNN tasks differ from image classification in their compute characteristics. Examples include additional layer types (upsampling, ROIalign, NMS, and sorting); moreover, the inputs have greater resolution. MLPerf uses the 2017 COCO data set[94] consisting of 118,000 training images and 5,000 validation images. Model quality measurement uses mAP for both detection and segmentation.

Mask R-CNN[63] is a popular object detection and instance segmentation model for images. It has two stages: the first proposes regions of interest, and the second processes them to compute bounding boxes and segmentation masks. Mask R-CNN provides high accuracy results for these tasks, but at the cost of higher latency as well as greater compute and memory requirements. The training benchmark uses images resized to 800 pixels on the shorter side and employs ResNet-50 as the backbone.

Single Shot Detection (SSD)[95] serves in real-time applications that require low-latency solutions. These applications include autonomous driving, robotics, and video analytics. Compared with Mask R-CNN[73] and other two-stage solutions, SSD trades speed for accuracy. Instead of full images, training uses 300 x 300 crops. We chose a ResNet-34 backbone to represent current real-time applications. ResNet-34 has a different residual structure than ResNet-50, increasing the diversity of computational motifs that MLPerf covers.

3.2.1.3 Translation

Neural machine translation converts a sequence of words from a source language to a target language; many industrial applications use this technology. As is common

in translation research, we use the WMG English-to-German (EN-DE) data set[141], which contains about 4.5 million sentence pairs. Our model quality metric is the Bilingual Evaluation Understudy Score (BLEU) score on the Newstest2014 test set. We include two translation benchmarks to account for the two model architectures that translation and other sequence-data tasks often employ.

Transformer[135] is an attention-based model that achieves state-of-the-art language-translation quality. It consists of an encoder and decoder, each being a stack of six blocks. Every block comprises a multi-head attention layer and point-wise fully connected layers.

GNMT[142] is a recurrent neural network (RNN) for language translation. Even though it achieves lower accuracy than Transformer on the WMT English-to-German data set, it appears in the suite to represent RNN applications. These applications span numerous tasks, but language-translation data sets and publications are more common, enabling clearer system comparison. GNMT is the suite’s only RNN. It consists of an eight-layer encoder and an eight-layer decoder, each using 1,024 LSTM cells with skip connections.

3.2.1.4 Reinforcement Learning

Reinforcement learning (RL) is responsible for the recent dramatic increase in compute demand[4], and it serves in control systems. RL algorithms can train agents (which include neural networks) that rival humans at video games, go, and chess - major milestones in machine learning[124, 104, 25]. RL has a different computational profile than the other ML benchmarks: it generates training data through exploration instead of relying on a predetermined dataset.

MiniGo[102], inspired by AlphaGo[123, 125, 124], trains a single model that represents both value and policy functions for a 9 x 9 game board. Training uses self-play (simulated games) between agents to generate data; rather than using a simulator, it performs many forward passes through the model to generate actions. We chose MiniGo to keep MLPerf more ML-oriented, since many other RL problems employ

simulators (physics, video game environments, etc.) to generate data, spending most of their time in computations unrelated to ML. To measure quality, we calculate the percentage of predicted moves that match human reference games.

3.2.1.5 Recommendation

Recommendation systems are a major commercial workload for Internet companies[107, 149, 30]. These workloads are characterized by large embedding tables followed by linear layers.

Neural collaborative filter (NCF)[66] was our choice for the benchmark. It is trained to predict user-item interactions. More so than for other tasks, this recommender’s compute characteristics depend on the data set. For example, the data set defines the embedding table size as well as the memory access patterns. Thus, a representative data set is crucial to a representative benchmark. Unfortunately, however, public data sets tend to be orders of magnitude smaller than industrial data sets. Although MLPerf v0.5 adopted the MovieLens-20M data set[60] for its NCF benchmark v0.5 will employ a synthetically generated data set and benchmark while retaining the characteristics of the original data[14].

3.2.2 Time-to-Train Performance Metric

To address the ML benchmarking challenges of system optimization and scale that we outlined in Sections 3.1.1.1 and 3.1.1.2, MLPerf’s performance metric is the time to train to a defined quality target. It incorporates both system speed and accuracy and is most relevant to ML practitioners. As an end-to-end metric, it also captures the auxiliary operations necessary for training such models, including data pipeline and accuracy calculations. The metric’s generality enables application to reinforcement learning, unsupervised learning, generative adversarial networks, and other training schemes. Time to train overcomes the challenges in Sections 3.1.1.1 and 3.1.1.2 by preventing submissions from using quality-reducing optimizations while still allowing for extensive system-scale and software-environment flexibility.

3.2.2.1 Timing Rules

We chose the timing requirements to ensure fair system comparisons and to represent various training use cases. Timing begins when the system touches any training or validation data, and it stops when the system achieves the defined quality target on the validation data set.

We exclude from timing several components that can carry substantial overhead and that are unrepresentative of real-world differences.

System initialization

Initialization, especially at large scale, varies on the basis of cluster-administrator choices and system-queue load. For example, it may involve running diagnostics on each node before starting the training job. Such overheads are unindicative of a system’s training capability, so we exclude them from timing.

Model creation and initialization

Some frameworks can compile the model graph to optimize subsequent execution. This compilation time is insignificant for the longer training sessions when using industry-scale data sets. MLPerf, however, uses public data sets that are usually much smaller than industry ones. Therefore, large distributed systems can train some MLPerf benchmark in minutes, making compilation times a substantial portion of the total time. To make benchmarks representative of training on the largest industrial data sets, we allow exclusion of up to 20 minutes of model creation time. This limit ensures that MLPerf captures smaller training jobs, and it discourages submissions with compilation approaches that are too computationally and operationally expensive to use in practice.

Data reformatting

The raw input data commonly undergoes reformatting once and then serves in many subsequent training sessions. Reformatting examples include changing image-file formats and creating a database (e.g., LMDB, TFRecords, or RecordIO) for more efficient access. Because these operations execute once for many training sessions, MLPerf timing excludes reformatting. However, it prohibits any data processing or

augmentation that occurs in training from moving to the reformatting stage (e.g., it prevents different crops of each image from being created and saved before the timed training stage).

3.2.2.2 Number of Timing Runs

To address the stochastic nature and resulting run-to-run variance of modern deep learning methods described in Section 3.1.1.3, MLPerf requires that submissions provide several runs of each benchmark to stabilize timing. We determined the number of runs, which varies among benchmarks, by studying the behavior of reference implementations. Vision tasks require 5 runs to ensure 90% of entries from the same system are within 5%; all other tasks require 10 runs to ensure 90% of entries from the same system are within 10%. MLPerf drops the fastest and slowest times, reporting the arithmetic mean of the remaining runs as the result.

3.2.3 Choice of Quality Thresholds

For each benchmark, we chose quality metrics near the state of the art for the corresponding model and data set, basing our choice on experiments with the reference implementations. Some of these thresholds are slightly lower than results in the literature, enabling us to benchmark across software frameworks and to ensure that training sessions consistently achieve the quality metric. Although selecting a lower threshold that is achievable earlier in a training session reduces submission resources, we chose higher thresholds that require longer training sessions for two reasons. First, we must prevent optimizations from adversely affecting the final results (challenges described in Sections 3.1.1.1 and 3.1.1.2). Second, we must minimize run-to-run variation, which tends to be much higher early in training. For example, Figure 3.4 shows accuracy for five training sessions of MLPerf v0.5’s ResNet-50 v1.5 reference implementation, where the first 30 epochs exhibit considerably more noise.

3.2.4 References and Hyperparameters

MLPerf provides a reference implementation for each benchmark, using either the PyTorch or TensorFlow framework. References also include scripts or directions to download and pre-process public data sets. References are not optimized for performance (meaning they should not be used for performance assessment or comparison), as their main purpose is to define a concrete implementation of a benchmark model and training procedure. All submitters must follow these references - they may re-implement a benchmark of their choice as long as the DNN model and training operations are mathematically equivalent to the reference. Furthermore, MLPerf uses reference implementations to establish the required quality thresholds.

MLPerf rules specify the modifiable hyperparameters (Figure 3.5), as well as restrictions on their modification. These restrictions are intended to balance the need to tune for different systems with limiting the size of the hyperparameter search space to be fair to submitters with smaller compute resources. For example, to accommodate a wide range of training system scales, submissions must be able to adjust to minibatch size used by SGD in order to showcase maximum system efficiency (this approach is similar in concept to the Top500 LINPACK benchmark, which allows systems to choose the problem size). To ensure that training still converges to the required threshold, other hyperparameters - such as the learning rate schedule - may need adjustment to match. For example, a common ResNet training practice is to increase the learning rate linearly with the minibatch size[58]. Although these hyperparameter searches are a common ML task, MLPerf’s focus is on system optimization, rather than hyperparameter exploration and we do not want to penalize submitters who are unable to do extensive searches. Therefore, we restrict hyperparameter tuning to a subset of all possible parameters and values.

Further, we allow “hyperparameter borrowing” during the post-submission review process in which one submitter may adopt another submitter’s hyperparameters for a specific benchmark and re-submit their result (with no other hardware or software changes allowed). In the first two rounds, hyperparameter borrowing was used

successfully to improve several submissions, indicating hyperparameters are somewhat portable. Typically, borrowing occurred across systems of similar scale, but did result in convergence across different numerics (FP16, bfloat16, and FP32), architectures (CPU, GPU, and TPU), and software implementations (TF, cuDNN, and MKL-DNN). MLPerf working groups review the hyperparameter choices and requirements for each benchmark round to account for advances in training ML models at scale.

3.3 Benchmarking Process

Next, we outline the process for submission and review and for reporting results to account for innovative solutions, availability, and scale. We have run two rounds of the MLPerf benchmark: v0.5 and v0.6. The time between rounds is about a few months, allowing us to update the benchmark suite after each one. Every round has a submission and review period followed by publication of results.

3.3.1 Submission and Review

An MLPerf submission consists of a system description, training session log files, and all code and libraries required to reproduce the training sessions. All of this information is publicly available on the MLPerf GitHub site, along with the MLPerf results, allowing for reproducibility and enabling the community to improve the results in subsequent rounds. A system description includes both the hardware (number of nodes, processor and accelerator counts and types, storage per node, and network interconnect) and the software (operating system, as well as libraries and their versions). A training session log file contains a variety of structured information including time stamps for important workload stages, quality metric evaluations at prescribed intervals, and hyperparameter choices. These logs are the foundation for analyzing results.

Before publishing results, submissions are peer-reviewed for compliance with MLPerf rules. Submitters receive notification of noncompliance, where applicable, and

they may resubmit after addressing any such problems. Additionally, we permit some hyperparameter borrowing as described earlier during this period.

3.3.2 Reporting Results

Each MLPerf submission has several labels: division (open or closed), category (available, preview, or research), and system type (on-premises or cloud).

3.3.2.1 Submission Divisions

MLPerf has two submission divisions: closed and open. Both require that submissions employ the same data set and quality metric as the corresponding reference implementation.

The *closed* division is intended for direct system comparison, so it strives to ensure workload equivalence by requiring that submissions be equivalent to reference implementations. Equivalence includes mathematically identical model implementations, parameter initialization, optimizer and training schedules, and data processing and traversal. To ensure fairness, this division also restricts hyperparameter modification.

The *open* division is intended to encourage innovative solutions of important practical problems and to encourage hardware/software co-design. It allows submissions to employ model architectures, optimization procedures, and data augmentations that differ from the reference implementations.

3.3.2.2 System Categories

To allow for a broad range of research and industry systems, we defined three submission categories: available, preview, and research. These categories encourage novel techniques and systems (e.g., from academic researchers), but they also distinguish between shipping products and proof-of-concept or early engineering samples.

The *available* category imposes requirements on both hardware and software availability. Hardware must be either available for third-party rental on a cloud service or, in the case of on-premises equipment, available for purchase. Supply and lead times

for renting or purchasing should benefit the system scale and company size. To ensure that benchmark submissions are widely consumable and to discourage benchmark-specific engineering, we also require that software in this category be versioned and supported for general use.

Preview systems contain components that meet the available category criteria within 60 days of the submission date or by the next submission cycle, whichever is later. Any preview system must also be submitted to the available category by that time.

Research submissions contain components unintended for production. An example is an academic research prototype designed as a proof of concept, rather than a robust product. This category also includes systems that are built from production hardware and software but are larger in scale than available category configurations.

3.3.2.3 Reporting Scale

Modern ML training spans multiple orders of magnitude in system power draw and cost. Therefore, comparisons are more useful if the reported performance includes the scale. A common scale metric, such as cost or power, is not definable across a wide range of systems (cloud, on-premises, and pre-production), so it requires differentiation by system type.

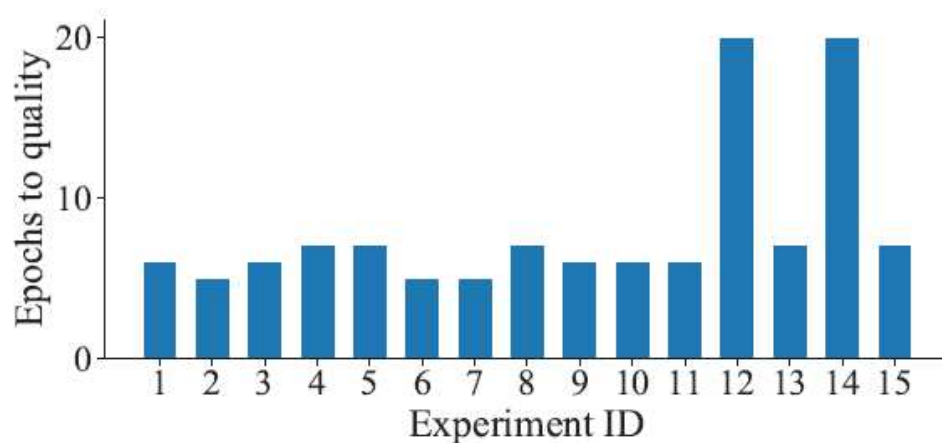
In the first two MLPerf rounds, we included the system configuration (number of processors and/or accelerators) alongside the performance scores. For on-premises examples, future versions will include a power management specification. For cloud systems, we derived a “cloud-scale” metric from the number of host processors, amount of host memory, and number and type of accelerators. We empirically verified that cloud scale correlates closely with cost across three major cloud providers. Reporting of these scale metrics was optional in MLPerf v0.5 and v0.6.

3.3.2.4 Reporting Scores

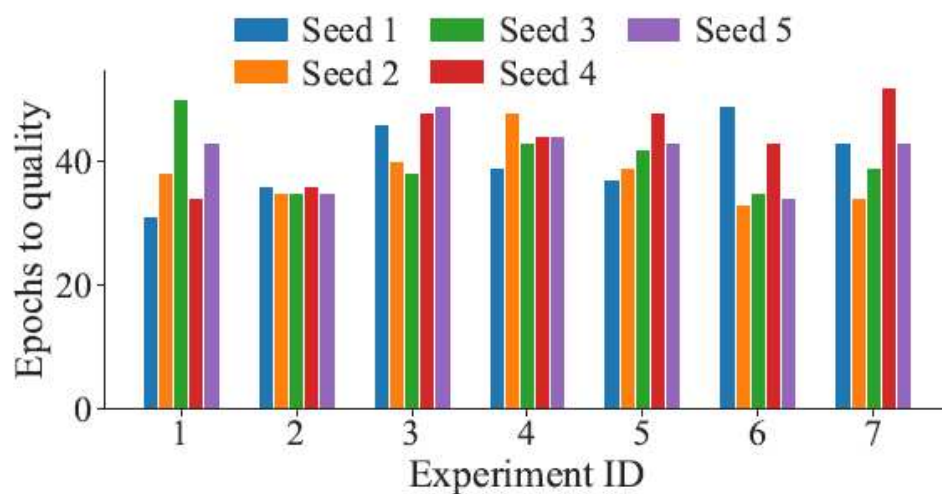
An MLPerf results report provides the time to train for each benchmark. Although a single summary score that spans the entire suite may be desirable for system comparisons, it is unsuited for MLPerf for two main reasons. First, a summary score implies some weighting of individual benchmark scores. Given the diversity of system users and the wide range of applications that MLPerf covers, no weighting scheme is universally representative. Second, a summary score becomes less meaningful if a submitter declines to report results on all benchmarks. Submitters can have multiple reasons for omitting some benchmarks - not all are practical at every system scale (for example, some models are untrainable at the minibatch sizes that the largest systems require for data parallel training). Additionally, some processors may target only certain applications.

3.4 Results

MLPerf, like all benchmarks, aims to encourage innovation through constructive competition; we measure progress by comparing results across submission rounds. We have conducted two MLPerf training rounds thus far, v0.5 and v0.6. They were six months apart, and the underlying hardware systems were unchanged. The results that were either unmodified or underwent minor modifications between rounds show that MLPerf is driving rapid performance and scaling improvement in both the implementations and software stacks. Figure 3.6 shows that between the two submission rounds, the best performance results for a 16-chip system increased by an average 1.3x despite the higher quality target. Figure 3.7 reveals that the number of chips necessary to produce the best overall performance result increased by an average of 5.5x. Some of this improvement owes to better benchmark implementations and some to rule changes, such as allowing the LARS[146] optimizer for large ResNet batches. However, we believe submitters incorporated much of the performance and scaling improvements into the underlying software infrastructure and passed them on to users. We expect MLPerf to drive similar improvements through focused hardware innovation.



(a) NCF.



(b) MiniGo.

Figure 3.1: Training epochs to reach the target quality for the MLPerf v0.5 NCF (a) and MiniGo (b) benchmarks. Each experiment uses identical hyperparameters except for the random seed. For MiniGo, we observed considerable variability across runs even when fixing the random seed (same color).

$$\text{momentum} = \alpha \cdot \text{momentum} + \eta \cdot \frac{\partial L}{\partial w} \quad (1)$$

$$w = w - \text{momentum}$$

$$\text{momentum} = \alpha \cdot \text{momentum} + \frac{\partial L}{\partial w} \quad (2)$$

$$w = w - \eta \cdot \text{momentum}$$

Figure 3.2:

Benchmark	Data set	Model	Quality Threshold
Image classification	ImageNet (Deng et al., 2009)	ResNet-50 v1.5 (MLPerf, 2019b)	74.9% Top-1 accuracy
Object detection (lightweight)	COCO 2017 (Lin et al., 2014)	SSD-ResNet-34 (Liu et al., 2016)	21.2 mAP
Instance segmentation and object detection (heavyweight)	COCO 2017 (Lin et al., 2014)	Mask R-CNN (He et al., 2017a)	37.7 Box min AP, 33.9 Mask min AP
Translation (recurrent)	WMT16 EN-DE (WMT, 2016)	GNMT (Wu et al., 2016)	21.8 Sacre BLEU
Translation (nonrecurrent)	WMT17 EN-DE (WMT, 2017)	Transformer (Vaswani et al., 2017)	25.0 BLEU
Recommendation	MovieLens-20M (GroupLens, 2016)	NCF (He et al., 2017b)	0.635 HR@10
Reinforcement learning	Go (9x9 Board)	MiniGo (MLPerf, 2019a)	40.0% Professional move prediction

Figure 3.3: MLPerf Training v0.5 benchmarks

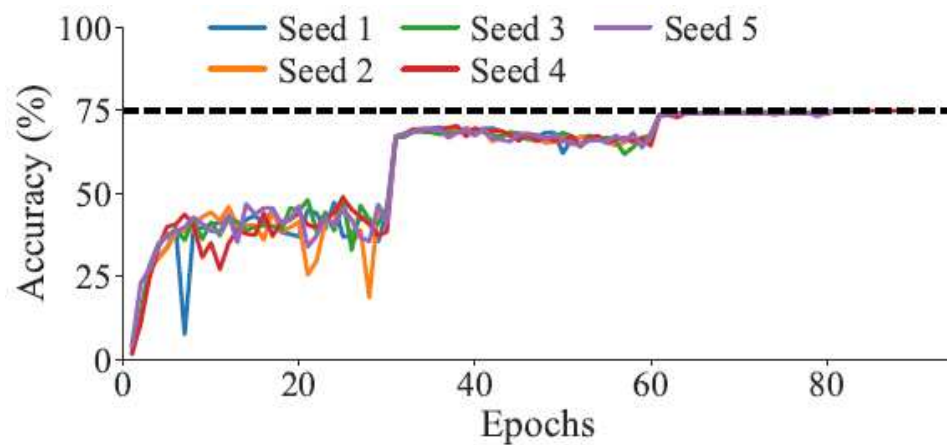
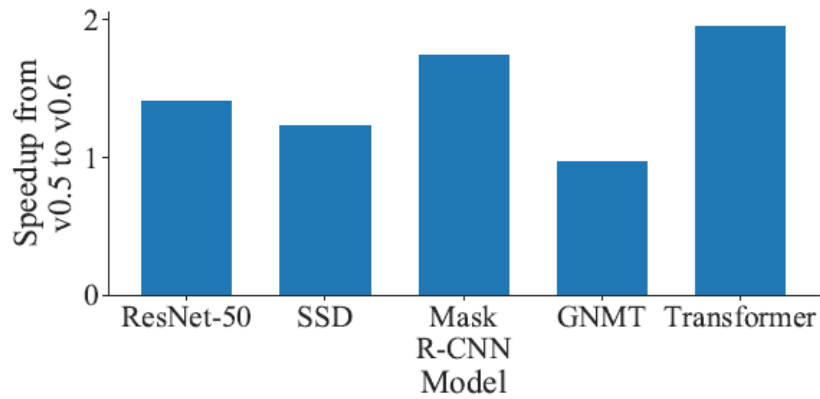


Figure 3.4: Top-1 accuracy of MLPerf v0.5 ResNet-50 benchmark over 100 epochs for five runs (denoted by color) with identical hyperparameters but different random seeds. The dashed line indicates the quality target of 74.9% Top-1 accuracy. The early training phase exhibits much more variability than later phases.

Model	Modifiable Hyperparameters
All that use SGD	Batch size, Learning-rate schedule parameters
ResNet-50 v1.5	
SSD-ResNet-34	Maximum samples per training patch
Mask R-CNN	Number of image candidates
GNMT	Learning-rate decay function, Learning rate, Decay start, Decay interval, Warmup function, Warmup steps
Transformer	Optimizer: Adam (Kingma & Ba, 2015) or Lazy Adam, Learning rate, Warmup steps
NCF	Optimizer: Adam or Lazy Adam, Learning rate, β_1 , β_2
Go (9x9 board)	

Figure 3.5: MLPerf modifiable hyperparameters



(a) Speedup.

Model	Metric	v0.5	v0.6
ResNet-50	Top-1 accuracy	74.9	75.9
SSD	mAP	21.2	23
Mask R-CNN	Box / Mask min AP	37.7 / 39.9	Same
GNMT	Sacre BLEU	21.8	24
Transformer	BLEU	25	Same

(b) Quality targets.

Figure 3.6: Speedup in the fastest 16-chip entry from MLPerf version v0.5 to v0.6 for various benchmarks common to both (a) along with quality target increases (b).

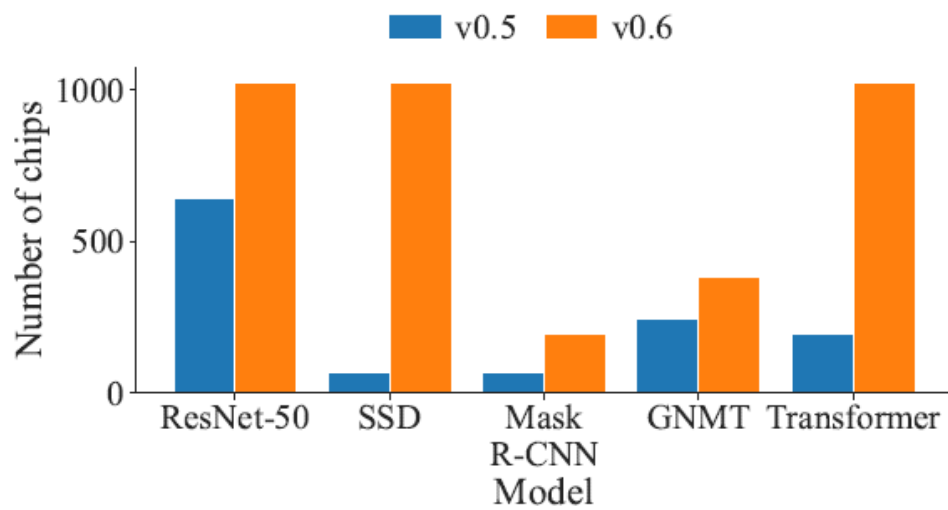


Figure 3.7: Number of chips necessary to produce the fastest time to solution for MLPerf versions v0.5 to v0.6. This number increased by as much as 5.5x

Chapter 4

MLPERF INFERENCE BENCHMARK

Machine learning (ML) powers a variety of applications from computer vision[64, 55, 95, 89] and natural language processing[135, 42] to self-driving cars[144, 7] and autonomous robots[93]. Although ML model training has been a development bottleneck and a considerable expense[4], inference has become a critical workload. Models can serve as many as 200 trillion queries and perform over 6 billion translations per day[92]. To address these growing computational demands, hardware, software, and system developers have focused on inference performance for a variety of use cases by designing optimized machine learning hardware and software. Estimates indicate that over 100 companies are targeting specialized inference chips[84]. By comparison, only about 20 companies are targeting training chips[129].

Each machine learning system takes a unique approach to inference, trading off latency, throughput, power, and model quality. The result is many combinations of machine learning tasks, models, data sets, frameworks, tool sets, libraries, architectures, and inference engines, making the task of evaluating inference performance nearly intractable. The spectrum of tasks is broad, including, but not limited to, image classification and localization, object detection and segmentation, machine translation, automatic speech recognition, text to speech, and recommendations. Even for a specific task, such as image classification, many machine learning models are viable. These models serve in a variety of scenarios from taking a single picture on a smartphone to continuously and concurrently detecting pedestrians through multiple cameras in an autonomous vehicle. Consequently, machine learning tasks have vastly different quality requirements and real-time processing demands. Even the implementations of the model's functions and operations can be highly framework specific, and they

increase the complexity of the design task. To quantify these tradeoffs, the machine learning field needs a benchmark that is architecturally neutral, representative, and reproducible.

Both academic and industrial organizations have developed ML inference benchmarks. Examples of prior art include AIMatrix[3], EEMBC MLMark[38], and AIXPRT[131] from industry, as well as AI Benchmark[74], TBD[151], Fathom[2], and DAWNbench[37] from academia. Each one has made substantial contributions to ML benchmarking, but they were designed without industry-wide input from ML system designers. As a result, there is no consensus on representative machine learning models, metrics, tasks, and rules.

We present MLPerf Inference[118], a standard ML inference benchmark suite with proper metrics and a benchmarking method (that complements MLPerf Training[97]) to fairly measure the inference performance of machine learning hardware, software, and services. Industry and academia jointly developed the benchmark suite and its methodology using input from researchers and developers at more than 30 organizations.

Over 200 ML engineers and practitioners contributed to the effort. MLPerf Inference is a consensus on the best benchmarking techniques, forged by experts in architecture, systems, and machine learning. We explain why designing the right ML benchmarking metrics, creating realistic ML inference scenarios, and standardizing the evaluation methods enables realistic performance optimization for inference quality.

We selected representative workloads for reproducibility and accessibility.

The ML ecosystem is rife with models. Comparing and contrasting ML system performance across these models is non-trivial because they vary dramatically in model complexity and execution characteristics. In addition, a name such as ResNet-50, fails to uniquely or portably describe a model. Consequently, quantifying system performance improvements with an unstable baseline is difficult.

A major contribution of MLPerf is the selection of representative models that permit reproducible measurements. Based on industry consensus, MLPerf Inference

comprises models that are mature and have earned community support. Because the industry has studied them and can build efficient systems, benchmarking is accessible and provides a snapshot of ML system technology. MLPerf models are also open source, making them a potential research tool.

We identified scenarios for realistic evaluation.

Machine learning inference systems range from deeply embedded devices to smartphones to data centers. They have a variety of real world applications and many figures of merit, each requiring multiple performance metrics. The right metrics, reflecting production use cases, allow not just MLPerf, but also publications, to show how a practical ML system would perform.

MLPerf Inference consists of four evaluation scenarios: single-stream, multi-stream, server, and offline. We arrived at them by surveying MLPerf’s broad membership, which includes both customers and vendors. These scenarios represent many critical inference applications. We show that performance can vary drastically under these scenarios and their corresponding metrics. MLPerf Inference provides a way to simulate the realistic behavior of the inference system under test; such a feature is unique among AI benchmarks.

We prescribe target qualities and tail-latency bounds in accordance with real world use cases.

Quality and performance are intimately connected for all forms of machine learning. System architectures occasionally sacrifice model quality to reduce latency, reduce total cost of ownership (TCO), or increase throughput. The tradeoffs among accuracy, latency, and TCO are application-specific. Trading 1% model accuracy for 50% lower TCO is prudent when identifying cat photos, but it is risky when detecting pedestrians for autonomous vehicles.

To reflect this aspect of real deployments, MLPerf defines model quality targets. We established per-model and scenario targets for inference latency and model quality. The latency bounds and target qualities are based on input gathered from ML system end users and ML practitioners. As MLPerf improves these parameters in accordance

with industry needs, the broader research community can track them to stay relevant.

We set permissive rules that allow users to show both hardware and software capabilities.

The community has embraced a variety of languages and libraries, so MLPerf Inference is a semantic-level benchmark. We specify the task and the rules, but we leave implementation to the submitters.

Our benchmarks allow submitters to optimize the reference models, run them through their preferred software tool chain, and execute them on their hardware of choice. Thus MLPerf Inference has two divisions: closed and open. Strict rules govern the closed division, which addresses the lack of a standard inference benchmarking workflow. The open division, on the other hand, allows submitters to change the model and demonstrate different performance and quality targets.

We present an inference benchmarking method that allows the models to change frequently while preserving the aforementioned contributions.

Machine learning evolves quickly, so the challenge for any benchmark is not performing the tasks, but implementing a method that can withstand rapid change.

MLPerf Inference focuses heavily on the benchmark’s modular design to make adding new models and tasks less costly while preserving the usage scenarios, target qualities, and infrastructure. As we show in Section 4.5.5, our design has allowed users to add new models easily. We plan to extend the scope to include more areas and tasks. We are working to add new models (e.g., recommendation and time series), new scenarios (e.g., “burst” mode), better tools (e.g., a mobile application), and better metrics (e.g., timing pre-processing) to better reflect the performance of the entire ML pipeline.

Impact

MLPerf Inference (version 0.5) was put to the test in October 2019. We received over 600 submissions from 14 organizations, spanning various tasks, frameworks, and platforms. Audit tests automatically evaluated the submissions and cleared 595 of them as valid. The results show a four-orders-of-magnitude performance variation, ranging

from embedded devices and smartphones to data center systems, demonstrating how the different metrics and inference scenarios are useful in more robustly assessing AI inference accelerators.

4.1 Inference Benchmarking Challenges

A useful ML benchmark must overcome three critical challenges: the diversity of models, the variety of deployment scenarios, and the array of inference systems.

4.1.1 Diversity of Models

Choosing the right models for a benchmark is difficult; the choice depends on the application. For example, pedestrian detection in autonomous vehicles has a much higher accuracy requirement than does labeling animals in photographs, owing to the different consequences of incorrect predictions. Similarly, quality-of-service (QoS) requirements for machine learning inference vary by several orders of magnitude from effectively no latency constraint for offline processes to milliseconds for real-time applications.

Covering the vast design space necessitates careful selection of models that represent realistic scenarios. Even for a single ML task, such as image classification, numerous models present different tradeoffs between accuracy and computational complexity, as Figure 4.1 shows. These models vary tremendously in compute and memory requirements (e.g., a 50x difference in FLOPS), while the corresponding Top-1 accuracy ranges from 55% to 83%[\[17\]](#). This variation creates a Pareto frontier rather than one of optimal choice. Even a small accuracy change (e.g., a few percent) can drastically alter the computational requirements (e.g., by 5-10x). For example, SE-ResNeXt-50[\[72, 143\]](#) and Xception[\[32\]](#) achieve roughly the same accuracy (79%), but exhibit a 2x computational difference.

4.1.2 Deployment Scenario Diversity

In addition to accuracy and computational complexity, a representative ML benchmark must take into account the input data’s availability and arrival pattern

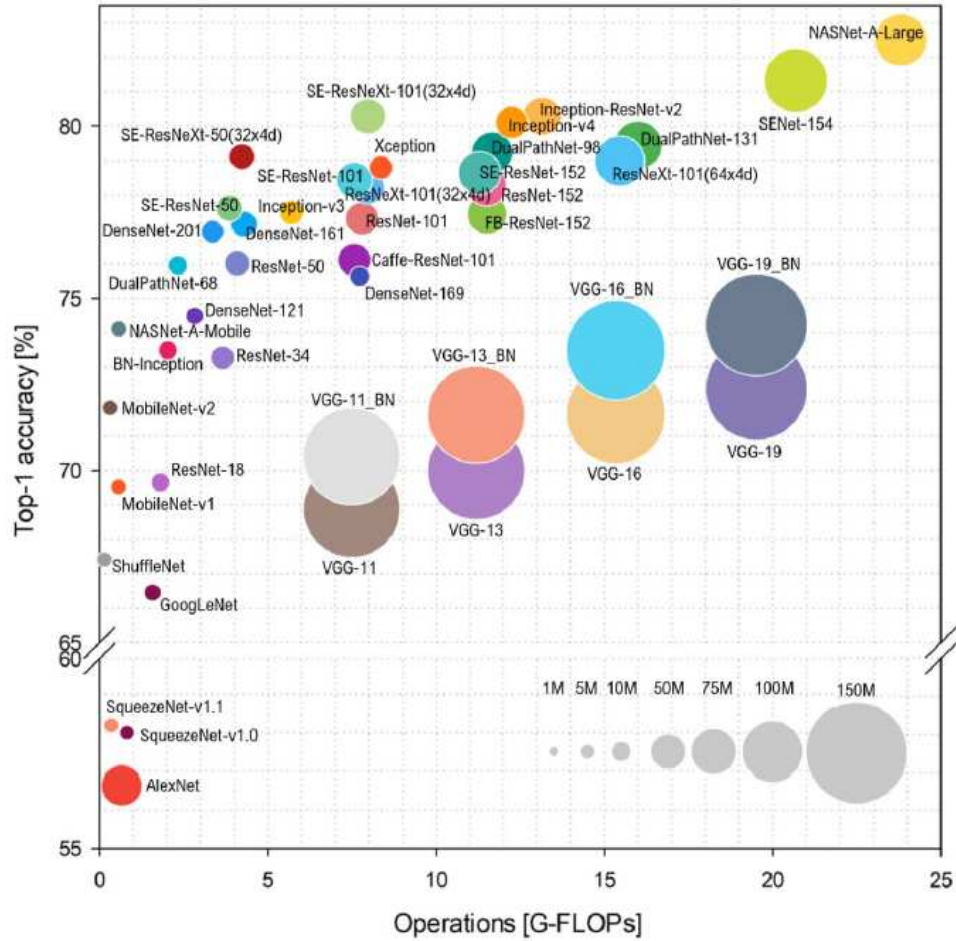


Figure 4.1: An example of ML model diversity for image classification (plot from [4]). No single model is optimal; each one presents a design tradeoff between accuracy, memory requirements, and computational complexity.

across a variety of application-deployment scenarios. For example, in offline batch processing, such as photo categorization, all the data may be readily available in (network) storage, allowing accelerators to reach and maintain peak performance. By contrast, translation, image tagging, and other tasks experience variable arrival patterns based on end-user traffic.

Similarly, real-time applications, such as augmented reality and autonomous vehicles handle a constant flow of data, rather than having it all in memory. Although the same model architecture could serve in each scenario, data batching and similar

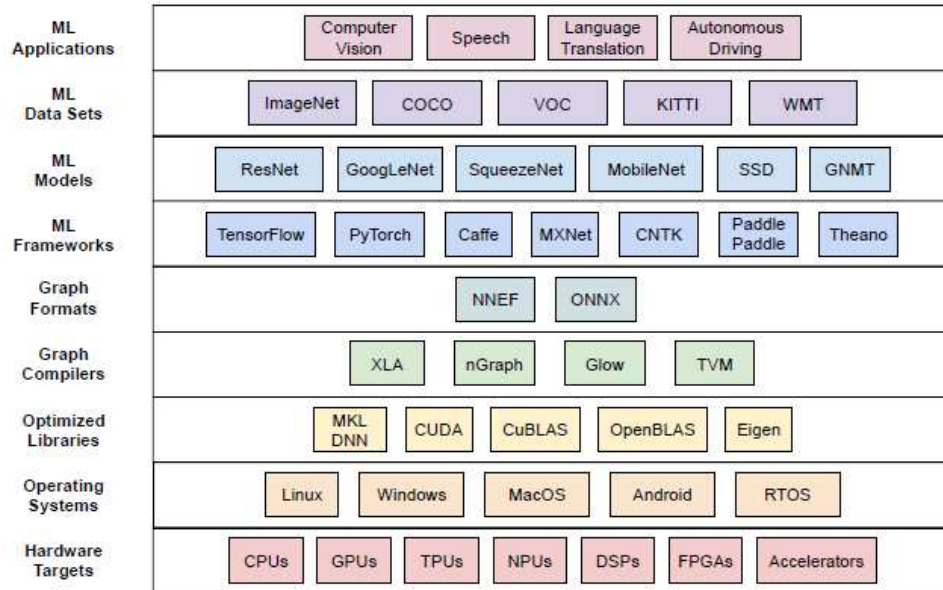


Figure 4.2: The diversity of options at every level of the stack, along with combinations across the layers, make benchmarking inference systems difficult.

optimizations may be applicable, leading to drastically different performance. Timing this on-device inference latency alone fails to reflect the real-world requirements.

4.1.3 Inference System Diversity

The possible combinations of inference applications, data sets, models, machine learning frameworks, tool sets, libraries, systems, and platforms are numerous, further complicating systematic and reproducible benchmarking. Figure 4.2 shows the wide breadth and depth of the ML space. The hardware and software side both exhibit substantial complexity.

On the software side, about a dozen ML frameworks commonly serve for developing deep learning models, such as Caffe/Caffe2[79], Chainer[132], CNTK[120], Keras[49], MXNet[27], TensorFlow[1], and PyTorch[112]. Independently, there are also many optimized libraries, such as cuDNN[31], Intel MKL[76], and FBGEMM[86], supporting various inference run times, such as Apple CoreML[5], Intel OpenVINO[75], NVIDIA TensorRT[109], ONNX Runtime[8], Qualcomm SNPE[115], and TF-Lite[91].

Each combination has idiosyncrasies that make supporting the most current neural network model architectures a challenge. Consider the non-maximum-suppression (NMS) operator for object detection. When training object detection models in TensorFlow, the regular NMS operator smooths out imprecise bounding boxes for a single object. However, this implementation is unavailable in TensorFlow Lite, which is tailored for mobile and instead implements fast NMS. As a result, when converting the model from TensorFlow to TensorFlow Lite, the accuracy of SSD-MobileNet-v1 decreases from 23.1% to 22.3% mAP. Such subtle differences make it hard to port models exactly from one framework to another.

On the hardware side, platforms are tremendously diverse, ranging from familiar processors (e.g., CPUs, GPUs, and DSPs) to FPGAs, ASICs, and exotic accelerators, such as analog and mixed-signal processors. Each platform comes with hardware-specific features and constraints that enable or disrupt performance depending on the model and scenario.

4.2 Benchmark Design

Combining model diversity with the range of software systems presents a unique challenge to deriving a robust ML benchmark that meets industry needs. To overcome that challenge, we adopted a set of principles for developing a robust yet flexible offering based on community input. In this section, we describe the benchmarks, quality targets, and scenarios under which the ML systems can be evaluated.

4.2.1 Representative, Broadly Accessible Workloads

Designing ML benchmarks is different from designing non-ML benchmarks. MLPerf defines high-level tasks (e.g., image classification) that a machine learning system can perform. For each, we provide one or more canonical reference models in a few widely used frameworks. Any implementation that is mathematically equivalent to the reference model is considered valid, and certain other deviations (e.g., numerical formats) are also allowed. For example, a fully connected layer can be implemented

AREA	TASK	REFERENCE MODEL	DATA SET	QUALITY TARGET
VISION	IMAGE CLASSIFICATION (HEAVY)	RESNET-50 v1.5 25.6M PARAMETERS 8.2 GOPS / INPUT	IMAGENET (224X224)	99% OF FP32 (76.456%) TOP-1 ACCURACY
VISION	IMAGE CLASSIFICATION (LIGHT)	MOBILENET-V1 224 4.2M PARAMETERS 1.138 GOPS / INPUT	IMAGENET (224X224)	98% OF FP32 (71.676%) TOP-1 ACCURACY
VISION	OBJECT DETECTION (HEAVY)	SSD-RESNET-34 36.3M PARAMETERS 433 GOPS / INPUT	COCO (1,200X1,200)	99% OF FP32 (0.20 MAP)
VISION	OBJECT DETECTION (LIGHT)	SSD-MOBILENET-V1 6.91M PARAMETERS 2.47 GOPS / INPUT	COCO (300X300)	99% OF FP32 (0.22 MAP)
LANGUAGE	MACHINE TRANSLATION	GNMT 210M PARAMETERS	WMT16 EN-DE	99% OF FP32 (23.9 SACREBLEU)

Figure 4.3: ML tasks in MLPerf Inference v0.5. Each one reflects critical commercial and research use cases for a large class of submitters, and together they cover a broad set of computing motifs (e.g., CNNs and RNNs).

with different cache-blocking and evaluation strategies. Consequently, submitted results require optimizations to achieve good performance.

A reference model and a valid class of equivalent implementations gives most ML systems freedom while still enabling relevant comparisons. MLPerf provides reference models using 32-bit floating point weights and, for convenience, carefully implemented equivalent models to address three formats: TensorFlow[1], PyTorch[112], and ONNX[8].

As Figure 4.3 illustrates, we chose an initial set of vision and language tasks, along with associated reference models. Together, vision and translation serve widely across computing systems, from edge devices to cloud data centers. Mature and well-behaved reference models with different architectures (e.g., CNNs and RNNs were available, too.

Image classification

Many commercial applications employ image classification, which is a de facto standard for evaluating ML system performance. A classifier network takes an image and selects the class that best describes it. Example applications include photo searches, text extraction, and industrial automation, such as object sorting and defect detection. We use the ImageNet 2012 data set[41], crop the images to 224x224 in pre-processing, and measure Top-1 accuracy.

We selected two models: a computationally heavyweight model that is more

accurate and a computationally lightweight model that is faster, but less accurate. The heavyweight model, ResNet-50 v1.5[64, 103], comes directly from the MLPerf Training suite to maintain alignment. ResNet-50 is the most common network used for performance claims. Unfortunately, it has multiple subtly different implementations that make most comparisons difficult. We specifically selected ResNet-50 v1.5 to ensure useful comparisons and compatibility across major frameworks. This network exhibits good reproducibility, making it a low-risk choice.

The lightweight model, MobileNet-v1-224[71], employs smaller, depthwise-separable convolutions to reduce the complexity and computational burden. MobileNet networks offer varying compute and accuracy options - we selected the full-width, full-resolution MobileNet-v1-1.0-224. It reduces the parameters by 6.1x and the operations by 6.8x compared with ResNet-50 v1.5. We evaluated both MobileNet-v1 and MobileNet-v2[119] for the MLPerf Inference v0.5 suite, selecting the former because of its wider adoption.

Object detection

Object detection is a vision task that determines the coordinates of bounding boxes around objects in an image and then classifies those boxes. Implementations typically use a pre-trained image classifier network as a backbone or feature extractor, then perform regression for localization and bounding box selection. Object detection is crucial for automotive tasks, such as detecting hazards and analyzing traffic, and for mobile-retail tasks, such as identifying items in a picture. We chose the COCO data set[94] with both a lightweight model and a heavyweight model.

Similar to image classification, we selected two models. Our small model uses the 300x300 image size, which is typical of resolutions in smartphones and other compact devices. For the larger model, we upscale the data set to more closely represent the output of a high-definition image sensor (1.44 MP total). The choice of the larger input size is based on community feedback, especially from automotive and industrial automation customers. The quality metric for object detection is mean average precision (mAP).

The heavyweight object detector’s reference model is SSD[95] with a ResNet-34 backbone, which also comes from our training benchmark. The lightweight object detector’s reference model uses a MobileNet-v1-1.0 backbone, which is more typical for constrained computing environments. We selected the MobileNet feature detector on the basis of feedback from the mobile and embedded communities.

Translation

Neural machine translation (NMT) is popular in natural language processing. NMT models translate a sequence of words from a source language to a target language and appear in translation applications and services. Our translation data set is WMT16 EN-DE[140]. The quality measurement is the Bilingual Evaluation Understudy (BLEU) score[111], implemented using SacreBLEU[114]. Our reference model is GNMT[142], which employs a well-established recurrent neural network (RNN) architecture and is part of the training benchmark. RNNs are popular for sequential and time-series data, so including GNMT ensures our reference suite captures a variety of compute motifs.

4.2.2 Robust Quality Targets

Quality and performance are intimately connected for all forms of machine learning. Although the starting point for inference is a pre-trained reference model that achieves a target quality, many system architectures can sacrifice model quality to reduce latency, reduce total cost of ownership (TCO), or increase throughput. The tradeoffs between accuracy, latency, and TCO are application specific. Trading 1% model accuracy for 50% lower TCO is prudent when identifying cat photos, but is less so during online pedestrian detection. To reflect this important aspect, we established per-model quality targets.

We require that almost all implementations achieve a quality target within 1% of the FP32 reference model’s accuracy. For example, the ResNet-50 v1.5 model achieves 76.46% Top-1 accuracy, and an equivalent model must achieve at least 75.70% Top-1

accuracy. Initial experiments, however, showed that for mobile-focused networks - MobileNet and SSD-MobileNet - the accuracy loss was unacceptable without re-training. We were unable to proceed with the low accuracy as performance benchmarking would become unrepresentative.

To address the accuracy drop, we took three steps. First, we trained the MobileNet models for quantization-friendly weights, enabling us to narrow the quality window to 2%. Second, to reduce the training sensitivity of MobileNet-based submissions, we provided equivalent MobileNet and SSD-MobileNet implementations quantized to an 8-bit integer format. Third, for SSD-MobileNet, we reduced the quality requirement to 22.0% mAP to account for the challenges of using a MobileNet backbone.

To improve the submission comparability, we disallow re-training. Our prior experience and feasibility confirmed that for 8-bit integer arithmetic, which was an expected deployment path for many systems, the 1% relative-accuracy target was easily achievable without re-training.

4.2.3 Realistic End-User Scenarios

ML applications have a variety of usage models and many figures of merit, which in turn require multiple performance metrics. For example, the figure of merit for an image recognition system that classifies a video camera’s output will be entirely different than for a cloud-based translation system.

To address these scenarios, we surveyed MLPerf’s broad membership, which includes both customers and vendors. On the basis of that feedback, we identified four scenarios that represent many critical inference applications: single-stream, multi-stream, server, and offline. These scenarios emulate the ML workload behavior of mobile devices, autonomous vehicles, robotics, and cloud-based systems (Figure 4.4).

Single-stream

The single-stream scenario represents one inference-query stream with a query sample size of 1, reflecting the many client applications where responsiveness is critical. An example is offline voice transcription on Google’s Pixel 4 smartphone. To measure

SCENARIO	QUERY GENERATION	METRIC	SAMPLES/QUERY	EXAMPLES
SINGLE-STREAM (SS)	SEQUENTIAL	90TH-PERCENTILE LATENCY	1	TYPING AUTOCOMPLETE, REAL-TIME AR
MULTISTREAM (MS)	ARRIVAL INTERVAL WITH DROPPING	NUMBER OF STREAMS SUBJECT TO LATENCY BOUND	N	MULTICAMERA DRIVER ASSISTANCE, LARGE-SCALE AUTOMATION
SERVER (S)	POISSON DISTRIBUTION	QUERIES PER SECOND SUBJECT TO LATENCY BOUND	1	TRANSLATION WEBSITE
OFFLINE (O)	BATCH	THROUGHPUT	AT LEAST 24,576	PHOTO CATEGORIZATION

Figure 4.4: Scenario description and metrics. Each scenario targets a real-world use case based on customer and vendor input.

performance, we inject a single query into the inference system; when the query is complete, we record the completion time and inject the next query. The metric is the query stream’s 90th-percentile latency.

Multi-stream

The multi-stream scenario represents applications with a stream of queries, but each query comprises multiple inferences, reflecting a variety of industrial automation and remote sensing tasks. For example, many autonomous vehicles analyze frames from multiple cameras simultaneously. To model a concurrent scenario, we send a new query comprising N input samples at a fixed time interval (e.g., 50 ms). The interval is benchmark-specific and also acts as a latency bound that ranges from 50 to 100 milliseconds. If the system is available, it processes the incoming query. If it is still processing the prior query in an interval, it skips that interval and delays the remaining queries by one interval. No more than 1% of the queries may produce one or more skipped intervals. A query’s N input samples are contiguous in memory, which accurately reflects production input pipelines and avoids penalizing systems that would otherwise require copying of samples to a contiguous memory region before starting inference. The performance metric is the integer number of streams that the system supports while meeting the QoS requirement.

Server

The server scenario represents online applicatoins where query arrival is random and latency is important. Almost every consumer-facing website is a good example, including services such as online translation from Baidu, Google, and Microsoft. For this scenario, queries have one sample each, in accordance with a Poisson distribution.

The system under test responds to each query within a benchmark-specific latency bound that varies from 15 to 250 milliseconds. No more than 1% of queries may exceed the latency bound for the vision tasks and no more than 3% may do so for translation. The server scenario’s performance metric is the Poisson parameter that indicates the queries-per-second (QPS) achievable while meeting the QoS requirement.

Offline

The offline scenario represents batch-processing applications where all data is immediately available and latency is unconstrained. An example is identifying the people and locations within a photo album. For this scenario, we send a single query that includes all sample-data IDs to be processed, and the system is free to process the input data in any order. Similar to the multi-stream scenario, neighboring samples in the query are contiguous in memory. The metric for the offline scenario is throughput measured in samples per second.

For the multi-stream and server scenarios, latency is a critical component of the system behavior and constrains various performance optimizations. For example, most inference systems require a minimum (and architecture-specific) batch size to fully utilize the underlying computational resources. However, the query arrival rate in servers is random, so they must optimize for tail latency and potentially process inferences with a sub-optimal batch size.

Figure 4.5 shows the latency constraints for each task in MLPerf Inference v0.5. As with other aspects of the benchmark, we selected these constraints on the basis of feasibility and community consultation. The multi-stream scenario’s arrival times for most vision tasks correspond to a framerate of 15-20 Hz, which is a minimum for many applications. The server scenario’s QoS constraints derive from estimates of the inference timing budget, given an overall user latency target.

4.2.4 Statistically Confident Tail-Latency Bounds

Each task and scenario combination requires a minimum number of queries to ensure results are statistically robust and adequately capture steady-state system

TASK	MULTISTREAM ARRIVAL TIME	SERVER QOS CONSTRAINT
IMAGE CLASSIFICATION (HEAVY)	50 MS	15 MS
IMAGE CLASSIFICATION (LIGHT)	50 MS	10 MS
OBJECT DETECTION (HEAVY)	66 MS	100 MS
OBJECT DETECTION (LIGHT)	50 MS	10 MS
MACHINE TRANSLATION	100 MS	250 MS

Figure 4.5: Latency constraints in the multi-stream and server scenarios.

behavior. That number is determined by the tail-latency percentile, the desired margin, and the desired confidence interval. Confidence is the probability that a latency bound is within a particular margin of the reported result. We chose a 99% confidence bound and set the margin to a value much less than the difference between the tail-latency percentage and 100%. Conceptually, that margin ought to be relatively small. Thus, our selection is one-twentieth of the difference between the tail-latency percentage and 100%. The equation is as follows:

$$Margin = \frac{1 - TailLatency}{20} \quad (1)$$

$$NumQueries = \left(NormsInv\left(\frac{1 - Confidence}{2}\right) \right)^2 \times \frac{TailLatency \times (1 - TailLatency)}{Margin^2} \quad (2)$$

Figure 4.6:

Equation 2 provides the number of queries that are necessary to achieve a statistically valid measurement. The math for determining the appropriate sample size for a latency-bound throughput experiment is the same as that for determining the appropriate sample size for an electoral poll given an infinite electorate where three variables determine the sample size: tail-latency percentage, confidence, and margin[128].

TAIL-LATENCY PERCENTILE	CONFIDENCE INTERVAL	ERROR MARGIN	INFERENCE	ROUNDED INFERENCE
90%	99%	0.50%	23,886	$3 \times 2^{13} = 24,576$
95%	99%	0.25%	50,425	$7 \times 2^{13} = 57,344$
99%	99%	0.05%	262,742	$33 \times 2^{13} = 270,336$

Figure 4.7: Query requirements for statistical confidence. All results must meet the minimum LoadGen scenario requirements.

MODEL	NUMBER OF QUERIES / SAMPLES PER QUERY			
	SINGLE-STREAM	MULTISTREAM	SERVER	OFFLINE
IMAGE CLASSIFICATION (HEAVY)	1K / 1	270K / N	270K / 1	1 / 24K
IMAGE CLASSIFICATION (LIGHT)	1K / 1	270K / N	270K / 1	1 / 24K
OBJECT DETECTION (HEAVY)	1K / 1	270K / N	270K / 1	1 / 24K
OBJECT DETECTION (LIGHT)	1K / 1	270K / N	270K / 1	1 / 24K
MACHINE TRANSLATION	1K / 1	90K / N	90K / 1	1 / 24K

Figure 4.8: Number of queries and samples per query for each task.

Figure 4.7 shows the query requirements. The total query count and tail-latency percentile are scenario- and task-specific. The single-stream scenario require 1,024 queries, and the offline scenario requires 1 query with at least 24,576 samples. The former has the fewest queries to execute, as we wanted the run time to be short enough that embedded platforms and smartphones could complete the benchmarks quickly.

For scenarios with latency requirements, our goal is to ensure a 99% confidence interval that the constraints hold. As a result, the benchmarks with more stringent latency constraints require more queries in a highly non-linear fashion. The number of queries is based on the aforementioned statistics and is rounded up to the nearest multiple of 2^{13} .

A 99th-percentile guarantee requires 262,742 queries, which rounds up to 33×2^{12} , or 270K. For both multi-stream and server, this guarantee for vision tasks requires 270K queries, as Figure 4.8 shows. Because a multi-stream benchmark will process N samples per query, the total number of samples will be $N \times 270K$. Machine translation has a 97th-percentile latency guarantee and requires only 90K queries.

For repeatability, we run the multi-stream and server scenarios several times. However, the multi-stream scenario’s arrival rate and query count guarantee a 2.5- to 7.0-hour run time. To strike a balance between repeatability and run time, we require five runs for the server scenario, with the result being the minimum of these five. The other scenarios require one run. We expect to revisit this choice in future MLPerf versions.

All benchmarks must also run for at least 60 seconds and process additional queries and/or samples as the scenarios require. The minimum run time ensures we measure the equilibrium behavior of power-management systems and systems that support dynamic voltage and frequency scaling (DVFS), particularly for the single-stream scenario with few queries.

4.3 Inference Submission System

An MLPerf Inference submission system contains a system under test (SUT), the load generator (LoadGen), a data set, and an accuracy script. In this section, we describe these various components. Figure 4.9 shows an overview of an inference system. The data set, LoadGen, and accuracy script are fixed for all submissions and are provided by MLPerf. Submitters have wide discretion to implement an SUT according to their architecture’s requirements and their engineering judgment. By establishing a clear boundary between submitter-owned and MLPerf-owned components, the benchmark maintains comparability among submissions.

4.3.1 System Under Test

The submitter is responsible for the system under test. The goal of MLPerf Inference is to measure system performance across a wide variety of architectures. However, realism, comparability, architecture neutrality, and friendliness to small submission teams require careful tradeoffs. For instance, some deployments have teams of compiler, computer architecture, and machine learning experts aggressively co-optimizing the training and inference systems to achieve cost, accuracy, and latency targets across

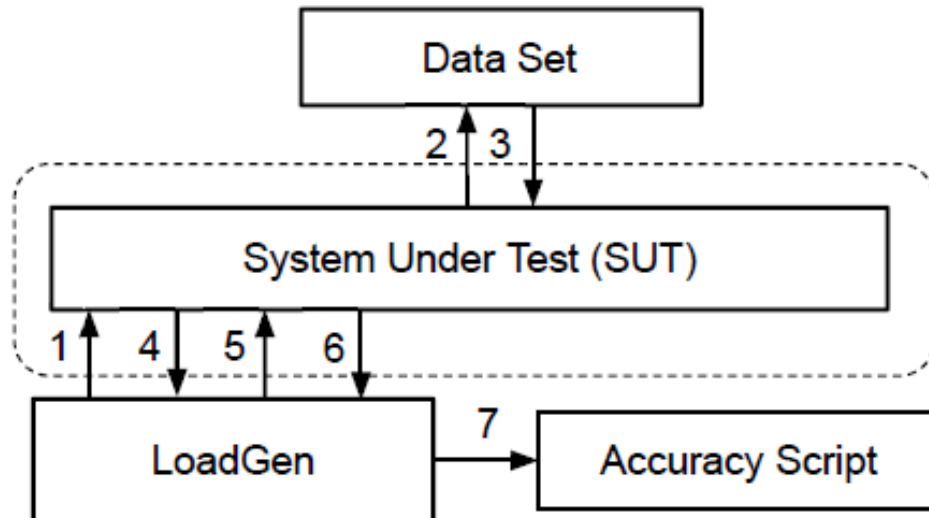


Figure 4.9: MLPerf Inference system under test (SUT) and associated components. First, the LoadGen requests that the SUT load samples (1). The SUT then loads samples into memory (2-3) and signals the LoadGen when it is ready (4). Next, the LoadGen issues requests to the SUT (5). The benchmark processes the results and returns them to the LoadGen (6), which then outputs logs for the accuracy script to read and verify (7).

a massive global customer base. An unconstrained benchmark would disadvantage companies with less experience and fewer ML training resources.

Therefore, we set the model equivalence rules to allow submitters to, for efficiency, re-implement models on different architectures. The rules provide a complete list of disallowed techniques and a list of allowed technique examples. We chose an explicit blacklist to encourage a wide range of techniques and to support architectural diversity. The list of examples illustrates the blacklist boundaries while also encouraging common and appropriate optimizations.

Examples of allowed techniques include arbitrary data arrangement, as well as different input and in-memory representations of weights, mathematically equivalent transformations, approximations (e.g., replacing a transcendental function with a polynomial), out-of-order query processing within the scenario’s limits, replacing dense

operations with mathematically equivalent sparse operations, fusing and unfusing operations, and dynamically switching between one or more batch sizes.

To promote architecture and application neutrality, we adopted untimed pre-processing. Implementations may transform their inputs into system-specific ideal forms as an untimed operation. Ideally, a whole-system benchmark should capture all performance-relevant operations. In MLPerf, however, we explicitly allow untimed pre-processing. For example, systems with integrated cameras can use hardware/software co-design to ensure that images reach memory in an ideal format; systems accepting JPEGs from the Internet cannot.

We also allow and enable quantization to many different numerical formats to ensure architecture neutrality. Submitters register their numerics ahead of time to help guide accuracy target discussions. The approved list includes INT4, INT8, INT16, UINT8, UINT16, FP11 (1-bit sign, 5-bit mantissa, and 5-bit exponent), FP16, bfloat16, and FP32. Quantization to lower-precision formats typically requires calibration to ensure sufficient inference quality. For each reference model, MLPerf provides a small, fixed data set that can be used to calibrate a quantized network. Additionally, it offers MobileNet versions that are pre-quantized to INT8, since without re-training (which we disallow), the accuracy falls dramatically.

Prohibited techniques

We prohibit re-training and pruning to ensure comparability. Although this restriction may fail to reflect realistic deployment in some cases, the interlocking requirements to use reference weights (possibly with calibration) and minimum accuracy targets are most important for comparability. We may eventually relax this restriction.

To simplify the benchmark evaluation, we disallow caching. In practice, inference systems cache queries. For example, “I love you” is one of Google Translate’s most frequent queries, but the service does not translate the phrase ab initio each time. Realistically modeling caching in a benchmark, however, is difficult because cache-hit rates vary substantially with the application. Furthermore, our data sets are relatively small, and large systems could easily cache them in their entirety.

We also prohibit optimizations that are benchmark- or dataset-aware and that are inapplicable to production environments. For example, real query traffic is unpredictable, but for the benchmark, the traffic pattern is pre-determined by the pseudorandom-number-generator seed. Optimizations that take advantage of a fixed number of queries or that take the LoadGen implementation into account are prohibited. Similarly, any optimization employing the statistics of the performance or accuracy data sets is forbidden.

4.3.2 Load Generator

The LoadGen is a traffic generator for MLPerf Inference that loads the SUT and measures performance. Its behavior is controlled by a configuration file it reads at the start of the run. The LoadGen produces the query traffic according to the rules of the previously described scenarios (i.e., single-stream, multi-stream, server, and offline). Additionally, it collects information for logging, debugging, and post-processing the data. It records queries and responses from the SUT, and at the end of the run, it reports statistics, summarizes the results, and determines whether the run was valid. Figure 4.10 shows how the LoadGen creates query traffic for each scenario. In the server scenario, for instance, it issues queries in accordance with a Poisson distribution to mimic a server’s query-arrival rates. In the single-stream case, it issues a query to the SUT and waits for the completion of that query before issuing another.

At startup, the LoadGen requests that the SUT load data set samples into memory. The SUT may load them into DRAM as an untimed operation and perform the other timed operations as the rules stipulate. These untimed operations include, but are not limited to, compilation, cache warmup, and pre-processing. The SUT signals the LoadGen when it is ready to receive the first query; a query is a request for inference on one or more samples. The LoadGen sends queries to the SUT in accordance with the selected scenario. Depending on that scenario, it can submit queries one at a time, at regular intervals, or in a Poisson distribution. The SUT runs inference on each query and sends the response back to the LoadGen, which either logs the response or

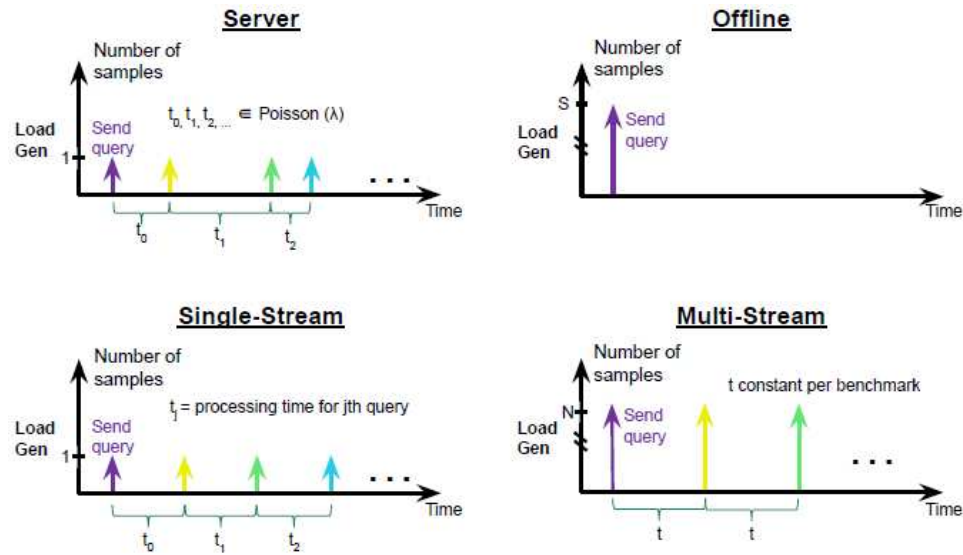


Figure 4.10: Timing and number of queries from the LoadGen.

discards it. After the run, an accuracy script checks the logged responses to determine whether the model accuracy is within tolerance.

The LoadGen has two primary operating modes: accuracy and performance. Both are necessary to validate MLPerf submissions. In accuracy mode, the LoadGen goes through the entire data set for the ML task. The model’s task is to run inference on the complete data set. Afterward, accuracy results appear in the log files, ensuring the model met the required quality target. In performance mode, the LoadGen avoids going through the entire data set, as the system’s performance can be determined by subjecting it to enough data set samples.

We designed the LoadGen to flexibly handle changes to the benchmark suite. MLPerf Inference has an interface between the SUT and LoadGen so it can handle new scenarios and experiments in the LoadGen and roll them out to all models and SUTs without extra effort. Doing so also facilitates compliance and auditing, since many technical rules about query arrivals, timing, and accuracy are implemented outside of submitter code. We achieved this feat by decoupling the LoadGen from the benchmarks and the internal representations (e.g., the model, scenarios, and quality and latency

metrics). The LoadGen implementation is a standalone C++ module.

The decoupling allows the LoadGen to support various language bindings, permitting benchmark implementations in any language. The LoadGen supports Python, C, and C++ bindings; additional bindings can be added. Another benefit of decoupling the LoadGen from the benchmark is that the LoadGen is extensible to support more scenarios, such as a multi-tenancy mode where the SUT must continuously serve multiple models while maintaining QoS constraints.

Moreover, placing the performance measurement code outside of submitter code fits with MLPerf’s goal of end-to-end system benchmarking. The LoadGen therefore measures the holistic performance of the entire SUT, rather than any individual part. Finally, this condition enhances the benchmark’s realism: inference engines typically serve as black box components of larger systems.

4.3.3 Data Set

We employ standard and publicly available data sets to ensure the community can participate. We do not host them directly, however. Instead, MLPerf downloads the data set before LoadGen uses it to run the benchmark. Figure 4.3 lists the data sets that we selected for each of the benchmarks.

4.3.4 Accuracy Checker

The LoadGen also has features that ensure the submission system complies with the rules. In addition, it can self-check to determine whether its source code has been modified during the submission process. To facilitate validation, the submitter provides an experimental config file that allows use of non-default LoadGen features. Details are in Section 4.4.2.

4.4 Submission System Evaluation

In this section, we describe the submission, review, and reporting process. Participants can submit results to different divisions and categories. All submissions are peer-reviewed for validity. Finally, we describe how the results are reported.

4.4.1 Result Submissions, Divisions, and Categories

A result submission contains information about the SUT: performance scores, benchmark code, a system description file that highlights the SUT’s main configuration characteristics (e.g., accelerator count, CPU count, software release, and memory system), and LoadGen log files detailing the performance and accuracy runs for a set of task and scenario combinations. All this data is uploaded to a public GitHub repository for peer review and validation before release.

MLPerf Inference is a suite of tasks and scenarios that ensures broad coverage, but a submission can contain a subset of them. Many traditional benchmarks, such as SPEC CPU, require submissions for all components. This approach is logical for a general purpose processor that runs arbitrary code, but ML systems are often highly specialized.

Divisions

MLPerf Inference has two divisions for submitting results: closed and open. Participants can send results to either or both, but they must use the same data set.

The closed division enables comparison of different systems. Submitters employ the same models, data sets, and quality targets to ensure comparability across wildly different architectures. This division requires pre-processing, post-processing, and a model that is equivalent to the reference implementation. It also permits calibration for quantization (using the calibration data set we provide) and prohibits re-training.

The open division fosters innovation in ML systems, algorithms, optimization, and hardware/software co-design. Participants must still perform the same ML task, but they may change the model architecture and the quality targets. This division allows arbitrary pre- and post-processing and arbitrary models, including techniques such as re-training. In general, submissions are directly comparable neither with each other, nor with closed submissions. Each open submission must include documentation about how it deviates from the closed division.

Categories

Following MLPerf Training, participants classify their submissions into one of three categories on the basis of hardware and software availability: available; preview; and research, development, or other systems (RDO). This categorization helps consumers identify the systems' maturity and whether they are readily available (for rent or purchase).

4.4.2 Result Review

A challenge of benchmarking inference systems is that many include proprietary and closed-source components, such as inference engines and quantization flows, that make peer review difficult. To accommodate these systems while ensuring reproducible results that are free from common errors, we developed a validation suite to assist with peer review. These validation tools perform experiments that help determine whether a submission complies with the defined rules.

Accuracy verification

The purpose of this test is to ensure valid inferences in performance mode. By default, the results that the inference system returns to the LoadGen are not logged and thus are not checked for accuracy. This choice reduces or eliminates processing overhead to allow accurate measurement of the inference system's performance. In this test, results returned from the SUT to the LoadGen are logged randomly. The log is checked against the log generated in accuracy mode to ensure consistency.

On-the-fly caching detection

The LoadGen produces queries by randomly selecting query samples with replacement from the data set, and inference systems may receive queries with duplicate samples. This duplication is likely for high performance systems that process many samples relative to the data set size. To represent realistic deployments, the rules prohibit caching of queries and intermediate data. The test has two parts: the first generates queries with unique sample indices, and the second generates queries with duplicate sample indices. It measures performance in each case. The way to detect

caching is to determine whether the test with duplicate sample indices runs significantly faster than the test with unique sample indices.

Alternate-random-seed testing

Ordinarily, the LoadGen produces queries on the basis of a fixed random seed. Optimizations based on that seed are prohibited. The alternate-random-seed test replaces the official random seed with alternates and measures the resulting performance.

Custom data sets

In addition to the LoadGen’s validation features, we use custom data sets to detect result caching. MLPerf Inference validates this behavior by replacing the reference data set with a custom data set. We measure the quality and performance of the system operating on the latter and compare the results with operation on the former.

4.4.3 Result Reporting

MLPerf Inference provides no “summary score.” Benchmarking efforts often elicit a strong desire to distill the capabilities of a complex system to a single number and thereby enable comparison of different systems. However, not all ML tasks are equally important for all systems, and the job of weighting some more heavily than others is highly subjective. At best, weighting and summarization are driven by the submitter catering to customer needs, as some systems may be designed for specific ML tasks. For instance, a system may be highly optimized for vision rather than translation. In such cases, averaging the results across all tasks makes no sense, as the submitter may not be targeting those markets.

4.5 Benchmark Assessment

On October 11, we put the inference benchmark to the test. We received from 14 organizations more than 600 submissions in all three categories (available, preview, and RDO) across the closed and open divisions. The results are the most extensive corpus of inference performance data available to the public, covering a range of ML tasks and scenarios, hardware architectures, and software runtimes. Each has gone

through extensive review before receiving approval as a valid MLPerf result. After review, we cleared 595 results as valid. In this section, we assess the closed division results on the basis of our inference benchmark objectives.

- Pick representative workloads for reproducibility, and allow everyone to access them (Section 4.5.1).
- Identify usage scenarios for realistic evaluation (Section 4.5.2).
- Set permissive rules that allow submitters to showcase both hardware and software capabilities (Sections 4.5.3 and 4.5.4).
- Describe a method that allows the benchmarks to change (Section 4.5.5).

4.5.1 Task Coverage

Because we allow submitters to pick any task to evaluate their system’s performance, the distribution of results across tasks can reveal whether those tasks are of interest to ML system vendors. We analyzed the submissions to determine the overall coverage. Figure 4.11 shows the breakdown for the tasks and models in the closed division. Although the most popular model was - unsurprisingly - ResNet-50 v1.5, it was just under three times as popular as GNMT, the least popular model. This small spread and otherwise uniform distribution suggests we selected a representative set of tasks.

In addition to selecting representative tasks, another goal is to provide vendors with varying quality and performance targets. The ideal ML model may differ with the use case (as Figure 4.1 shows, a vast range of models can target a given task). Our results reveal that vendors equally supported different models for the same task because each model has unique quality and performance tradeoffs. In the case of object detection, we saw about the same number of submissions for both SSD-MobileNet-v1 and SSD-ResNet-34.

4.5.2 Scenario Usage

We aim to evaluate systems in realistic use cases - a major motivator for the LoadGen and scenarios. To this end, Figure 4.12 shows the distribution of results

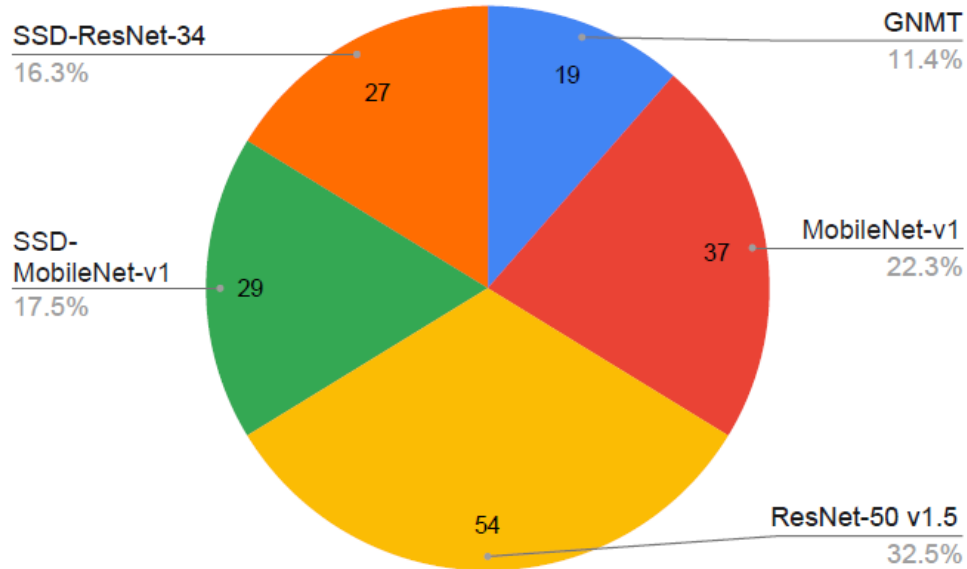


Figure 4.11: Results from the closed division. The distribution indicates we selected representative workloads for the benchmark’s initial release.

among the various task and scenario combinations. Across all tasks, the single-stream and offline scenarios are the most widely used and are also the easiest to optimize and run. Server and multi-stream were more complicated and had longer run times because of the QoS requirements and more numerous queries. GNMT garnered no multi-stream submissions, possibly because the constant arrival interval is unrealistic in machine translation. Therefore, it was the only model and scenario combination with no submissions.

The realistic MLPerf Inference scenarios are novel and illustrate many important and complex performance considerations that architects face, but that studies often overlook. Figure 4.13 demonstrates that all systems deliver less throughput for the server scenario than for the offline scenario owing to the latency constraint and attendant sub-optimal batching. Optimizing for latency is challenging and underappreciated.

Not all systems handle latency constraints equally well, however. For example, system B loses about 50% or more of its throughput for all three models, while system

	SINGLE-STREAM	MULTISTREAM	SERVER	OFFLINE
GNMT	2	0	6	11
MOBILENET-V1	18	3	5	11
RESNET-50 v1.5	19	5	10	20
SSD-MOBILENET-V1	8	3	5	13
SSD-RESNET-34	4	4	7	12
TOTAL	51	15	33	67

Figure 4.12: High coverage of models and scenarios.

A loses as much as 40% for NMT, but approximately 10% for the vision models. The throughput degradation differences may be a result of a hardware architecture optimized for low batch size or more effective dynamic batching in the inference engine and software stack - or, more likely, a combination of the two. Even with this limited data, one clear implication is that a performance comparison with unconstrained latency has little bearing on a latency-constrained scenario. Therefore, performance analysis should ideally include both.

Additionally, the performance impact of latency constraints varies with network type. Across all five systems with NMT results, the throughput reduction for the server scenario is 39-55%. In contrast, the throughput reduction for ResNet-50 v1.5 varies from 3% to 35% with an average of about 20%, and the average for MobileNet-v1 is under 10%. The vast throughput reduction differences likely reflect some combination of NMT’s variable text input, more significant software stack optimization, and NMT’s more complex network architecture. A second lesson from this data is that the impact of latency constraints on different models extrapolates poorly. Even among classification models, the average performance loss for ResNet-50 v1.5 is approximately double that of MobileNet-v1.

4.5.3 Processor Types and Software Frameworks

A variety of platforms can employ ML solutions, from fully general purpose CPUs to programmable GPUs and DSPs, FPGAs, and fixed-function accelerators. Our

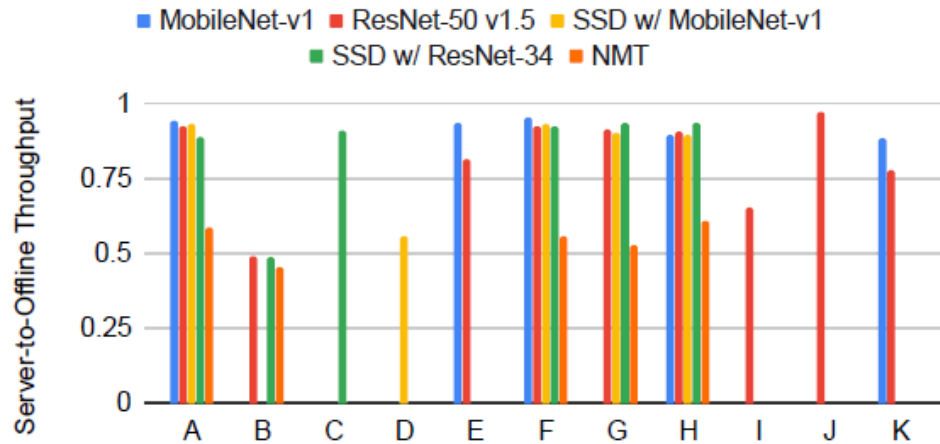


Figure 4.13: Throughput degradation from server scenario (which has a latency constraint) for 11 arbitrarily chosen systems from the closed division. The server scenario performance for each model is normalized to the performance of that same model in the offline scenario. A score of 1 corresponds to a model delivering the same throughput for the offline and server scenarios. Some systems, including C, D, I, and J, lack results for certain models because participants need not submit results for all models.

results reflect this diversity. Figure 4.14 shows that the MLPerf Inference submissions covered most hardware categories, indicating our v0.5 benchmark suite and method can evaluate any processor architecture.

Many ML software frameworks accompany the various processor types. Figure 4.15 shows the frameworks for benchmarking the hardware platforms. ML software plays a vital role in unleashing the hardware’s performance. Some runtimes are designed to work with certain hardware types to fully harness their capabilities; employing the hardware without the corresponding framework may still succeed, but the performance may fall short of the hardware’s potential. The figure shows that CPUs have the most framework diversity and that TensorFlow has the most architectural variety.

4.5.4 System Diversity

The submissions cover a broad power and performance range, from mobile and edge devices to cloud computing. The performance delta between the smallest and

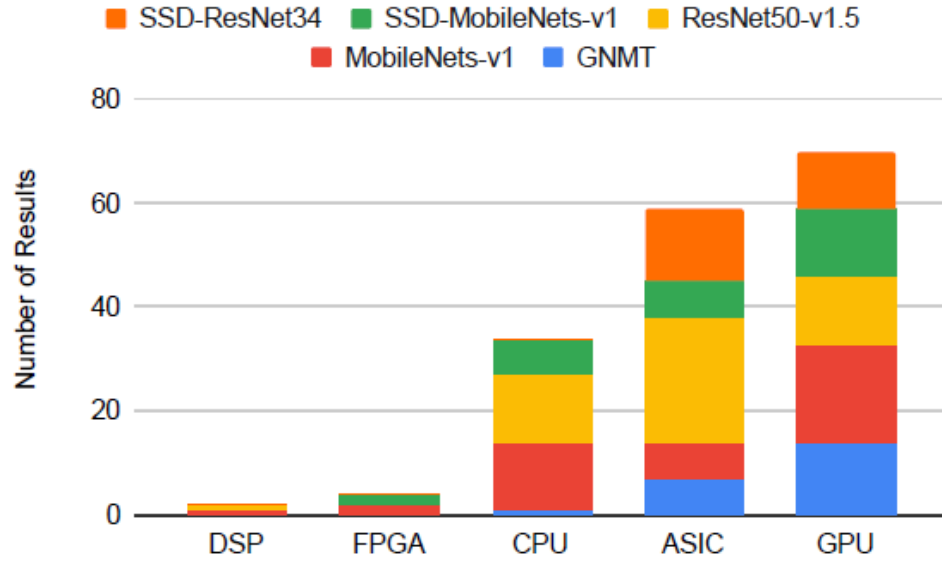


Figure 4.14: Results from the closed division. They cover almost every kind of processor architecture - CPUs, GPUs, DSPs, FPGAs, and ASICs.

largest inference systems is four orders of magnitude, or about 10,000x.

Figure 4.16 shows the results across all tasks and scenarios except for GNMT (MS), which had no submissions. In cases such as the MobileNet-v1 single-stream scenario (SS), ResNet-50 v1.5 (SS), and SSD-MobileNet-v1 offline (O), systems exhibit a large performance difference (100x). Because these models have many applications, the systems that target them cover everything from low-power embedded devices to high performance servers. GNMT server (S) exhibits much less performance variation among systems.

The broad performance range implies that the tasks we initially selected for MLPerf Inference v0.5 are general enough to represent many use cases and market segments. The wide array of systems also indicates that our method (the LoadGen, metrics, etc.) is widely applicable.

4.5.5 Open Division

We received 429 results in the less restrictive open division. A few highlights include 4-bit quantization to boost performance, exploration of various models (instead

	ASIC	CPU	DSP	FPGA	GPU
ARM NN		X			X
FURIOSAAI				X	
HAILO SDK	X				
HANGUANG AI	X				
ONNX		X			
OPENVINO		X			
PYTORCH		X			
SNPE			X		
SYNAPSE	X				
TENSORFLOW	X	X			X
TENSORFLOW LITE		X			
TENSORRT					X

Figure 4.15: Framework versus hardware architecture.

of the reference model) to perform the task, and high throughput under latency bounds tighter than what the closed division rules stipulate. We also saw submissions that pushed the limits of mobile chipset performance. Typically, vendors use one accelerator at a time. We are seeing instances of multiple accelerators working concurrently to deliver high throughput in a multi-stream scenario - a rarity in conventional mobile situations. Together these results show the open division is encouraging the industry to push system limits.

4.6 Lessons Learned

Over the course of a year, we have learned several lessons and identified opportunities for improvement, which we present here.

4.6.1 Models: Breadth vs. Use-Case Depth

Balancing the breadth of applications (e.g., image recognition, object detection, and translation) and models (e.g., CNNs and LSTMs) and the depth of the use cases (Figure 4.4) is important to industry. We therefore implemented 4 versions of each benchmark, 20 in total. Limited resources and the need for speedy innovation prevented

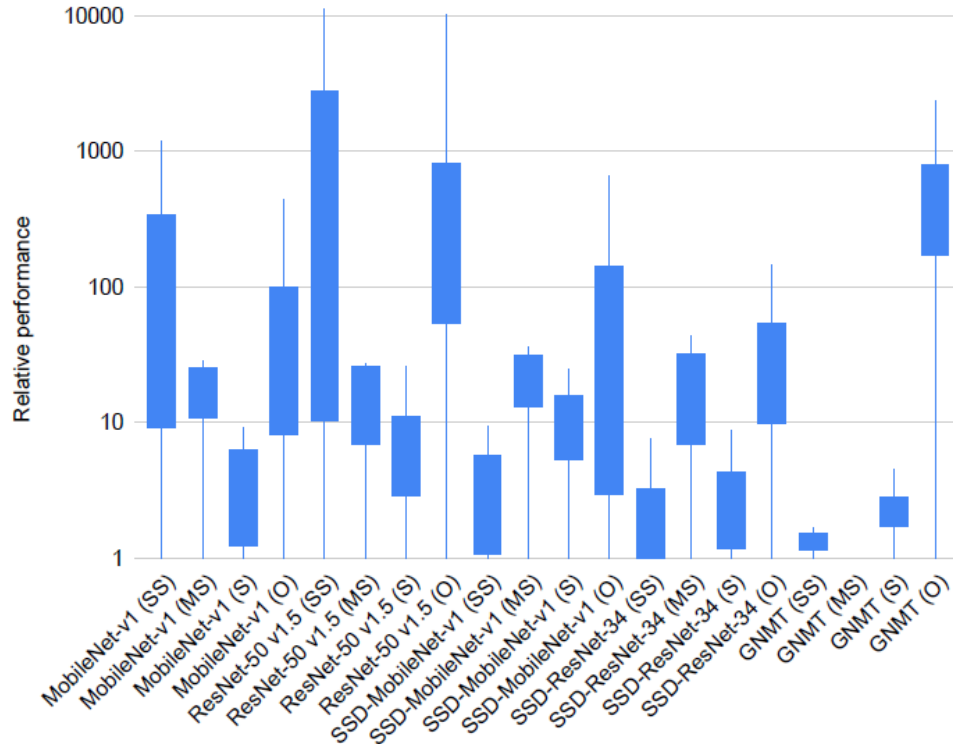


Figure 4.16: Performance for different models in the single-stream (SS), multi-stream (MS), server (S), and offline (O) scenarios. Scores are relative to the performance of the slowest system for the particular scenario.

us from including more applications (e.g., speech recognition and recommendation) and models (e.g., Transformers[135], BERT[42], and DLRM[107, 61]), but we aim to add them soon.

4.6.2 Metrics: Latency vs. Throughput

Latency and throughput are intimately related, and considering them together is crucial: we use latency-bounded throughput (Figure 4.5). A system can deliver excellent throughput yet perform poorly when latency constraints arise. For instance, the difference between the offline and server scenarios is that the latter imposes a latency constraint and implements a non-uniform arrival rate. The result is lower throughput because large input batches become more difficult to form. For some systems, the server scenario’s latency constraint reduces performance by as little as 3%

(relative to offline); for others, the loss is much greater (50%).

4.6.3 Data Sets: Public vs. Private

The industry needs larger and better quality public data sets for ML system benchmarking. After surveying various industry and academic scenarios, we found for the SSD-large use case a wide spectrum of input image sizes, ranging roughly from 200x200 to 8 MP. We settled on two resolutions as use case proxies: small images, where 0.09 MP (300x300) represents mobile and some data center applications, and large images, where 1.44 MP (1,200x1,200) represents robotics (including autonomous vehicles) and high-end video analytics. In practice, however, SSD-large employs an upscaled (1,200x1,200) COCO data set (Figure 4.3), as dictated by the lack of good public detection data sets with large images. Some such data sets exist, but they are less than ideal. ApolloScape[136], for example, contains large images, but it lacks bounding box annotations and its segmentation annotations omit labels for some object pixels (e.g., when an object has pixels in two or more non-contiguous regions, owing to occlusion). Generating the bounding box annotations is therefore difficult in these cases. The Berkeley DeepDrive[147] images are lower in resolution (720p). We need more data set generators to address these issues.

4.6.4 Performance: Modeled vs. Measured

Although it is common practice, characterizing a network’s computational difficulty on the basis of parameter size or operator count (Figure 4.1) can be an oversimplification. For example, 10 systems in the offline and server scenarios computed the performance of both SSD-ResNet-34 and SSD-MobileNet-v1. The former requires 175x more operations per image, but the actual throughput is only 50-60x less. This consistent 3x difference between the operation count and the observed performance shows how network structure can affect performance.

4.6.5 Process: Audits and Auditability

Because submitters can re-implement the reference benchmarks to maximize their system’s capabilities, the result review process (Section 4.4.2) was crucial to ensuring validity and reproducibility. We found about 40 issues in the approximately 180 results from the closed division. We ultimately released only 166 of these results. Issues ranged from failing to meet the quality targets (Figure 4.3), latency bounds (Figure 4.5), and query requirements (Figure 4.8) to inaccurately interpreting the rules. Thanks to the LoadGen’s accuracy checkers (Section 4.3.4 and submission checker scripts, we identified many issues automatically. The checkers also reduced the burden so only about three engineers had to comb through the submissions. In summary, since the diversity of options at every level of the ML inference stack is complicated (Figure 4.2), we found auditing and auditability to be necessary for ensuring result integrity and reproducibility.

Chapter 5

A SCHEDULER-DRIVEN ADAPTIVE FRAMEWORK FOR EXTREME SCALE SOFTWARE STACKS

Dynamic runtime schedulers have experienced broad success in efficiently executing ill-balanced, dynamic parallel programs on multi- and many-core systems. In particular, task graph-based runtimes[13, 20, 26, 47, 62, 81, 82, 90, 98, 117, 133, 152] have proven to offer new levels of load balancing and scaling capabilities.

In extreme scale systems, increased disparity between wire and transistor switching speeds aggravates the need for such schedulers, as new hardware solutions to cope with limited power envelope increase hardware heterogeneity, in their functionality (dark silicon) and efficiency (NTV, DVFS).

Scratchpad-based systems also emerge (as seen in GPUs, Runnemedede[23], Traleika Glacier[24], Cell[28], and SOCs) to cope with power constraints and the lack of cache scalability. Programming scratchpads requires the explicit specification of data movement in and out of the scratchpad memory.

Good utilization of scratchpads is achieved when data transfers are coarse. Hence, to some degree, extreme scale programs will be characterized by the execution of coarse data transfers followed by computations sized according to the amount of transferred data, and so on. Needless to say, the efficiency of static optimizations is jeopardized in such variable runtime environments, as their benefit is a direct (possibly negative) function of the runtime environment.

We explore the explicit discrimination of tasks according to a *task type*[126] that reflects the utilization of the underlying architecture. We expose task type knowledge to the runtime schedulers, and allow them to toggle the software stack between type-specific modes, enabling and disabling optimizations as a function of the current mode.

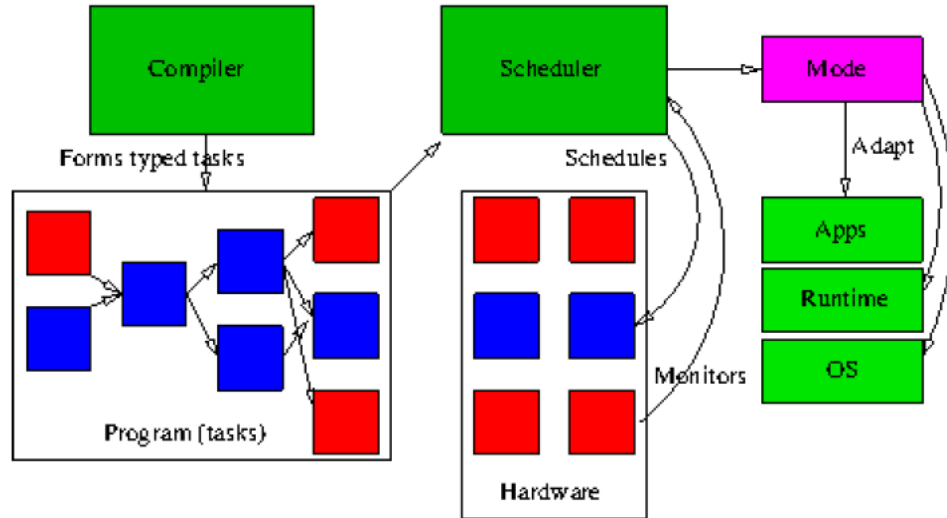


Figure 5.1: Flow of the scheduler-driven adaptive optimization framework

The general idea is to decompose the execution of programs into tasks of different types, which reflect a quality of the operations within a task. As depicted in Figure 5.1, the types should be generated as annotations to tasks by the compiler. The runtime scheduler determines a run-time characterization of the set of tasks currently scheduled, which we call “run-time mode.” Determining types at the task level enables low mode computation overheads.

An example of this is a type that would discriminate tasks as a function of their utilization of floating point operations. A mode that specifies that the currently scheduled tasks are low in floating point operations would allow a numerical library to use software floating point emulation in this mode, increasing the profitability of turning floating point units off. Turning the floating point units off can be performed by the runtime when the mode represents low utilization. By representing a mode with a vector (of modes) v , this system generalizes to a broad set of optimizations, through the definition of a domain of efficiency D for each optimization o :

$$v \in D(o) \Rightarrow \text{turn } o \text{ to state } x$$

A vector mode seems general enough to handle the runtime optimization of dark

silicon processors, in which only a subset of specialized hardware parts (as for instance Taylor’s “conservation cores” [130]) can be powered on simultaneously in order to stay within the processor’s power budget.

We apply this general principle to a simple and universally applicable set of load types, which discriminates computation from communication. We validate our model using an adaptive data compression engine.

5.1 Adaptive Data Compression Engine

The optimization considered here is a transformation of a tiled array, in which an array in dense representation may be turned into sparse representation before being processed. Data tiles are marked accordingly as either sparse or dense.

The cost of optimization is roughly linear in the number of elements E in the tile, while the benefit is in $(E - nnz)$ where nnz is the number of non-zero elements. Dynamically determining whether a tile is worth compressing by analyzing the data has limited profitability since it is also linear with respect to E . Ideally, we would like compression to occur (mostly) in phases where nnz is small enough and omitted in phases where nnz is larger.

In terms of execution time, the transformation optimization is worth applying when the computation time for a tile is low as compared to its communication time. Compression is also worth turning off when computations are significantly more important w.r.t. communications (i.e., when nnz is large enough). This defines the two transition rules for our adaptive compression engine, which turns compression on when the computation-to-communication ratio goes below a certain threshold α and turns compression off when the ratio goes above a different threshold β .

Interestingly, the efficiency domain of the compression optimization is defined in the production of the computation-to-communication ratio and of its own state (on or off).

5.2 Experimental Results

We have validated our approach using matrix-vector multiplication. To ensure the presence of both sparse and dense blocks in the input matrix, test cases were generated by modifying the Graph500[59] generator to produce the initial sparse matrix and overlaying it with dense tiles.

These initial tests were run on four processors using matrices of size 2048 x 2048 and blocks of size 128 x 128. In these cases, a block was considered to be a suitable candidate for compression if it contained fewer than 128 non-zero elements. We observed speedups ranging between 57% and 68% compared to standard matrix-vector multiplication and speedup of 15% compared to sparse matrix-vector multiplication.

Chapter 6

RELATED WORK

6.1 Related Work

Cantonnet and co-workers evaluated UPC using NPB[10] and other benchmarks in [48, 22]. Berlin and co-workers considered Pthreads, OpenMP, MPI, and UPC using numerical (CG) and non-numerical (hash table update and integer sort) workloads in [16]. Coarfa and co-workers compared MPI, UPC, and Coarray Fortran using the NPB in [34, 35]. These papers considered the NPB, MG, CG, SP, and BT. Because of the generation of the hardware employed in the experiments, multi-core effects were not considered. In [113], MPI and UPC were evaluated from both a productivity and performance standpoint using the power method on sparse matrices. [122] compares HPCS languages (X10, Chapel, and Fortress) using a simple version of the Hartree-Fock method found in quantum chemistry. In [39], the authors evaluate a wide range of programming models on their maturity (including tool support) and suitability to their applications. Performance experiments were not the focus of that paper. In [105], the authors present comparative experiments with Cilk, Go, TBB, and Chapel. Of these, only Chapel can run on clusters. However, in that study, even on a single shared memory node, Chapel did not show good performance or scalability. MPI, UPC, and Coarray Fortran were compared in the context of the Cray Gemini interconnect in [121] using NPB FT and microbenchmarks. Dun et al. evaluated Chapel using a range of tests, from microbenchmarks to a molecular dynamics mini-application[46]. In [85], the authors compare Serial, OpenMP, MPI, and CUDA against Chapel, Charm++, Liszt, and Loci when implementing LULESH. This interesting work focuses solely on the parallel pattern of the explicit stencil computation, which, while important, leaves

out numerous patterns and applications. Moreover, at the cluster level, only weak scaling results are presented, which do not fully stress scalability. In [12], the authors compare MPI against MPIOPENMP for an explicit stencil operation and use that as a motivation to create an MPI+X programming model, where X consists of asynchronously spawned tasks, based on overdecomposition of the computational domain. This approach shows promise and has the advantage that it is only a small departure from “classical” MPI1 style programming.

Our contribution differs from the above works in that each PRK focuses on a single, parameterized application pattern, allowing us to separate performance artifacts in different parts of an application. In addition, our work is unique in terms of the combination of number of *scalable* runtimes considered, and the extensiveness of the multiple kernel tests. Previous studies using Synchron_p2p include [43, 15], which proposed new features for OpenSHMEM and MPI, respectively.

6.2 MLPerf Training Benchmark

Prior ML benchmarks vary in granularity and scope. Microbenchmarks such as DeepBench[9] measure kernel-level operations that appear in commonly deployed models. Benchmarking such low-level operations fails to address the challenges associated with numerical precision, hyperparameter choices, and system scale, which we described in the previous section. Furthermore, it neither captures the end-to-end application, nor accounts for memory- and cache-hierarchy effects across layers and operations, nor measures the data pre-processing that deep learning commonly employs.

Several benchmarks are defined at the granularity of entire DNN models. Fathom[2] and Google TF Benchmarks[56] provide a reference suite of DNN models that span a wide application space, but they specifically measure model throughput and fail to account for accuracy. Similarly, TBD (Training Benchmarks for DNNs)[151] profiles training on GPUs (but not other architectures) across diverse workloads, measuring characteristics such as memory and hardware utilization. Our benchmark builds

on the diversity of applications in these projects while also capturing the quality and performance tradeoffs.

DAWNBench[37] was the first multi-entrant benchmark competition to use “time to train” (originally called time to accuracy) to measure the end-to-end performance of deep learning systems; it allowed optimizations across model architectures, optimization procedures, software frameworks, and hardware platforms. Our benchmark follows a similar approach, but handles more diverse tasks3.2.1, and it uses important rules and mechanisms in the Closed division3.3.2.1 to enable fair comparisons of hardware and software systems.

6.3 MLPerf Inference Benchmark

MLPerf strives to incorporate and build on the best aspects of prior work while also including community input.

AI Benchmark

AI Benchmark[74] is arguably the first mobile inference benchmark suite. Its results and leaderboard focus on Android smartphones and only measure latency. The suite provides a summary score, but it fails to explicitly specify quality targets. We aim at a variety of devices (our submissions range from IoT devices to smartphones and edge/server-scale systems), as well as four scenarios per benchmark.

EEMBC MLMark

EEMBC MLMark[38] measures the performance and accuracy of embedded inference devices. It also includes image classification and object detection tasks, as MLPerf does, but it lacks use case scenarios. MLMark measures performance at explicit batch sizes, whereas MLPerf allows submitters to choose the best batch sizes for different scenarios. Additionally, the former imposes no target quality restrictions, whereas the latter does impose restrictions.

Fathom

Fathom[2] provides a suite of models that incorporate several layer types (e.g., convolution, fully connected, and RNN). Still, it focuses on throughput rather than

accuracy. Like Fathom, we include a suite of models that comprise various layers. Compared with Fathom, MLPerf provides both PyTorch and TensorFlow reference implementations for optimization, and it introduces a variety of inference scenarios with different performance metrics.

AI Matrix

AI Matrix[3] is Alibaba’s AI accelerator benchmark. It uses microbenchmarks to cover basic operators such as matrix multiplication and convolution, measures performance for fully connected and other common layers, includes full models that closely track internal applications, and offers a synthetic benchmark to match the characteristics of real workloads. MLPerf has a smaller model collection and focuses on simulating scenarios using the LoadGen.

DeepBench

Microbenchmarks such as DeepBench[9] measure the library implementation of kernel-level operations (e.g., 5,124x700x2,048 GEMM) that are important for performance in production models. They are useful for efficient development, but fail to address the complexity of full models.

TBD (Training Benchmarks for DNNs)

TBD[128] is a joint project from the University of Toronto and Microsoft Research that focuses on ML training. It provides a wide spectrum of ML models in three frameworks (TensorFlow, MXNet, and CNTK), along with a powerful tool chain for their improvement. It focuses on evaluating GPU performance and only has one full model (Deep Speech 2) that covers inference.

DAWNBench

DAWNBench[37] was the first multi-entrant benchmark competition to measure the end-to-end performance of deep learning systems. It allowed optimizations across model architectures, optimization procedures, software frameworks, and hardware platforms. DAWNBench inspired MLPerf, but we offer more tasks, models, and scenarios.

Chapter 7

CONCLUSION

7.1 Parallel Research Kernels

We compared performance and scalability of eight programming models using three important patterns from the PRK. While there was no clear winner, a number of trends emerged. The incumbent models, MPI1 and SHMEM, performed well in general, although with Transpose, both suffered at scale compared to MPISHM, which sends fewer and larger messages. SHMEM was better than MPI1 for Transpose, but otherwise, they were roughly the same. The critical role of explicit aggregation of messages is clear from the Transpose results, with both variants of MPI+X winning by a large margin at scale, which is attributed to the message sizes. While MPIOPENMP is the easiest way for programmers to achieve aggregation and to take advantage of the low-latency synchronizations in shared memory, we see evidence that MPISHM is the superior way to realize it, both from the empirical results of Synchron_p2p and Transpose, and because of the challenges of mixing different runtimes (MPI and OpenMP). The success of SHMEM and MPI+X merits future investigation of SHMEM+X.

Both implementations of UPC delivered lower performance than the more explicit MPI1 and SHMEM, and the standard-compliant UPC implementation of Synchron_p2p suffered due to lack of support for this synchronization motif (partly solved by a language extension). It may be possible to improve performance by eliminating barriers, but that is non-trivial, due to the lack of explicit support for point-to-point synchronization in UPC (or other PGAS languages).

Charm++ performs best for Stencil at granularities for which it was designed, but fared poorly in Synchron_p2p and Transpose. We do not observe much benefit from

overdecomposition. The simple and naturally load balanced PRK are not well-suited to evaluate this feature. Grappa performed decently for the fine-grain Synchron_p2p and medium-grain Stencil workloads, but scalability needs to be improved by better and more flexible message aggregation. In our kernels we found FEB most effective for synchronization through control variables, not directly through primary solution values.

The scope of this work was limited in the kernels and models considered. All the kernels we used can be load balanced statically, which makes them amenable to easy implementation in MPI and derivatives. Asynchronous, adaptive models such as Charm++, Grappa, HPX (High Performance ParallelX)[81], and Legion[13] cannot demonstrate their strength with such simple workloads. We are developing a new PRK[54] that defies static load balancing, to explore this design space that is clearly less favorable for MPI and static PGAS models; the latter provide only low-level primitives for building dynamic load-balancing capability. None of the kernels we used tests fault tolerance, a feature that is today only provided by the Charm++ runtime, but that is rapidly gaining importance. We are working on an objective framework, using existing PRK, to measure effectiveness and overhead of runtime error recovery mechanisms. We are also expanding our PRK implementations to Legion and HPX, and will include performance and scalability measurements of Fine-Grain MPI[83] in our next evaluation.

7.2 MLPerf Training Benchmark

MLPerf Training is a suite of ML benchmarks that represent both industrial and academic use cases. In addition to being the only widely used ML-training benchmark suite boasting such coverage, it has made the following contributions:

- Precise definition of model architectures and training procedures for each benchmark. This feature enables system comparisons for equivalent workloads, whereas previous results often involved substantially different variants of a given model (for example, ResNet-50 has at least five variants).
- Reference implementations and rule definitions to address the challenges unique to benchmarking ML training. These challenges include the stochastic nature

of training processes, the necessity of training to completion to determine the quality impact of performance optimizations, and the need for workload variation at different system scales.

Although MLPerf focuses on relative system performance, as the online results demonstrate, it also offers general lessons about ML and benchmarking:

- Realistic data set size is critical to ensuring realistic memory system behavior - for example, the initial NCF data set was too small and could reside entirely in memory. Furthermore, when benchmarking data sets that are smaller than industrial scale, training time should exclude the startup time, which would be proportionally less in actual use.
- Small hyperparameter changes can produce considerable performance changes. However, based on our experience with hyperparameter borrowing, hyperparameters are relatively portable at similar system scales, even across architectures, numerics, or software stacks.
- Frameworks exhibit subtle optimizer-algorithm variations that affect convergence.

Machine learning is an evolving field, however, and we have much more to learn. To keep pace, MLPerf establishes a process to maintain and update the suite. For example, MLPerf v0.6 includes several updates: the ResNet-50 benchmark added LARS[146], GNMT’s model architecture improved to increase translation quality, and the MiniGo reference switched from Python to C++ to increase performance.

7.3 MLPerf Inference Benchmark

MLPerf Inference’s core contribution is a comprehensive framework for measuring ML inference performance across a spectrum of use cases. We briefly summarize the three main aspects of inference benchmarking here.

Performance metrics

To make fair, apples-to-apples comparisons of AI systems, consensus on performance metrics is critical. We crafted a collection of such metrics: latency, latency-bounded throughput, throughput, and maximum number of inferences per query - all subject to a pre-defined accuracy target and some likelihood of achieving that target.

Latency or inference execution time is often the metric that system and architecture designers employ. Instead, we identify latency-bounded throughput as a measure of inference performance in industrial use cases, representing data center inference processing. Although this metric is common for data center CPUs, we introduce it for data center ML accelerators. Prior work often uses throughput or latency; the formulation in this work reflects more realistic deployment constraints.

Accuracy/performance tradeoff

ML systems often trade off between accuracy and performance. Prior art varies widely concerning acceptable inference accuracy degradation. We consulted with domain experts from industry and academia to set both the accuracy and the tolerable degradation thresholds for MLPerf Inference, allowing distributed measurement and optimization of results to tune the accuracy/performance tradeoff. This approach standardizes AI system design and evaluation, and it enables academic and industrial studies, which can now use the accuracy requirements of MLPerf Inference workloads to compare their efforts to industrial implementations and the established accuracy standards.

Evaluation of AI inference accelerators

An important contribution of this work is identifying and describing the metrics and inference scenarios (server, single-stream, multi-stream, and offline) in which AI inference accelerators are useful. An accelerator may stand out in one category while underperforming in another. Such a degradation owes to optimizations such as batching (or the lack thereof), which is use-case dependent. MLPerf introduces batching for three out of four inference scenarios (server, multi-stream, and offline) across the five networks, and these scenarios can expose additional optimizations for AI system development and research.

7.4 ASAFESSS

The presentation adaptive framework is general in that it can drive many different software behaviors by modeling them as optimizations associated with an efficiency domain. However, it is limited to optimizations that can make use of the current context. This assumes that the execution of programs happens in *phases* in which properties of the environment don't change too quickly. A potential direction for future work would be automating the process by having a compiler introduce type annotations for the scheduler. R-Stream[99] is a natural candidate for this as it natively discriminates communications from computations and forms parallel task-graph programs for SWARM, OCR, and CnC.

BIBLIOGRAPHY

- [1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 265283, USA, 2016. USENIX Association.
- [2] Robert Adolf, Saketh Rama, Brandon Reagen, Gu-Yeon Wei, and David Brooks. Fathom: reference workloads for modern deep learning methods. IISWC '16, pages 1–10, 09 2016.
- [3] Alibaba. Ai matrix. <https://aimatrix.ai/en-us/>, 2018.
- [4] D. Amodei and D. Hernandez. Ai and compute. <https://blog.openai.com/ai-and-compute/>, 2018.
- [5] Apple. Core ml: Integrate machine learning models into your app. <https://developer.apple.com/documentation/coreml>, 2017.
- [6] Peter Auer, Mark Herbster, and Manfred K. K Warmuth. Exponentially many local minima for single neurons. In D. Touretzky, M. C. Mozer, and M. Hasselmo, editors, *Advances in Neural Information Processing Systems*, volume 8, pages 316–322. MIT Press, 1996.
- [7] V. Badrinarayanan, A. Kendall, and R. Cipolla. Segnet: A deep convolutional encoder-decoder architecture for image segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 39(12):2481–2495, 2017.
- [8] J. Bai, F. Lu, and et al. K. Zhang. Onnx: Open neural network exchange. <https://github.com/onnx/onnx>, 2019.
- [9] Baidu. Deepbench: Benchmarking deep learning operations on different hardware. <https://github.com/baidu-research/DeepBench>, 2017.
- [10] D.H. Bailey, E. Barszcz, J.T. Barton, D.S. Browning, R.L. Carter, L. Dagum, R.A. Fatoohi, P.O. Frederickson, T.A. Lasinski, R.S. Schreiber, H.D. Simon, V. Venkatakrisnan, and S.K. Weeratunga. The nas parallel benchmarks. *The International Journal of Supercomputing Applications*, 5(3):63–73, 1991.

- [11] Ron Banner, Itay Hubara, Elad Hoffer, and Daniel Soudry. Scalable methods for 8-bit training of neural networks. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, page 51515159, Red Hook, NY, USA, 2018. Curran Associates Inc.
- [12] Richard F. Barrett, Dylan T. Stark, Courtenay T. Vaughan, Ryan E. Grant, Stephen L. Olivier, and Kevin T. Pedretti. Toward an evolutionary task parallel integrated mpi + x programming model. In *Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores*, PMAM '15, page 3039, New York, NY, USA, 2015. Association for Computing Machinery.
- [13] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: Expressing locality and independence with logical regions. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, Washington, DC, USA, 2012. IEEE Computer Society Press.
- [14] Francois Belletti, Karthik Lakshmanan, Walid Krichene, Yi-Fan Chen, and John Anderson. Scalable realistic recommendation datasets through fractal expansions. *ArXiv*, abs/1901.08910, 2019.
- [15] Roberto Belli and Torsten Hoefler. Notified access: Extending remote memory access programming models for producer-consumer synchronization. IPDPS '15, pages 871–881, 05 2015.
- [16] Konstantin Berlin, Jun Huan, Mary Jacob, Garima Kochhar, Jan Prins, William Pugh, Ponnuswamy Sadayappan, Jaime Spacco, and Chau-Wen Tseng. Evaluating the impact of programming language features on the performance of parallel applications on cluster architectures. LCPC '03, pages 194–208, 2003.
- [17] S. Bianco, R. Cadene, L. Celona, and P. Napoletano. Benchmark analysis of representative deep neural network architectures. *IEEE Access*, 6:64270–64277, 2018.
- [18] Dan Bonachea, Rajesh Nishtala, Paul Hargrove, and Katherine Yelick. Efficient point-to-point synchronization in upc. 2006.
- [19] Shekhar Borkar and Andrew A. Chien. The future of microprocessors. *Commun. ACM*, 54(5):6777, May 2011.
- [20] Zoran Budimlić, Michael Burke, Vincent Cavé, Kathleen Knobe, Geoff Lowney, Ryan Newton, Jens Palsberg, David Peixotto, Vivek Sarkar, Frank Schlimbach, and Saĝnak Taşırlar. Concurrent collections. *Sci. Program.*, 18(34):203217, August 2010.

- [21] Mark Bull and Carwyn Ball. Point-to-point synchronisation on shared memory architectures. 2003.
- [22] Francois Cantonnet, Yiyi Yao, Mohamed Zahran, and Tarek El-Ghazawi. Productivity analysis of the upc language. IPDPS '04, 2004.
- [23] Nicholas P. Carter, Aditya Agrawal, Shekhar Borkar, Romain Cledat, Howard David, Dave Dunning, Joshua Fryman, Ivan Ganey, Roger A. Golliver, Rob Knauerhase, Richard Lethin, Benoit Meister, Asit K. Mishra, Wilfred R. Pinfold, Justin Teller, Josep Torrellas, Nicolas Vasilache, Ganesh Venkatesh, and Jianping Xu. Runnemed: An architecture for ubiquitous high-performance computing. In *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, HPCA '13, page 198209, USA, 2013. IEEE Computer Society.
- [24] Vincent Cave, Romain Cldat, Paul Griffin, Ankit More, Bala Seshasayee, Shekhar Borkar, Sanjay Chatterjee, Dave Dunning, and Joshua Fryman. Traleika glacier. *Parallel Comput.*, 64(C):3349, May 2017.
- [25] B. Chan. Openai five. <https://openai.com/blog/openai-five>, 2018.
- [26] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA '05*, page 519538, New York, NY, USA, 2005. Association for Computing Machinery.
- [27] T. Chen, Mu Li, Y. Li, M. Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, B. Xu, C. Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *ArXiv*, abs/1512.01274, 2015.
- [28] T. Chen, R. Raghavan, J. N. Dale, and E. Iwata. Cell broadband engine architecture and its first implementation: A performance view. *IBM J. Res. Dev.*, 51(5):559572, September 2007.
- [29] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Tvm: An automated end-to-end optimizing compiler for deep learning. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation, OSDI'18*, page 579594, USA, 2018. USENIX Association.
- [30] Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishikesh Aradhye, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Ispir, Rohan Anil, Zakaria Haque, Lichan Hong, Vihan Jain, Xiaobing Liu, and Hemal Shah.

- Wide & deep learning for recommender systems. In *Proceedings of the 1st Workshop on Deep Learning for Recommender Systems*, DLRS 2016, page 710, New York, NY, USA, 2016. Association for Computing Machinery.
- [31] Sharan Chetlur, C. Woolley, Philippe Vandermersch, J. Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *ArXiv*, abs/1410.0759, 2014.
- [32] F. Chollet. Xception: Deep learning with depthwise separable convolutions. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1800–1807, 2017.
- [33] A. Choromanska, Mikael Henaff, Michaël Mathieu, G. B. Arous, and Y. LeCun. The loss surfaces of multilayer networks. In *AISTATS*, 2015.
- [34] Cristian Coarfa, Yuri Dotsenko, Jason Eckhardt, and John Mellor-Crummey. Co-array fortran performance and potential: An npb experimental study. *LCPC '03*, pages 177–193, 10 2003.
- [35] Cristian Coarfa, Yuri Dotsenko, John Mellor-Crummey, François Cantonnet, Tarek El-Ghazawi, Ashrujit Mohanti, Yiyi Yao, and Daniel Chavarría-Miranda. An evaluation of global address space languages: Co-array fortran and unified parallel c. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '05, page 3647, New York, NY, USA, 2005. Association for Computing Machinery.
- [36] Cody Coleman, Daniel Kang, Deepak Narayanan, Luigi Nardi, Tian Zhao, Jian Zhang, Peter Bailis, Kunle Olukotun, Chris Ré, and Matei Zaharia. Analysis of dawnbench, a time-to-accuracy machine learning performance benchmark. *SIGOPS Oper. Syst. Rev.*, 53(1):1425, 7 2019.
- [37] Cody A. Coleman, D. Narayanan, Daniel Kang, T. Zhao, Jian Zhang, L. Nardi, Peter Bailis, K. Olukotun, C. Ré, and M. Zaharia. Dawnbench : An end-to-end deep learning benchmark and competition. In *NIPS ML Systems Workshop*, 2017.
- [38] Embedded Microprocessor Benchmark Consortium. Introducing the eembc ml-mark benchmark. <https://www.eembc.org/mlmark/index.php>, 2019.
- [39] R. Cook, E. Dube, I. Lee, C. Shereda, F. Wang, and L. Nau. Survey of novel programming models for parallelizing applications at exascale. Technical report, Lawrence Livermore National Laboratory, 2011.
- [40] T.P.P. Council. Transaction processing performance council. <http://www.tpc.org>, 2005.

- [41] J. Deng, W. Dong, R. Socher, L. Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, 2009.
- [42] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *ArXiv*, abs/1810.04805, 10 2018.
- [43] J. Dinan, G. Jost, K. Underwood, and R.W. Wisniewski. Reducing synchronization overhead through bundled communication. In *OpenSHMEM and Related Technologies*, 2014.
- [44] Kaivalya M. Dixit. The spec benchmarks. *Parallel Computing*, 17(10):1195 – 1209, 1991.
- [45] J. J. Dongarra. The linpack benchmark: An explanation. In *Proceedings of the 1st International Conference on Supercomputing*, page 456474, Berlin, Heidelberg, 1988. Springer-Verlag.
- [46] N. Dun and K. Taura. An empirical performance study of chapel programming language. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum*, pages 497–506, 2012.
- [47] Alejandro Duran, Eduard Ayguad, Rosa M. Badia, Jess Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. Ompss: a proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21:173–193, 06 2011.
- [48] Tarek El-Ghazawi and Francois Cantonnet. Upc performance and potential: A npb experimental study. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing, SC '02*, page 126, Washington, DC, USA, 2002. IEEE Computer Society Press.
- [49] F. Chollet *et al.* Keras. <https://keras.io>, 2015.
- [50] K. Feind. Shared memory access (shmem) routines. In *CUG 1995*, 1995.
- [51] MPI Forum. Mpi: A message-passing interface standard, 1994.
- [52] MPI Forum. Mpi-2: Extensions to the message-passing interface, 1996.
- [53] MPI Forum. Mpi: A message-passing interface standard. version 3.0., 2012.
- [54] Evangelos Georganas, Rob Wijngaart, and Tim Mattson. Design and implementation of a parallel research kernel for assessing dynamic load-balancing capabilities. *IPDPS '16*, pages 73–82, 05 2016.

- [55] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'14, page 26722680, Cambridge, MA, USA, 2014. MIT Press.
- [56] Google. Tensorflow benchmarks. <https://www.tensorflow.org/performance/benchmarks>, 2019.
- [57] M. Gori and A. Tesi. On the problem of local minima in backpropagation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 14(1):76–86, 1992.
- [58] Priya Goyal, Piotr Dollár, Ross B. Girshick, P. Noordhuis, L. Wesolowski, Aapo Kyrola, Andrew Tulloch, Y. Jia, and Kaiming He. Accurate, large minibatch sgd: Training imagenet in 1 hour. *ArXiv*, abs/1706.02677, 2017.
- [59] Graph500. The graph500 list. <http://www.graph500.org>.
- [60] GroupLens. Movielens 20m dataset. <https://grouplens.org/datasets/movielens/20m/>, 2016.
- [61] Udit Gupta, X. Wang, M. Naumov, Carole-Jean Wu, Brandon Reagen, D. Brooks, Bradford Cottel, K. Hazelwood, Bill Jia, H. S. Lee, A. Malevich, D. Mudigere, M. Smelyanskiy, Liang Xiong, and X. Zhang. The architectural implications of facebook’s dnn-based personalized recommendation. *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 488–501, 2020.
- [62] Habanero. Habanero-c. <http://hc.rice.edu>.
- [63] K. He, G. Gkioxari, P. Dollr, and R. Girshick. Mask r-cnn. In *2017 IEEE International Conference on Computer Vision (ICCV)*, pages 2980–2988, 2017.
- [64] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.
- [65] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. In *European Conference on Computer Vision*, volume 9908, pages 630–645, 10 2016.
- [66] Xiangnan He, Lizi Liao, Hanwang Zhang, Liqiang Nie, Xia Hu, and Tat-Seng Chua. Neural collaborative filtering. In *Proceedings of the 26th International Conference on World Wide Web, WWW '17*, page 173182, Republic and Canton of Geneva, CHE, 2017. International World Wide Web Conferences Steering Committee.

- [67] J.L. Hennessy and D.A. Patterson. Computer architecture: A quantitative approach. Elsevier, 2011.
- [68] M. Heroux, R. Brightwell, and M. Wolf. Bi-modal mpi and mpi + threads computing on scalable multicore systems. 2011.
- [69] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, and B. Kingsbury. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, 29(6):82–97, 2012.
- [70] Torsten Hoefler, James Dinan, Darius Buntinas, Pavan Balaji, Brian Barrett, Ron Brightwell, William Gropp, Vivek Kale, and Rajeev Thakur. Mpi + mpi: A new hybrid approach to parallel programming with mpi plus shared memory. *Computing*, 95, 2013.
- [71] A. Howard, Menglong Zhu, Bo Chen, D. Kalenichenko, W. Wang, Tobias Weyand, M. Andreetto, and H. Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *ArXiv*, abs/1704.04861, 2017.
- [72] J. Hu, L. Shen, and G. Sun. Squeeze-and-excitation networks. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 7132–7141, 2018.
- [73] Jonathan Huang, Vivek Rathod, Chen Sun, Menglong Zhu, Anoop Korattikara, Alireza Fathi, Ian Fischer, Zbigniew Wojna, Yang Song, Sergio Guadarrama, and Kevin Murphy. Speed/accuracy trade-offs for modern convolutional object detectors. pages 3296–3297, 07 2017.
- [74] Andrey Ignatov, R. Timofte, W. Chou, Ke Wang, Max Wu, Tim Hartley, and L. Gool. Ai benchmark: Running deep neural networks on android smartphones. In *ECCV Workshops*, 2018.
- [75] Intel. Intel distribution of openvino toolkit. <https://software.intel.com/en-us/openvino-toolkit>, 2018.
- [76] Intel. Math kernel library. <https://software.intel.com/en-us/mkl>, 2018.
- [77] Intel. Bigdl: Distributed deep learning library for apache spark. <https://github.com/intel-analytics/BigDL>, 2019.
- [78] X. Jia, S. Song, W. He, Yangzihao Wang, Haidong Rong, F. Zhou, L. Xie, Zhenyu Guo, Yuanzhou Yang, L. Yu, Tiegang Chen, G. Hu, Shaohuai Shi, and Xiaowen Chu. Highly scalable deep learning training system with mixed-precision: Training imagenet in four minutes. *ArXiv*, abs/1807.11205, 2018.

- [79] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM International Conference on Multimedia*, MM '14, page 675678, New York, NY, USA, 2014. Association for Computing Machinery.
- [80] N. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, Raminder Bajwa, Sarah Bates, S. Bhatia, Nan Boden, Al Borchers, R. Boyle, P. Cantin, C. Chao, Chris Clark, Jeremy Coriell, M. Daley, M. Dau, J. Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, Diemthu Le, C. Leary, Z. Liu, K. Lucke, Alan Lundin, Gordon MacKean, A. Maggiore, Maire Mahony, K. Miller, R. Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, A. Phelps, J. Ross, Matt Ross, A. Salek, Emad Samadiani, Chris Severn, G. Sizikov, Matthew Snelham, J. Souter, D. Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, H. Toma, Erick Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon. In-datacenter performance analysis of a tensor processing unit. *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–12, 2017.
- [81] Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey. Hpx: A task based programming model in a global address space. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, PGAS '14, New York, NY, USA, 2014. Association for Computing Machinery.
- [82] Laxmikant V. Kale and Sanjeev Krishnan. Charm++: A portable concurrent object oriented system based on c++. In *Proceedings of the Eighth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '93, page 91108, New York, NY, USA, 1993. Association for Computing Machinery.
- [83] Humaira Kamal and Alan Wagner. FG-MPI: Fine-grain MPI for multicore and clusters. In *11th IEEE Intl. Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC) held in conjunction with IPDPS-24*, pages 1–8, April 2010.
- [84] D. Kanter. Hpc meets machine learning. <https://www.realworldtech.com/sc19-hpc-meets-machine-learning>, 2019.
- [85] Ian Karlin, Abhinav Bhatele, Jeff Keasler, Bradford Chamberlain, Jonathan Cohen, Zachary DeVito, Riyaz Haque, Dan Laney, Edward Luke, Felix Wang, David

- Richards, Martin Schulz, and Charles Still. Exploring traditional and emerging parallel programming models using a proxy application. pages 919–932, 05 2013.
- [86] D. S. Khudia, P. Basu, and S. Deng. Open-sourcing fbgemm for state-of-the-art server-side inference. <https://engineering.fb.com/ml-applications/fbgemm/>, 2018.
- [87] Urs Koster, Tristan Webb, Xin Wang, Marcel Nassar, Arjun Bansal, William Constable, Oguz Elibol, Stewart Hall, Luke Hornof, Amir Khosrowshahi, Carey Kloss, Ruby Pai, and Naveen Rao. Flexpoint: An adaptive numerical format for efficient training of deep neural networks. 11 2017.
- [88] A. Krizhevsky. One weird trick for parallelizing convolutional neural networks. *ArXiv*, abs/1404.5997, 2014.
- [89] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS’12, page 10971105, Red Hook, NY, USA, 2012. Curran Associates Inc.
- [90] Christopher Lauderdale and Rishi Khan. Towards a codelet-based runtime for exascale computing: Position paper. In *Proceedings of the 2nd International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, EX-ADAPT ’12, page 2126, New York, NY, USA, 2012. Association for Computing Machinery.
- [91] J. Lee, N. Chirkov, Ekaterina Ignasheva, Yury Pisarchyk, Mogan Shieh, F. Riccardi, Raman Sarokin, Andrei Kulik, and Matthias Grundmann. On-device neural net inference with mobile gpus. *ArXiv*, abs/1907.01989, 2019.
- [92] K. Lee, V. Rao, and W. C. Arnold. Accelerating facebook’s infrastructure with application-specific hardware. <https://engineering.fb.com/data-center-engineering/accelerating-infrastructure/>, 2019.
- [93] Sergey Levine, Peter Pastor, Alex Krizhevsky, and Deirdre Quillen. Learning hand-eye coordination for robotic grasping with deep learning and large-scale data collection. *The International Journal of Robotics Research*, 37, 03 2016.
- [94] Tsung-Yi Lin, M. Maire, Serge J. Belongie, James Hays, P. Perona, D. Ramanan, Piotr Dollár, and C. L. Zitnick. Microsoft coco: Common objects in context. In *ECCV*, 2014.
- [95] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander Berg. Ssd: Single shot multibox detector. In *ECCV*, volume 9905, pages 21–37, 10 2016.

- [96] S. Markidis, S. W. D. Chien, E. Laure, I. B. Peng, and J. S. Vetter. Nvidia tensor core programmability, performance precision. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 522–531, 2018.
- [97] Peter Mattson, Christine Cheng, Cody Coleman, Greg Diamos, Paulius Micikevicius, David Patterson, Hanlin Tang, Gu-Yeon Wei, Peter Bailis, Victor Bittorf, David Brooks, Dehao Chen, Debojyoti Dutta, Udit Gupta, Kim Hazelwood, Andrew Hock, Xinyuan Huang, Atsushi Ike, Bill Jia, Daniel Kang, David Kanter, Naveen Kumar, Jeffery Liao, Guokai Ma, Deepak Narayanan, Tayo Oguntebi, Gennady Pekhimenko, Lillian Pentecost, Vijay Janapa Reddi, Taylor Robie, Tom St. John, Tsuguchika Tabaru, Carole-Jean Wu, Lingjie Xu, Masafumi Yamazaki, Cliff Young, and Matei Zaharia. Mlperf training benchmark. 2020.
- [98] T. G. Mattson, R. Cledat, V. Cav, V. Sarkar, Z. Budimli, S. Chatterjee, J. Fryman, I. Ganey, R. Knauerhase, Min Lee, B. Meister, B. Nickerson, N. Pepperling, B. Seshasayee, S. Tasirlar, J. Teller, and N. Vrvilo. The open community runtime: A runtime system for extreme scale computing. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, 2016.
- [99] Benoît Meister, Nicolas Vasilache, David Wohlford, M. Baskaran, Allen Leung, and R. Lethin. R-stream compiler. In *Encyclopedia of Parallel Computing*, 2011.
- [100] P. Micikevicius, Sharan Narang, J. Alben, G. Diamos, E. Elsen, D. García, B. Ginsburg, Michael Houston, O. Kuchaiev, G. Venkatesh, and H. Wu. Mixed precision training. 2018.
- [101] Hiroaki Mikami, Hisahiro Suganuma, Pongsakorn U.-Chupala, Y. Tanaka, and Y. Kageyama. Massively distributed sgd: Imagenet/resnet-50 training in a flash. *arXiv*, abs/1811.05233, 2018.
- [102] MLPerf. Mlperf reference: Minigo. <https://github.com/mlperf/training/tree/master/reinforcement>, 2019.
- [103] MLPerf. Mlperf reference: Resnet in tensorflow. https://github.com/mlperf/training/tree/master/image_classification/tensorflow/official, 2019.
- [104] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *ArXiv*, abs/1312.5602, 2013.
- [105] Sebastian Nanz, S. West, Kaue Soares da Silveira, and B. Meyer. Benchmarking usability and performance of multicore languages. *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 183–192, 2013.

- [106] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. Pipedream: Generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP '19*, page 115, New York, NY, USA, 2019. Association for Computing Machinery.
- [107] M. Naumov, D. Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, X. Wang, Udit Gupta, Carole-Jean Wu, A. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, I. Cherniavskii, Yinghai Lu, R. Krishnamoorthi, Ansha Yu, V. Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, V. Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. Deep learning recommendation model for personalization and recommendation systems. *ArXiv*, abs/1906.00091, 2019.
- [108] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. Latency-tolerant software distributed shared memory. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 291–305, Santa Clara, CA, 2015. USENIX Association.
- [109] NVIDIA. Nvidia tensorrt programmable inference accelerator. <https://developer.nvidia.com/tensorrt>.
- [110] OpenSHMEM. Openshmem specification. <http://www.openshmem.org>.
- [111] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: A method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics, ACL '02*, page 311318, USA, 2002. Association for Computational Linguistics.
- [112] Adam Paszke, S. Gross, Soumith Chintala, G. Chanan, E. Yang, Zachary Devito, Zeming Lin, Alban Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in pytorch. 2017.
- [113] I. Patel and J. R. Gilbert. An empirical study of the performance and productivity of two parallel programming models. In *2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–7, 2008.
- [114] M. Post. A call for clarity in reporting bleu scores. *ArXiv*, abs/1804.08771, 2018.
- [115] Qualcomm. Snapdragon neural processing engine sdk reference guide. <https://developer.qualcomm.com/docs/snpe/overview.html>.
- [116] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever. Language models are unsupervised multi-task learners. OpenAI Blog, 2019.
- [117] K.H. Randall. Cilk: Efficient multithreaded computing. <http://supertech.csail.mit.edu/papers/randall-phdthesis.pdf>.

- [118] V. J. Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling, C. Wu, B. Anderson, M. Breughe, M. Charlebois, W. Chou, R. Chukka, C. Coleman, S. Davis, P. Deng, G. Diamos, J. Duke, D. Fick, J. S. Gardner, I. Hubara, S. Idgunji, T. B. Jablin, J. Jiao, T. St. John, P. Kanwar, D. Lee, J. Liao, A. Lokhmotov, F. Massa, P. Meng, P. Micikevicius, C. Osborne, G. Pekhimenko, A. T. R. Rajan, D. Sequeira, A. Sirasao, F. Sun, H. Tang, M. Thomson, F. Wei, E. Wu, L. Xu, K. Yamada, B. Yu, G. Yuan, A. Zhong, P. Zhang, and Y. Zhou. Mlperf inference benchmark. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 446–459, 2020.
- [119] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L. Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4510–4520, 2018.
- [120] Frank Seide and Amit Agarwal. Cntk: Microsoft’s open-source deep-learning toolkit. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD ’16*, page 2135, New York, NY, USA, 2016. Association for Computing Machinery.
- [121] Hongzhang Shan, Nicholas J. Wright, John Shalf, Katherine Yelick, Marcus Wagner, and Nathan Wichmann. A preliminary evaluation of the hardware acceleration of the cray gemini interconnect for pgas languages and comparison with mpi. In *Proceedings of the Second International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computing Systems, PMBS ’11*, page 1314, New York, NY, USA, 2011. Association for Computing Machinery.
- [122] A. G. Shet, W. R. Elwasif, R. J. Harrison, and D. E. Bernholdt. Programmability of the hpcs languages: A case study with a quantum chemistry kernel. In *2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–8, 2008.
- [123] David Silver, Aja Huang, Christopher Maddison, Arthur Guez, Laurent Sifre, George Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–489, 01 2016.
- [124] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dhharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362:1140–1144, 12 2018.

- [125] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of go without human knowledge. *Nature*, 550:354–359, 10 2017.
- [126] Tom St. John, Benoît Meister, Andres Marquez, Joseph B. Manzano, Guang R. Gao, and Xiaoming Li. Asafesss: A scheduler-driven adaptive framework for extreme scale software stacks. In *Proceedings of International Workshop on Adaptive Self-Tuning Computing Systems*, ADAPT '14, page 2123, New York, NY, USA, 2014. Association for Computing Machinery.
- [127] P. Sun, W. Feng, R. Han, S. Yan, and Y. Wen. Optimizing network performance for distributed dnn training on gpu clusters: Imagenet/alexnet training in 1.5 minutes. *ArXiv*, abs/1902.06855, 2019.
- [128] A Tamhane and Dunlop Dunlop. *Statistics and Data Analysis: From Elementary to Intermediate*. Prentice-Hall, 2000.
- [129] S. Tang. Ai-chip. <https://basicmi.github.io/AI-Chip/>, 2019.
- [130] Michael B. Taylor. Is dark silicon useful? harnessing the four horsemen of the coming dark silicon apocalypse. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, page 11311136, New York, NY, USA, 2012. Association for Computing Machinery.
- [131] Principled Technologies. Aixprt community preview. <https://www.principledtechnologies.com/benchmarkxprt/aixprt/>, 2019.
- [132] Seiya Tokui and Kenta Oono. Chainer : a next-generation open source framework for deep learning. In *Proceedings of Workshop on Machine Learning Systems (LearningSys) in Neural Information Processing Systems (NeurIPS)*, 2015.
- [133] Sean Treichler, Michael Bauer, and Alex Aiken. Realm: An event-based low-level runtime for distributed memory architectures. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, page 263276, New York, NY, USA, 2014. Association for Computing Machinery.
- [134] UPC. Upc lang. spec. v. 1.3, 2013.
- [135] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 60006010, Red Hook, NY, USA, 2017. Curran Associates Inc.

- [136] Peng Wang, Xinyu Huang, Xinjing Cheng, Dingfu Zhou, Qichuan Geng, and Ruigang Yang. The apolloscape open dataset for autonomous driving and its application. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 42:2702–2719, 2020.
- [137] R. V. D. Wijngaart and T. Mattson. The parallel research kernels a tool for architecture and programming system investigation. HPEC '14, 2014.
- [138] Rob Wijngaart, Abdullah Kayi, Jeff Hammond, Gabriele Jost, Tom St. John, Srinivas Sridharan, Tim Mattson, John Abercrombie, and Jacob Nelson. Comparing runtime systems with exascale ambitions using the parallel research kernels. ISC '16, pages 321–339, 06 2016.
- [139] Rob Wijngaart, Srinivas Sridharan, Abdullah Kayi, Gabriele Jost, Jeff Hammond, Tim Mattson, and Jacob Nelson. Using the parallel research kernels to study pgas models. PGAS '15, pages 76–81, 09 2015.
- [140] WMT. First conference on machine translation. <http://statmt.org/wmt16/>, 2016.
- [141] WMT. Second conference on machine translation. <http://statmt.org/wmt17/>, 2017.
- [142] Y. Wu, Mike Schuster, Z. Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, M. Krikun, Yuan Cao, Q. Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, M. Johnson, X. Liu, L. Kaiser, S. Gouws, Y. Kato, Taku Kudo, H. Kazawa, K. Stevens, G. Kurian, Nishant Patil, W. Wang, C. Young, J. Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, G. S. Corrado, Macduff Hughes, and J. Dean. Google’s neural machine translation system: Bridging the gap between human and machine translation. *ArXiv*, abs/1609.08144, 2016.
- [143] S. Xie, R. Girshick, P. Dollr, Z. Tu, and K. He. Aggregated residual transformations for deep neural networks. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 5987–5995, 2017.
- [144] Danfei Xu, Dragomir Anguelov, and Ashesh Jain. Pointfusion: Deep sensor fusion for 3d bounding box estimation. *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 244–253, 2018.
- [145] Chris Ying, S. Kumar, Dehao Chen, T. Wang, and Youlong Cheng. Image classification at supercomputer scale. *ArXiv*, abs/1811.06992, 2018.
- [146] Yang You, Igor Gitman, and B. Ginsburg. Large batch training of convolutional networks. *arXiv*, abs/1708.03888, 2017.

- [147] F. Yu, Wenqi Xian, Yingying Chen, Fangchen Liu, Mike Liao, V. Madhavan, and Trevor Darrell. Bdd100k: A diverse driving video database with scalable annotation tooling. *ArXiv*, abs/1805.04687, 2018.
- [148] R. Zerr and R. Baker. Snap: Sn application proxy. Technical report, Los Alamos National Laboratory, 2013.
- [149] Guorui Zhou, Xiaoqiang Zhu, Chenru Song, Ying Fan, Han Zhu, Xiao Ma, Yanghui Yan, Junqi Jin, Han Li, and Kun Gai. Deep interest network for click-through rate prediction. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '18, page 10591068, New York, NY, USA, 2018. Association for Computing Machinery.
- [150] Chenzhuo Zhu, Song Han, Huizi Mao, and W. Dally. Trained ternary quantization. *ArXiv*, abs/1612.01064, 2016.
- [151] H. Zhu, M. Akrouf, B. Zheng, A. Pelegris, A. Jayarajan, A. Phanishayee, B. Schroeder, and G. Pekhimenko. Benchmarking and analyzing deep neural network training. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pages 88–100, 2018.
- [152] Stéphane Zuckerman, Joshua Suetterlein, Rob Knauerhase, and Guang R. Gao. Using a "codelet" program execution model for exascale machines: Position paper. In *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, EXADAPT '11, page 6469, New York, NY, USA, 2011. Association for Computing Machinery.