

A Survey of Machine Learning Approaches for Computer Architecture Performance Modeling

MICRO 2026 Submission #NaN – Confidential Draft – Do NOT Distribute!!

Abstract

Machine learning-based performance modeling has emerged as a powerful alternative to traditional analytical models and cycle-accurate simulators for predicting computer system behavior. This survey comprehensively analyzes ML techniques for performance prediction across CPUs, GPUs, accelerators, and distributed systems, covering over 60 papers from architecture and ML venues published between 2016–2025. We propose an eight-dimension taxonomy organizing approaches by modeling technique, target hardware, workload types, prediction targets, accuracy metrics, input requirements, evaluation scope, and reproducibility. Our analysis reveals that specialized models achieve remarkable accuracy—below 5% error for narrow domains—while general-purpose models trade accuracy for broader applicability. Transfer learning and meta-learning techniques increasingly enable adaptation to new hardware with minimal profiling, addressing the challenge of hardware diversity. We identify key open challenges including benchmark diversity, cross-platform generalization, and integration with compiler and architecture exploration workflows. Hybrid approaches combining analytical structure with learned components represent a promising direction, offering both interpretability and accuracy. This survey provides practitioners guidance for selecting appropriate techniques and researchers a roadmap for advancing the field.

Keywords

machine learning, performance modeling, computer architecture, neural networks, survey

1 Introduction

Performance modeling is fundamental to computer architecture research and development. Architects rely on accurate performance predictions to navigate vast design spaces, optimize hardware-software co-design, and make informed decisions about resource allocation. Traditional approaches—analytical models [14] and cycle-accurate simulators [3]—have served the community well, but face growing challenges as workloads and hardware become increasingly complex. Analytical models often oversimplify system behavior, while simulators can require hours or days to evaluate a single design point, making exhaustive exploration impractical.

The rise of deep learning workloads has intensified these challenges. Modern neural networks exhibit diverse computational patterns—from dense matrix operations in transformers to sparse irregular accesses in graph neural networks—that stress traditional modeling assumptions. Simultaneously, hardware diversity has exploded: GPUs, TPUs, custom accelerators, and multi-device distributed systems each present unique performance characteristics

that resist unified analytical treatment. This complexity has motivated a new generation of *machine learning-based* performance models that learn predictive functions directly from profiling data.

ML-based performance modeling has emerged as a compelling alternative. Learned models can capture complex, non-linear relationships between workload characteristics and hardware behavior that elude closed-form analysis. Recent work demonstrates remarkable accuracy: NeuSight [11] achieves 2.3% error predicting GPT-3 latency on H100 GPUs, while nn-Meter [18] reaches 99% accuracy for edge device latency prediction. Beyond accuracy, these approaches offer practical benefits: models trained on one platform can transfer to new hardware with minimal adaptation [6], and inference-time predictions complete in milliseconds rather than hours.

This survey provides a comprehensive analysis of ML-based performance modeling techniques for computer architecture. We make the following contributions:

- A **taxonomy** organizing approaches along eight dimensions: modeling technique, target hardware, workload types, prediction targets, accuracy metrics, input requirements, evaluation scope, and reproducibility.
- A **systematic survey** of over 60 papers from architecture venues (MICRO, ISCA, HPCA, ASPLOS) and ML venues (MLSys, NeurIPS, ICML) published between 2016–2025.
- A **comparative analysis** examining trade-offs between accuracy, training cost, generalization, and interpretability across approaches.
- An identification of **open challenges** including data scarcity, cross-platform generalization, and integration with design automation flows.

The remainder of this paper is organized as follows. Section 2 provides background on traditional performance modeling and relevant ML techniques. Section 3 presents our classification taxonomy. Section 4 surveys approaches organized by target hardware platform. Section 5 offers comparative analysis across key dimensions. Section 6 discusses open challenges and future directions. Section 7 presents hands-on reproducibility evaluations of representative tools. Section 8 concludes.

2 Background

2.1 Traditional Performance Modeling

Performance modeling has traditionally relied on two complementary approaches: analytical models and cycle-accurate simulation. This section reviews both paradigms and their limitations, motivating the emergence of ML-based alternatives.

2.1.1 Analytical Models. Analytical models express performance as closed-form functions of hardware and workload parameters. The roofline model [14] exemplifies this approach, bounding attainable performance by peak compute throughput and memory bandwidth.

Given operational intensity I (FLOP/byte), the roofline predicts performance as $P = \min(\pi, \beta \cdot I)$, where π is peak FLOPS and β is memory bandwidth. Despite its simplicity, roofline reasoning guides optimization by revealing compute-bound versus memory-bound regimes.

For DNN accelerators, analytical cost models have become standard practice. Timeloop [12] models data movement across memory hierarchies for any given mapping (loop order and tiling), computing access counts and energy from architectural parameters. MAESTRO [9] provides a data-centric framework that derives performance from dataflow descriptions. Sparseloop [16] extends this methodology to sparse tensor operations, achieving 2000 \times speedup over RTL simulation while maintaining accuracy.

Analytical models offer several advantages: fast evaluation (microseconds per design point), interpretability (designers can trace predictions to specific terms), and extrapolation to unseen configurations. However, they require manual derivation for each target architecture, struggle to capture complex microarchitectural effects (contention, pipeline stalls, caching behavior), and may oversimplify non-linear interactions.

2.1.2 Cycle-Accurate Simulation. Cycle-accurate simulators model hardware at register-transfer level, faithfully reproducing timing behavior. General-purpose simulators like gem5 [3] support flexible configuration of CPU cores, caches, memory controllers, and interconnects. For GPUs, simulators such as GPGPU-Sim [2] and Accel-Sim [7] model SIMT execution, warp scheduling, and memory coalescing.

Cycle-accurate simulation achieves high fidelity—typically within 5–15% of real hardware [3]—and supports detailed microarchitectural studies. However, simulation speed presents a fundamental limitation: evaluating a single ResNet-50 inference may require hours, making design space exploration impractical. ASTRA-sim [15] addresses distributed training at scale through analytical abstractions, but even coarse-grained simulation struggles with the combinatorial explosion of modern ML workloads and hardware configurations.

2.1.3 The Modeling Gap. Neither approach fully addresses modern performance modeling needs. Analytical models are fast but imprecise for complex microarchitectures. Simulators are accurate but too slow for iterative design. This tension has intensified as ML workloads diversify (from CNNs to transformers to mixture-of-experts models) and hardware specializes (GPUs, TPUs, custom accelerators). ML-based performance models offer a middle path: learning complex relationships from profiling data while enabling millisecond-scale inference.

2.2 Machine Learning Fundamentals

This section provides a brief primer on ML techniques frequently employed in performance modeling, establishing terminology used throughout the survey.

2.2.1 Classical Machine Learning. Linear regression and its regularized variants (ridge, LASSO) remain widely used for performance prediction due to their simplicity and interpretability. Given feature vector \mathbf{x} (e.g., operator parameters, hardware counters), linear

models predict $\hat{y} = \mathbf{w}^\top \mathbf{x} + b$. While unable to capture non-linear relationships, linear models provide baselines and feature importance rankings.

Tree-based ensembles—random forests and gradient boosted trees (XGBoost, LightGBM)—handle non-linearities through recursive partitioning. These methods dominate when training data is limited (<10K samples) and features are well-engineered, often outperforming deep learning in low-data regimes [18].

2.2.2 Deep Learning. Multi-layer perceptrons (MLPs) learn hierarchical feature representations through stacked non-linear transformations: $\mathbf{h}_{i+1} = \sigma(\mathbf{W}_i \mathbf{h}_i + \mathbf{b}_i)$. MLPs require minimal feature engineering but need sufficient training data and careful regularization to avoid overfitting.

Recurrent neural networks (RNNs) and their gated variants (LSTM, GRU) process sequential inputs, making them suitable for modeling operator sequences in neural network execution graphs. However, sequential processing limits parallelization and can miss long-range dependencies.

2.2.3 Graph Neural Networks. Graph neural networks (GNNs) operate on graph-structured data through message passing. For a node v with features \mathbf{h}_v , GNNs iteratively update representations by aggregating information from neighbors $N(v)$:

$$\mathbf{h}_v^{(k+1)} = \phi \left(\mathbf{h}_v^{(k)}, \bigoplus_{u \in N(v)} \psi(\mathbf{h}_u^{(k)}, \mathbf{e}_{uv}) \right) \quad (1)$$

where ϕ and ψ are learnable functions and \oplus is a permutation-invariant aggregation (sum, mean, max).

GNNs are particularly appealing for performance modeling because DNN computation graphs have natural graph structure. Nodes represent operators with features (type, parameters), edges represent data dependencies with features (tensor shapes, datatypes). GNNs can learn to propagate performance-relevant information along these dependencies [13].

2.2.4 Attention and Transformers. Attention mechanisms compute weighted combinations over input elements, with weights determined by learned compatibility functions. Self-attention allows each position to attend to all other positions:

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax} \left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}} \right) \mathbf{V} \quad (2)$$

Transformers stack self-attention with feedforward networks, enabling long-range dependency modeling without sequential processing. Recent performance models leverage transformer architectures to capture complex inter-operator interactions across entire computation graphs.

2.2.5 Transfer Learning. Transfer learning adapts models trained on one domain (source) to perform well on another (target). In performance modeling, this enables training on easily-profiled hardware and transferring to new platforms with limited data. Common approaches include fine-tuning (adjusting pre-trained weights with target data), domain adaptation (learning domain-invariant representations), and meta-learning (learning to adapt quickly from few examples) [6].

2.3 Problem Formulation

We now formally define the performance modeling problem and establish the evaluation framework used throughout this survey.

2.3.1 Inputs and Outputs. Performance modeling maps workload and hardware descriptions to performance metrics. Formally, given workload specification \mathcal{W} and hardware configuration \mathcal{H} , a performance model f predicts metric y :

$$\hat{y} = f(\mathcal{W}, \mathcal{H}; \theta) \quad (3)$$

where θ represents model parameters (weights for ML models, equations for analytical models).

Workload representations vary by granularity and abstraction:

- *Operator-level*: Individual layer parameters (kernel size, channels, batch size)
- *Graph-level*: Full computation graph with node and edge features
- *IR-level*: Intermediate representations from compilers (TVM [4], XLA)
- *Trace-level*: Execution traces capturing runtime behavior

Hardware representations similarly span multiple levels:

- *Specification*: Static parameters (core count, memory size, bandwidth)
- *Counter-based*: Runtime performance counters (cache misses, stalls)
- *Embedding*: Learned dense representations of hardware platforms

2.3.2 Prediction Targets. Performance models target various metrics depending on application requirements:

Latency measures execution time, typically end-to-end inference time or per-layer latency. Latency prediction is critical for real-time applications with strict deadlines and for optimizing user-facing services.

Throughput captures sustained processing rate: samples per second for inference, tokens per second for language models, or images per second for training. Throughput optimization maximizes hardware utilization for batch processing.

Energy encompasses power consumption (Watts) and energy per operation (Joules/inference). Energy prediction is essential for mobile deployment, data center cost optimization, and sustainability considerations.

Memory includes peak memory footprint (for feasibility checking), memory bandwidth utilization, and memory access patterns.

Multi-objective formulations jointly predict multiple metrics, enabling Pareto-optimal design selection balancing latency, energy, and accuracy.

2.3.3 Accuracy Metrics. The field employs several accuracy metrics, each with distinct interpretations:

Mean Absolute Percentage Error (MAPE) measures average relative deviation:

$$\text{MAPE} = \frac{100\%}{n} \sum_{i=1}^n \left| \frac{y_i - \hat{y}_i}{y_i} \right| \quad (4)$$

MAPE is scale-invariant and interpretable (5% MAPE means predictions typically differ by 5% from ground truth).

Root Mean Square Error (RMSE) penalizes large errors more heavily:

$$\text{RMSE} = \sqrt{\frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2} \quad (5)$$

Correlation coefficients (Pearson, Spearman) measure how well predictions track relative ordering—important when models guide design space exploration.

Ranking accuracy directly evaluates whether models correctly order configurations, often measured via Kendall's τ or top- k accuracy.

2.3.4 Hardware Targets. Modern performance modeling spans diverse hardware platforms:

CPUs remain important for general-purpose inference and training of smaller models. CPU modeling must account for complex cache hierarchies, branch prediction, out-of-order execution, and SIMD vectorization.

GPUs dominate ML training and large-scale inference. GPU modeling addresses SIMD execution, warp scheduling, memory coalescing, and multi-GPU scaling.

TPUs and custom accelerators employ specialized dataflows for matrix operations. Modeling these devices requires understanding systolic arrays, on-chip memory hierarchies, and dataflow mappings.

Edge devices (mobile SoCs, embedded NPUs) impose strict power and memory constraints. Edge modeling emphasizes latency under thermal throttling and memory-limited execution.

Distributed systems scale training across multiple devices and nodes. Distributed modeling must capture communication overhead, synchronization barriers, and pipeline parallelism.

This diversity of targets, workloads, and metrics motivates our comprehensive taxonomy in Section 3.

3 Taxonomy

We organize the surveyed literature along three primary dimensions: the hardware target being modeled, the machine learning techniques employed, and the input representations used. Figure 1 illustrates how these dimensions intersect to characterize different performance modeling approaches. This taxonomy extends existing classifications [9, 12] by incorporating the emerging diversity of ML-based methods and their distinctive design choices.

Our classification scheme serves two purposes. First, it provides a systematic framework for understanding the design space of ML-based performance models—researchers can identify which combinations of targets, techniques, and representations have been explored versus those that remain open. Second, it enables practitioners to select appropriate methods for their use cases by matching problem characteristics (target hardware, available data, accuracy requirements) to model capabilities.

3.1 By Modeling Target

The choice of hardware target fundamentally shapes model design, as different platforms exhibit distinct performance characteristics and modeling challenges.

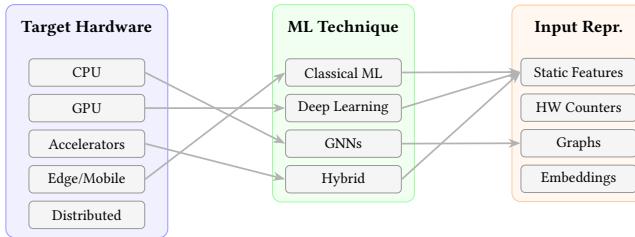


Figure 1: Three-dimensional taxonomy for ML-based performance modeling. Arrows indicate common pairings observed in the literature (e.g., GPU models often use deep learning with static features).

3.1.1 CPU Performance Modeling. CPUs present complex modeling challenges due to deep out-of-order pipelines, sophisticated cache hierarchies, and branch prediction. ML models for CPU performance must capture instruction-level parallelism, cache behavior, and memory access patterns. Traditional approaches relied on microbenchmark-based linear regression [3], while recent work employs graph neural networks to model basic block throughput [13]. CPU modeling remains challenging due to the diversity of microarchitectures and the difficulty of capturing dynamic effects like branch misprediction and cache contention.

3.1.2 GPU Performance Modeling. GPUs dominate modern ML training and inference, making accurate GPU performance prediction critical. GPU modeling must account for SIMD execution, warp scheduling, memory coalescing, and memory bandwidth limitations. Early approaches used analytical roofline models [14], but these struggle with the complex memory hierarchies and occupancy effects of modern GPUs.

ML-based GPU models have achieved remarkable accuracy. NeuSight [11] introduces tile-based prediction that mirrors CUDA’s execution model, achieving 2.3% error on GPT-3 inference across H100, A100, and V100 GPUs. Habitat [17] pioneered runtime-based cross-GPU prediction using wave scaling analysis. These approaches demonstrate that learned models can capture GPU performance characteristics that elude analytical treatment.

3.1.3 DNN Accelerator Modeling. Custom DNN accelerators—including TPUs, NPUs, and systolic array designs—employ specialized dataflows optimized for matrix operations. Modeling these devices requires understanding the interaction between dataflow, memory hierarchy, and tensor tiling.

Analytical frameworks like Timeloop [12] and MAESTRO [9] provide systematic approaches for accelerator design space exploration. Timeloop models data movement and compute utilization for any valid mapping of operations to hardware, achieving 5–10% accuracy versus RTL simulation at 2000× speedup. MAESTRO offers a data-centric perspective using intuitive dataflow directives. Sparseloop [16] extends these frameworks to sparse tensor operations, critical for efficient transformer inference.

ML-based approaches complement analytical models by learning residual corrections or capturing effects not modeled analytically. ArchGym [8] demonstrates that ML surrogate models can achieve

0.61% RMSE while providing 2000× speedup over simulation, enabling rapid design space exploration for accelerator development.

3.1.4 Edge and Mobile Device Modeling. Edge devices impose strict power, memory, and latency constraints, making accurate prediction essential for deploying ML models on mobile phones, IoT devices, and embedded systems. The diversity of edge hardware—spanning mobile CPUs, mobile GPUs, NPUs, and DSPs—creates significant challenges for cross-platform prediction.

nn-Meter [18] addresses this challenge through kernel-level prediction with adaptive sampling, achieving 99% accuracy across mobile CPUs, GPUs, and Intel VPUs. LitePred [6] extends this work with transfer learning, achieving 99.3% accuracy across 85 edge platforms with less than one hour of adaptation per new device. These results demonstrate that ML models can effectively generalize across the heterogeneous edge hardware landscape.

3.1.5 Distributed System Modeling. Multi-GPU and multi-node systems introduce communication overhead, synchronization barriers, and parallelism strategy choices that fundamentally change performance characteristics. Distributed training performance depends on the interplay between compute, memory bandwidth, and network communication.

ASTRA-sim [15] provides end-to-end distributed training simulation, modeling collective communication algorithms, network topology, and compute-communication overlap. VIDUR [1] focuses specifically on LLM inference serving, capturing the unique characteristics of prefill and decode phases, KV cache management, and request scheduling. These simulation frameworks achieve 5–15% accuracy versus real clusters while enabling exploration of parallelization strategies at scale.

3.2 By ML Technique

The choice of ML technique reflects trade-offs between accuracy, data efficiency, interpretability, and generalization capability.

3.2.1 Classical Machine Learning. Tree-based ensembles—random forests and gradient boosted trees (XGBoost, LightGBM)—remain highly effective for performance modeling, particularly in low-data regimes. These methods handle non-linear relationships through recursive partitioning, provide feature importance rankings for interpretability, and require minimal hyperparameter tuning.

Classical ML models dominate when training data is limited (<10K samples) or when features are well-engineered. nn-Meter [18] demonstrates that random forests achieve competitive accuracy with careful kernel-level feature engineering. The ALCOP framework combines XGBoost with analytical pre-training, using analytical model predictions as features to accelerate autotuning convergence.

3.2.2 Deep Learning. Multi-layer perceptrons (MLPs) learn hierarchical feature representations without manual feature engineering. MLPs are widely used as the prediction head in more complex architectures and as standalone models when sufficient training data is available. NeuSight [11] uses MLPs to predict tile-level GPU utilization, learning complex interactions between tile parameters and hardware characteristics.

465 Recurrent neural networks (RNNs and LSTMs) process sequential
 466 inputs, making them suitable for modeling operator sequences
 467 in neural network execution. However, sequential processing limits
 468 parallelization, and attention-based architectures increasingly
 469 replace RNNs for sequence modeling tasks.
 470

471 **3.2.3 Graph Neural Networks.** Graph neural networks (GNNs) have
 472 emerged as particularly effective for performance modeling because
 473 computational graphs have natural graph structure. Nodes represent
 474 operators with features (type, parameters, shapes), edges represent
 475 data dependencies with features (tensor dimensions, datatypes).
 476 GNNs propagate performance-relevant information along these de-
 477 pendencies through message passing.

478 GRANITE [13] applies GNNs to basic block throughput estimation,
 479 learning to predict CPU performance from instruction dependency
 480 graphs. For DNN workloads, GNN-based models capture
 481 inter-operator interactions that flat feature representations miss.
 482 The graph structure also enables natural handling of variable-size
 483 networks without padding or truncation.

484 **3.2.4 Hybrid Analytical+ML Models.** Hybrid approaches combine
 485 physics-based analytical models with learned components, achiev-
 486 ing both interpretability and high accuracy. The analytical com-
 487 ponent provides a strong prior based on hardware characteristics,
 488 while the ML component learns residual corrections and complex
 489 interactions.
 490

491 This design philosophy has produced state-of-the-art results.
 492 Analytical pre-training initializes ML models with reasonable pre-
 493 dictions, reducing data requirements and improving convergence.
 494 Physics-informed architectures incorporate analytical insights into
 495 model structure—NeuSight’s tile-based prediction mirrors CUDA’s
 496 execution model, providing inductive bias that improves general-
 497 ization. Residual learning trains ML models to predict the error of
 498 analytical models, combining analytical interpretability with ML’s
 499 ability to capture unmodeled effects.

500 The latency predictor study [5] demonstrates that hybrid ap-
 501 proaches with transfer learning achieve 22.5% average improve-
 502 ment over baselines, with up to 87.6% improvement on challenging
 503 cross-platform prediction tasks.

504 **3.3 By Input Representation**

505 Input representation determines what information the model can
 506 access and how effectively it can learn performance-relevant pat-
 507 terns.
 508

509 **3.3.1 Static Features.** Static features derive from workload and
 510 hardware specifications without runtime measurement. For DNN
 511 workloads, these include layer parameters (kernel size, channels,
 512 stride, batch size), tensor dimensions, and operator types. Hardware
 513 specifications include core counts, memory sizes, bandwidth, and
 514 clock frequencies.
 515

516 Static features enable prediction without profiling, supporting
 517 use cases like neural architecture search where thousands of can-
 518 didate networks must be evaluated. Feature engineering plays
 519 a critical role: effective representations capture computation-to-
 520 communication ratios, memory footprint estimates, and paralleliza-
 521 tion potential.

523 **3.3.2 Hardware Counters.** Performance counters provide runtime
 524 measurements of hardware behavior: cache miss rates, memory
 525 bandwidth utilization, instruction throughput, and stall cycles. Counter-
 526 based models can capture dynamic effects invisible to static analysis,
 527 including contention, thermal throttling, and runtime scheduling
 528 decisions.
 529

530 The primary limitation is that counter-based models require hard-
 531 ware execution, limiting their applicability for design space explo-
 532 ration or new architecture evaluation. However, for optimizing ex-
 533 isting deployments or debugging performance anomalies, counter-
 534 based models provide valuable insights that static approaches can-
 535 not match.
 536

537 **3.3.3 Graph Representations.** Graph representations encode com-
 538 putational graphs with nodes representing operators and edges
 539 representing data dependencies. Node features capture operator
 540 characteristics (type, parameters), while edge features encode ten-
 541 sor properties (shape, datatype, memory format).
 542

543 Graph representations provide several advantages over flat fea-
 544 ture vectors: they naturally handle variable-size networks, pre-
 545 serve structural information about operator interactions, and enable
 546 permutation-invariant predictions. GNNs operating on these repre-
 547 sentations can learn which subgraph patterns indicate performance
 548 bottlenecks.
 549

550 **3.3.4 Learned Embeddings.** Learned embeddings compress high-
 551 dimensional or categorical information into dense vector represen-
 552 tations. Hardware embeddings represent diverse devices as points
 553 in a learned feature space, enabling transfer learning across plat-
 554 forms. Operator embeddings capture semantic similarities between
 555 operator types that may share performance characteristics.
 556

557 HELP formulates hardware prediction as meta-learning, learning
 558 hardware embeddings that represent devices as black-box functions.
 559 With just 10 measurement samples on a new device, HELP achieves
 560 accurate predictions by positioning the device appropriately in the
 561 learned embedding space. This approach is particularly valuable
 562 for the fragmented edge hardware landscape, where collecting
 563 exhaustive training data for each device is impractical.
 564

565 Table 1 summarizes representative papers across our taxonomy
 566 dimensions, illustrating the diversity of approaches and their key
 567 characteristics.
 568

569 **4 Survey of Approaches**

570 This section surveys ML-based performance modeling approaches
 571 organized by target hardware platform. For each category, we ex-
 572 amine the modeling challenges specific to that platform, describe
 573 representative techniques, and synthesize key findings across the
 574 literature. Table 2 provides a comprehensive comparison of the
 575 surveyed approaches.
 576

577 **4.1 CPU Performance Modeling**

578 CPU performance modeling for ML workloads presents unique
 579 challenges due to complex microarchitectural effects including out-
 580 of-order execution, branch prediction, and deep cache hierarchies.
 581 While GPUs have received more attention for DNN training, CPUs
 582 remain important for inference—particularly on edge devices and
 583 for operators that map poorly to SIMD execution.
 584

581 **Table 1: Representative papers classified by our taxonomy dimensions. Accuracy reported as MAPE or correlation where
582 available.**

Paper	Target	Technique	Input	Accuracy	Key Contribution
NeuSight [11]	GPU	Hybrid	Static	2.3%	Tile-based prediction
nn-Meter [18]	Edge	Classical ML	Static	<5%	Kernel detection
LitePred [6]	Edge	Transfer	Static	0.7%	85-platform transfer
GRANITE [13]	CPU	GNN	Graph	0.97 corr	Basic block modeling
Timeloop [12]	Accelerator	Analytical	Static	5–10%	Loop-nest DSE
ASTRA-sim [15]	Distributed	Simulation	Traces	5–15%	Collective modeling
ArchGym [8]	Accelerator	Hybrid	Static	0.61% RMSE	ML-aided DSE

593 **Table 2: Summary of surveyed ML-based performance modeling approaches, organized by target hardware platform.**

Paper	Platform	ML Technique	Prediction Target	Error	Key Innovation
<i>CPU Performance Modeling</i>					
GRANITE [13]	CPU	GNN	Basic block throughput	0.97 corr	Instruction graph encoding
gem5+ML [3]	CPU	Hybrid	Execution time	10–20%	Simulation + learning
<i>GPU Performance Modeling</i>					
NeuSight [11]	GPU	Hybrid MLP	Kernel/E2E latency	2.3%	Tile-based prediction
Habitat [17]	GPU	MLP	Training time	11.8%	Wave scaling analysis
Accel-Sim [7]	GPU	Simulation	Cycle-accurate	10–20%	SASS trace-driven
<i>DNN Accelerator Modeling</i>					
Timeloop [12]	NPU	Analytical	Latency/Energy	5–10%	Loop-nest DSE
MAESTRO [9]	NPU	Analytical	Latency/Energy	5–15%	Data-centric directives
Sparseloop [16]	NPU	Analytical	Sparse tensors	5–10%	Compression modeling
ArchGym [8]	Multi	RL+Surrogate	Multi-objective	0.61%	ML-aided DSE
<i>Edge Device Modeling</i>					
nn-Meter [18]	Edge	RF ensemble	Latency	<1%	Kernel detection
LitePred [6]	Edge	VAE+MLP	Latency	0.7%	85-platform transfer
HELP [10]	Multi	Meta-learning	Latency	1.9%	10-sample adaptation
<i>Distributed and LLM Systems</i>					
ASTRA-sim [15]	Distributed	Simulation	Training time	5–15%	Collective modeling
VIDUR [1]	GPU cluster	Simulation	LLM serving	<5%	Prefill/decode phases

617 **4.1.1 Traditional CPU Performance Modeling.** Traditional CPU
618 modeling relies on cycle-accurate simulation through frameworks
619 like gem5 [3]. The gem5 simulator provides multiple fidelity levels:
620 fast functional simulation for correctness validation, and detailed
621 out-of-order models achieving 10–20% accuracy versus real hard-
622 ware. For ML workloads, gem5 extensions such as gem5-Aladdin
623 and SMAUG enable accelerator integration studies.

624 However, cycle-accurate simulation suffers from fundamental
625 speed limitations—simulating even modest DNN inference requires
626 hours, making design space exploration impractical. This limitation
627 has motivated ML-based alternatives that learn to predict per-
628 formance from static program features.

632 **4.1.2 ML-Based Basic Block Modeling.** GRANITE [13] represents
633 the state of the art in ML-based CPU performance modeling. The
634 key insight is that basic block throughput—the steady-state exe-
635 cution rate of a loop body—can be predicted from the instruction
636 dependency graph without simulation. GRANITE encodes basic
637 blocks as directed graphs where nodes represent instructions with

641 features (opcode, operand types) and edges capture data dependen-
642 cies.

643 A graph neural network processes this representation through
644 message passing layers:

$$\mathbf{h}_i^{(k+1)} = \text{MLP} \left(\mathbf{h}_i^{(k)} + \sum_{j \in N(i)} \mathbf{h}_j^{(k)} \right) \quad (6)$$

683 where $\mathbf{h}_i^{(k)}$ represents instruction i 's embedding at layer k . After
684 several message passing rounds, a global pooling operation aggre-
685 gates instruction embeddings into a single block representation,
686 which a final MLP maps to throughput prediction.

687 GRANITE achieves 0.97 Kendall's τ correlation with ground-
688 truth measurements on x86 basic blocks, significantly outperform-
689 ing prior analytical models like IACA and llvm-mca. Critically,
690 the learned model generalizes across microarchitectures—a model
691 trained on Skylake transfers to Haswell with only modest accuracy
692 degradation.

693 **4.1.3 Challenges and Opportunities.** Despite GRANITE's success,
694 several challenges remain for CPU performance modeling. First,

697 DNN operators often involve memory-bound execution where
 698 cache behavior dominates—GRANITE focuses on compute-bound
 699 basic blocks and does not model memory hierarchy effects. Sec-
 700 ond, modern CPUs feature increasingly complex prefetchers and
 701 branch predictors whose behavior is difficult to capture in static
 702 features. Third, CPU-based DNN inference often involves highly
 703 optimized library code (Intel MKL, ARM Compute Library) whose
 704 performance depends on runtime scheduling decisions.

705 Hybrid approaches combining coarse-grained simulation with
 706 learned correction factors represent a promising direction. Rather
 707 than simulating every cycle, these methods use fast simulation
 708 to establish approximate behavior, then train ML models to pre-
 709 dict residual errors, potentially achieving simulation accuracy at
 710 reduced cost.

4.2 GPU Performance Modeling

712 GPUs are the dominant platform for ML training and large-scale
 713 inference. Accurate GPU performance prediction is essential for
 714 neural architecture search, compiler optimization, and serving sys-
 715 tem design. However, GPU performance modeling is challenging
 716 due to SIMT execution, complex memory hierarchies, and workload-
 717 dependent scheduling behavior.

718 **4.2.1 Cycle-Accurate GPU Simulation.** GPGPU-Sim [2] pioneered
 719 detailed GPU simulation, modeling SIMT cores, warp scheduling,
 720 memory coalescing, and cache hierarchies. Accel-Sim [7] extended
 721 this foundation with trace-driven simulation and improved corre-
 722 lation with modern GPUs (Turing, Ampere), achieving 0.90–0.97 IPC
 723 correlation.

724 These simulators provide high fidelity—essential for microarchi-
 725 tectural studies—but suffer from 1000–10000× slowdown versus
 726 real GPU execution. Simulating a single ResNet-50 inference can
 727 require hours, making design space exploration impractical. This
 728 has motivated the development of ML-based predictors that achieve
 729 comparable accuracy at dramatically reduced cost.

730 **4.2.2 Learned GPU Performance Models.** Habitat [17] introduced
 731 *wave scaling* for cross-GPU prediction. The key insight is that GPU
 732 execution time can be decomposed into compute and memory com-
 733 ponents that scale differently across devices:

$$734 T_{\text{target}} = T_{\text{compute}} \cdot \frac{P_{\text{source}}}{P_{\text{target}}} + T_{\text{memory}} \cdot \frac{B_{\text{source}}}{B_{\text{target}}} \quad (7)$$

735 where P denotes peak compute throughput and B memory band-
 736 width. By profiling on a source GPU and measuring how kernels
 737 respond to artificially reduced parallelism (“wave scaling”), Habitat
 738 estimates the compute and memory fractions, enabling prediction
 739 on unseen target GPUs.

740 Habitat achieves 11.8% average error predicting training iteration
 741 time across GPU generations (V100 to A100). However, it requires
 742 actual GPU execution for wave scaling measurements and cannot
 743 predict performance for unseen models.

744 NeuSight [11] addresses these limitations through *tile-based pre-*
 745 *diction*. The key innovation is decomposing GPU kernel execution
 746 into tiles—the basic scheduling unit in CUDA—and predicting per-
 747 tile behavior:

$$748 T_{\text{kernel}} = \max_{w \in \text{waves}} \sum_{t \in w} \left(T_{\text{compute}}^{(t)} + T_{\text{memory}}^{(t)} \right) \quad (8)$$

755 This formulation mirrors actual GPU execution semantics: tiles
 756 are scheduled in waves, and kernel time is dominated by the slow-
 757 est wave. NeuSight uses MLPs to predict tile-level compute and
 758 memory times from static features (tile dimensions, register usage,
 759 shared memory allocation).

760 By capturing the wave-level structure, NeuSight achieves re-
 761 markable accuracy: 2.3% error on GPT-3 inference across H100,
 762 A100, and V100 GPUs. This represents a 50× reduction in error
 763 compared to prior approaches like Habitat (121.4% → 2.3% on H100
 764 for GPT-3). NeuSight’s physics-informed architecture—encoding
 765 GPU execution semantics into the model structure—provides strong
 766 inductive bias that enables generalization to unseen models and
 767 GPUs.

768 **4.2.3 Compiler Cost Models for GPUs.** The TVM [4] and Ansor [19]
 769 systems use learned cost models to guide tensor program optimiza-
 770 tion. Rather than executing every candidate program, XGBoost or
 771 MLP models predict execution time from program features (loop
 772 bounds, vectorization widths, memory access patterns).

773 Ansor’s hierarchical search combines sketch generation, random
 774 annotation, and evolutionary refinement, using the cost model to
 775 prune the search space. With 10K profiled samples, Ansor achieves
 776 approximately 15% MAPE on GPU kernel prediction. The TenSet
 777 dataset provides 52 million program performance records across
 778 CPUs and GPUs, enabling pre-trained cost models that accelerate
 779 autotuning convergence by 10×.

780 **4.2.4 LLM Inference Prediction.** Large language model inference
 781 presents unique GPU modeling challenges. LLM execution exhibits
 782 distinct *prefill* (compute-bound, parallel prompt processing) and
 783 *decode* (memory-bound, sequential token generation) phases with
 784 fundamentally different performance characteristics.

785 VIDUR [1] provides discrete-event simulation for LLM serving
 786 systems. Rather than modeling GPU microarchitecture, VIDUR sim-
 787 ulates request scheduling, KV cache management, and batching
 788 decisions—the system-level factors that dominate serving perfor-
 789 mance. VIDUR achieves <5% error on end-to-end serving metrics
 790 including time-to-first-token and request latency.

791 Roofline-LLM extends traditional roofline analysis to LLM infer-
 792 ence by decomposing transformer execution into compute-bound
 793 (*prefill* attention, FFN) and memory-bound (*decode* attention, KV
 794 cache access) components. Combined with learned correction fac-
 795 tors, this hybrid approach achieves 87% reduction in MSE compared
 796 to pure roofline predictions.

4.3 Accelerator Performance Modeling

797 DNN accelerators—including TPUs, NPUs, and custom ASIC designs—
 798 employ specialized dataflows and memory hierarchies optimized
 799 for tensor operations. Modeling these devices requires understand-
 800 ing the interaction between dataflow choices, memory hierarchy
 801 utilization, and workload characteristics.

802 **4.3.1 Analytical Accelerator Modeling.** Timeloop [12] provides the
 803 foundational framework for DNN accelerator design space explo-
 804 ration. The key insight is that accelerator performance can be accu-
 805 rately predicted from loop-nest representations of tensor computa-
 806 tions. For a given architecture specification and mapping (loop
 807 order, tiling, spatial distribution), Timeloop analytically computes:

- 813 • **Data reuse** at each memory level: how many times each
 814 tensor element is accessed from each buffer
 815 • **Latency**: compute cycles plus memory stall cycles based
 816 on bandwidth constraints
 817 • **Energy**: access counts multiplied by per-access energy at
 818 each memory level

819 Timeloop decouples architecture specification (PEs, buffer sizes,
 820 bandwidth) from mapping decisions, enabling systematic explo-
 821 ration of dataflow choices. The framework achieves 5–10% accuracy
 822 versus RTL simulation while providing 2000× speedup, making
 823 million-point design sweeps tractable.

824 MAESTRO [9] offers a complementary *data-centric* perspective.
 825 Rather than loop-nest transformations, MAESTRO models perfor-
 826 mance through data movement analysis using compact dataflow
 827 directives. This representation is more intuitive—designers specify
 828 how tensors flow through the architecture rather than manipulating
 829 loop indices—while achieving comparable accuracy.

830 Sparseloop [16] extends analytical modeling to sparse tensor ac-
 831 celerators. The key challenge is that sparse execution time depends
 832 on runtime sparsity patterns, not just static tensor dimensions.
 833 Sparseloop models compression formats (CSR, bitmap, RLE), gating
 834 logic, and sparse-dense conversion overhead, enabling accurate pre-
 835 diction for pruned neural networks and sparse attention patterns.

836 4.3.2 *ML-Augmented Accelerator Design*. ArchGym [8] demon-
 837 strates how ML-based surrogate models can accelerate accelerator
 838 design. The framework connects ML optimization algorithms (rein-
 839 forcement learning, Bayesian optimization, evolutionary strategies)
 840 to hardware simulators through a unified interface.

841 A key finding is the *hyperparameter lottery*: ML algorithms show
 842 high variance across hyperparameter choices, with optimal settings
 843 differing substantially between target designs. ArchGym addresses
 844 this through systematic hyperparameter sweeps enabled by fast
 845 surrogate models. Trained surrogate models achieve 0.61% RMSE
 846 while providing 2000× speedup over simulation, enabling explo-
 847 ration of hyperparameter configurations that would be intractable
 848 with direct simulation.

849 4.3.3 *FPGA and Emerging Accelerator Modeling*. FPGA-based accelerators present additional modeling challenges due to the flexibility
 850 of reconfigurable fabric and the complexity of HLS-generated data-
 851 paths. Recent work applies transfer learning to FPGA design space
 852 exploration: models trained on one design can adapt to new archi-
 853 tectures with limited additional profiling.

854 Emerging accelerators—including processing-in-memory (PIM),
 855 neuromorphic, and analog compute-in-memory designs—remain
 856 underexplored. These platforms exhibit fundamentally different per-
 857 formance characteristics (energy-dominated by activations, analog
 858 noise effects, sparse event-driven computation) that existing frame-
 859 works do not address. Developing unified modeling approaches
 860 for this diverse hardware landscape represents an important open
 861 challenge.

862 4.4 Memory System Modeling

863 Memory system behavior increasingly dominates ML workload
 864 performance. Large language models may require hundreds of
 865 gigabytes for weights and KV cache, while training workloads

866 stress memory bandwidth through gradient communication. Accu-
 867 rate memory modeling is essential for understanding performance
 868 across the modern hardware landscape.

869 4.4.1 *Cache and Memory Hierarchy Modeling*. Traditional memory
 870 system modeling relies on cache simulation within frameworks like
 871 gem5 [3] and GPGPU-Sim [2]. These simulators model replacement
 872 policies, bank conflicts, memory coalescing (for GPUs), and DRAM
 873 controller behavior with high fidelity.

874 For DNN workloads, memory access patterns are often highly
 875 regular—streaming through weight and activation tensors—making
 876 analytical prediction feasible. Timeloop [12] models memory hier-
 877 archy through data reuse analysis: given a tiling and loop order, the
 878 framework computes exact access counts at each memory level. This
 879 analytical approach achieves high accuracy for regular workloads
 880 but may miss dynamic effects like cache contention in multi-tenant
 881 scenarios.

882 4.4.2 *KV Cache for LLM Inference*. KV cache management has
 883 emerged as the dominant memory challenge for LLM serving. The
 884 attention mechanism requires storing key-value tensors for all
 885 previously generated tokens, with memory growing linearly with
 886 sequence length and batch size. For long-context models serving
 887 concurrent requests, KV cache can consume hundreds of gigabytes.

888 vLLM’s PagedAttention introduces virtual memory concepts to
 889 KV cache management. By storing KV blocks in non-contiguous
 890 physical memory with page tables for address translation, PagedAt-
 891 tention achieves near-zero memory waste from fragmentation. This
 892 system-level optimization yields 2–4× throughput improvement
 893 over prior approaches.

894 VIDUR [1] models KV cache behavior at the serving system level,
 895 simulating allocation, eviction, and paging decisions that affect
 896 request latency. More recent work explores KV cache compres-
 897 sion through quantization (Oaken), sparsity (ALISA), and adaptive
 898 token selection (MorphKV), with potential memory savings ex-
 899 ceeding 50%. Accurate performance models for these compression
 900 techniques—predicting the latency-accuracy tradeoff for different
 901 compression levels—remain an open challenge.

902 4.4.3 *Distributed Memory and Communication*. Multi-GPU and
 903 multi-node training introduces communication overhead that can
 904 dominate performance at scale. ASTRA-sim [15] provides end-to-
 905 end simulation of distributed training, modeling collective com-
 906 munication algorithms (ring, tree, halving-doubling all-reduce),
 907 network topology, and compute-communication overlap.

908 The simulation decomposes collective operations into point-to-
 909 point messages, tracks network contention, and models the interac-
 910 tion between computation and communication phases. ASTRA-sim
 911 achieves 5–15% error versus real multi-GPU clusters, enabling ex-
 912 ploration of parallelization strategies (data parallel, model parallel,
 913 pipeline parallel) before expensive hardware experiments.

914 A key insight from distributed training modeling is that commu-
 915 nication overhead depends strongly on message granularity and
 916 overlap opportunities. Chunked gradient communication, where
 917 gradients are transmitted in pieces overlapped with backward pass
 918 computation, can hide communication latency. Accurate modeling
 919 of this overlap—which depends on operator ordering, chunk sizes,

929 and network bandwidth—is essential for predicting distributed training
 930 performance.

932 4.5 Cross-Platform and Transfer Learning

933 The proliferation of hardware platforms—from edge devices to
 934 datacenter GPUs to custom accelerators—creates demand for per-
 935 formance models that generalize across configurations. Training
 936 separate models for each target device is impractical given the diver-
 937 sity of the hardware landscape. Transfer learning and meta-learning
 938 approaches address this challenge by learning shared representa-
 939 tions that adapt efficiently to new platforms.

940 **4.5.1 Hardware-Adaptive Latency Prediction.** HELP [10] formulates
 941 cross-hardware prediction as meta-learning. The key insight is that
 942 hardware platforms can be treated as “tasks” in meta-learning: each
 943 device provides a small sample of profiled networks, and the goal
 944 is rapid adaptation to new devices.

945 HELP learns:

- 946 • **Architecture encoder:** A GNN that embeds neural net-
 947 work architectures into a fixed-dimensional space
- 948 • **Hardware encoder:** A learned function that represents
 949 devices from their profiled samples
- 950 • **Predictor:** An MLP that maps (architecture, hardware)
 951 pairs to latency

952 Using MAML-style meta-learning, HELP achieves 93.2% accuracy
 953 with just 10 profiled samples on new devices, reaching 98.1% with
 954 100 samples. This sample efficiency is critical for the fragmented
 955 edge hardware landscape where collecting exhaustive training data
 956 for each device type is impractical.

957 **4.5.2 Transfer Learning at Scale.** LitePred [6] scales cross-platform
 958 prediction to 85 edge devices—the most comprehensive evalua-
 959 tion to date. The framework introduces a VAE-based data sampler that
 960 intelligently selects which architectures to profile on new devices.
 961 Rather than random sampling, the VAE identifies architectures
 962 that are most informative for learning the device’s performance
 963 characteristics.

964 With less than one hour of profiling on a new device, LitePred
 965 achieves 99.3% accuracy on held-out architectures. This combines
 966 pre-trained representations from source platforms with efficient
 967 adaptation, demonstrating that the cross-platform transfer learning
 968 problem is tractable even at scale.

969 The latency predictors study [5] provides a systematic compari-
 970 son of transfer learning approaches for NAS. Key findings include:

- 971 • End-to-end training on pooled multi-platform data outper-
 972 forms sequential fine-tuning
- 973 • Transfer learning provides 22.5% average improvement over
 974 training from scratch
- 975 • Benefits are largest for challenging cross-platform transfers
 976 (up to 87.6% improvement)

977 **4.5.3 Hybrid Analytical-ML Transfer.** Hybrid approaches combine
 978 analytical models with learned components to improve transfer
 979 efficiency. SynPerf decomposes GPU kernel execution into pipeline
 980 demands (compute, memory, cache) using analytical models, then
 981 trains MLPs to capture cross-pipeline interactions. The analyti-
 982 cal decomposition provides physics-based structure that transfers
 983

984 across GPUs, while the learned component captures device-specific
 985 effects.

986 This hybrid architecture achieves 6.1% kernel-level error and
 987 has been applied to guide Triton kernel optimization, demon-
 988 strating 1.7× speedup on generated kernels. The combination of inter-
 989 pretive analytical structure with learned flexibility represents a
 990 promising direction for transferable performance modeling.

991 **4.5.4 Open Challenges in Transfer Learning.** Despite progress, sev-
 992 eral challenges remain. First, most transfer learning work focuses
 993 on CNN architectures; transformers and mixture-of-experts models
 994 remain underexplored. Second, transfer across *workload types* (not
 995 just hardware) is challenging—models trained on vision networks
 996 may not transfer to language models or graph neural networks.
 997 Third, continual learning for performance models—adapting to
 998 hardware and software evolution over time—is largely unexplored.

999 Foundation models for performance prediction represent an
 1000 emerging opportunity. Pre-trained on large-scale profiling datasets
 1001 spanning diverse architectures and hardware, such models could
 1002 provide strong initialization for any new prediction task. The TenSet
 1003 dataset with 52 million records represents a step in this direction,
 1004 but comprehensive datasets covering the full range of modern work-
 1005 loads and hardware remain to be developed

1006 5 Comparison and Analysis

1007 Having surveyed the landscape of ML-based performance model-
 1008 ing approaches, we now provide a comparative analysis across key
 1009 dimensions, including commonly used analytical and simulation-
 1010 based baselines. This analysis synthesizes trade-offs that practi-
 1011 tioners face when selecting or developing performance models,
 1012 examining accuracy, training cost, generalization, and interpretabil-
 1013 ity. Table 3 provides a comprehensive comparison across these
 1014 dimensions.

1015 5.1 Accuracy vs. Training Cost

1016 A fundamental trade-off exists between prediction accuracy and the
 1017 cost of data collection and model training. We analyze this trade-off
 1018 across the surveyed approaches, identifying regimes where different
 1019 techniques excel.

1020 **5.1.1 Data Collection Overhead.** The cost of obtaining training
 1021 data varies dramatically across approaches. *Profiling-based methods*
 1022 require executing workloads on target hardware, with costs ranging
 1023 from minutes (single operators) to hours (full model sweeps). nn-
 1024 Meter [18] requires approximately 1,000 profiled samples per kernel
 1025 type per device, translating to several hours of automated measure-
 1026 ment. LitePred [6] reduces this to approximately 100 samples for
 1027 new devices through intelligent VAE-based sampling.

1028 *Simulation-based training* uses cycle-accurate or analytical sim-
 1029 ulators as ground truth. ArchGym [8] trains surrogate models on
 1030 Timeloop [12] outputs, avoiding real hardware entirely but requir-
 1031 ing validated simulator configurations. This approach achieves
 1032 0.61% RMSE while providing 2000× speedup over direct simulation.

1033 *Transfer learning* amortizes data collection across platforms.
 1034 HELP [10] demonstrates that meta-learning enables 93.2% accu-
 1035 racy with just 10 samples on new devices, reaching 98.1% with 100

Table 3: Comparative analysis of representative performance models—including ML-based and analytical/simulation approaches—across key dimensions. The Accuracy column reports the metric and value as given in each original work (e.g., MAPE, RMSE, Kendall’s τ , ranges).

Model	Accuracy (as reported)	Training Data	Adaptation Cost	Generalization	Interpretability	Inference Time
<i>Classical ML</i>						1108
nn-Meter [18]	<1% MAPE	1K/kernel	Hours/device	Device-specific	Medium	1109
XGBoost (TVM) [4]	20% MAPE	10K+	Online	Operator-level	Medium	1110
<i>Deep Learning</i>						1111
NeuSight [11]	2.3% MAPE	100K+	Pre-trained	Cross-GPU	Low	1112
Habitat [17]	11.8% MAPE	Online profiling runs	None (requires GPU)	Cross-GPU	Medium	Per-kernel profiling 1113 1114
<i>Graph Neural Networks</i>						1115
GRANITE [13]	0.97 τ	10K+	Hours	Cross- μ arch	Low	1116
HELP [10]	1.9% MAPE	Meta-training	10 samples	Cross-platform	Low	1117
<i>Transfer Learning</i>						1118
LitePred [6]	0.7% MAPE	85 platforms	100 samples	85+ devices	Low	<1ms 1119
<i>Hybrid Analytical+ML</i>						1120
Timeloop [12]	5–10%	Arch spec	None	Any accelerator	High	μ s 1121
ArchGym [8]	0.61% RMSE	Simulation	Surrogate training	Architecture-specific	Medium	ms 1122
VIDUR [1]	<5%	Kernel profiles	Per-model	LLM-specific	High	ms 1123

samples. This sample efficiency is critical for the fragmented edge hardware landscape.

5.1.2 Model Training Cost. Training complexity varies from minutes for classical ML to days for large-scale pre-training. Tree-based ensembles (random forests, XGBoost) train in minutes on modest datasets and require minimal hyperparameter tuning. Deep learning models require careful architecture design, regularization, and often GPU training, but can achieve higher accuracy on large datasets.

The TenSet dataset [20] with 52 million tensor program performance records enables pre-trained cost models that accelerate autotuning convergence by 10 \times . However, creating such datasets requires substantial infrastructure investment.

5.1.3 Accuracy Stratification. We observe three accuracy tiers across the surveyed approaches:

Tier 1 (<5% error): Specialized models achieving near-perfect accuracy on narrow domains. nn-Meter achieves <1% error on edge device latency through kernel-level decomposition. NeuSight reaches 2.3% error on GPU inference through physics-informed tile-based prediction. LitePred achieves 0.7% error across 85 edge platforms through extensive transfer learning.

Tier 2 (5–15% error): General-purpose models with broader applicability. Habitat achieves 11.8% error on cross-GPU prediction using wave scaling. Analytical frameworks like Timeloop and MAESTRO typically achieve 5–15% error versus RTL simulation.

Tier 3 (15–25% error): Compiler cost models optimized for ranking rather than absolute accuracy. TVM’s AutoTVM [4] achieves approximately 20% MAPE, sufficient for guiding autotuning search. These models prioritize speed and online adaptation over absolute precision.

The key insight is that accuracy requirements depend on the use case: neural architecture search may tolerate 10–15% error if rankings are preserved, while hardware cost estimation for procurement decisions demands <5% accuracy.

5.2 Generalization Capabilities

Generalization—the ability to predict accurately on unseen workloads, configurations, or hardware—is perhaps the most critical capability for practical deployment. We analyze generalization along three axes: workload generalization, hardware generalization, and temporal generalization.

5.2.1 Workload Generalization. Models must handle neural network architectures not seen during training. GNN-based approaches offer natural workload generalization because the graph structure captures compositional relationships. GRANITE [13] generalizes across basic blocks by learning instruction-level patterns that compose into block-level predictions.

However, generalization often fails across workload *types*. Models trained on CNNs may not transfer to transformers due to fundamentally different computational patterns. NeuSight [11] addresses this by training on diverse operator types (GEMM, attention, convolution) and learning GPU execution semantics that generalize across operations.

5.2.2 Hardware Generalization. Cross-hardware prediction remains challenging due to microarchitectural diversity. Three approaches have shown promise:

Meta-learning treats hardware platforms as tasks. HELP [10] learns hardware embeddings that position devices in a shared latent space, enabling few-shot adaptation to new platforms.

Feature-based transfer uses hardware specifications as input features. LitePred [6] learns relationships between hardware characteristics (compute capability, memory bandwidth) and performance, enabling zero-shot prediction (92.1% accuracy) on entirely new devices.

Analytical decomposition factors predictions into hardware-dependent and hardware-independent components. Habitat [17] decomposes execution into compute and memory components that scale with

1161 known hardware parameters, achieving cross-GPU prediction without
 1162 retraining.

1163 *5.2.3 Temporal Generalization.* An underexplored dimension is
 1164 generalization across time—as software stacks evolve (new com-
 1165 piler versions, framework updates, driver changes), performance
 1166 characteristics shift. Models trained on older configurations may
 1167 degrade on current systems.

1168 Continual learning approaches that adapt to evolving hardware-
 1169 software stacks represent an important open direction. The TenSet
 1170 dataset’s versioned releases provide a starting point for studying
 1171 temporal generalization in compiler cost models.

1173 5.3 Interpretability

1175 Interpretability—understanding *why* a model makes particular predictions
 1176 is valuable for debugging, optimization guidance, and building prac-
 1177 titioner trust. We categorize approaches by their interpretability
 1178 characteristics.

1180 *5.3.1 Analytical Models: High Interpretability.* Analytical frame-
 1181 works like Timeloop [12] and MAESTRO [9] provide full inter-
 1182 pretability. Predictions decompose into explicit terms: data move-
 1183 ment at each memory level, compute utilization, bandwidth con-
 1184 straints. Practitioners can trace high-latency predictions to specific
 1185 bottlenecks (e.g., “DRAM bandwidth limits this mapping”).

1186 This interpretability enables *actionable insights*: if the model pre-
 1187 dictions memory-bound execution, the designer knows to explore map-
 1188 pings with better data reuse. The roofline model [14] exemplifies
 1189 this—identifying compute-bound versus memory-bound regimes
 1190 immediately suggests optimization directions.

1192 *5.3.2 Classical ML: Medium Interpretability.* Tree-based ensem-
 1193 bles provide feature importance rankings, indicating which input
 1194 features most influence predictions. nn-Meter’s kernel-level decom-
 1195 position enables interpretability: practitioners can identify which
 1196 kernels dominate latency and focus optimization efforts accord-
 1197 ingly.

1198 However, feature importance does not explain *how* features in-
 1199 teract. A model may indicate that “kernel size” is important without
 1200 revealing whether large or small kernels are faster for a given hard-
 1201 ware platform.

1202 *5.3.3 Deep Learning: Low Interpretability.* Deep neural networks,
 1203 including GNNs and transformers, function as black boxes. While
 1204 techniques like attention visualization and gradient-based attribu-
 1205 tion provide some insight, they rarely yield actionable optimization
 1206 guidance.

1207 NeuSight [11] partially addresses this through physics-informed
 1208 architecture: by decomposing predictions into compute and mem-
 1209 ory components that mirror GPU execution, the model structure
 1210 itself provides interpretability even though individual weight values
 1211 remain opaque.

1213 *5.3.4 Hybrid Approaches: Balanced Interpretability.* Hybrid analyt-
 1214 ical+ML models offer a middle ground. The analytical component
 1215 provides interpretable baselines, while the ML component captures
 1216 residual effects. When predictions diverge from analytical expec-
 1217 tations, practitioners know the difference stems from effects not

1219 captured in the analytical model (contention, cache effects, sched-
 1220 uling decisions).

1221 VIDUR [1] exemplifies this for LLM serving: discrete-event sim-
 1222 ulation provides interpretable system-level behavior, while learned
 1223 kernel-time predictors capture GPU execution details. The sim-
 1224 ulation structure enables “what-if” analysis (e.g., “how would P99
 1225 latency change with larger batch sizes?”) that pure ML models
 1226 cannot support.

1227 *5.3.5 The Interpretability-Accuracy Trade-off.* A general trade-off
 1228 exists between interpretability and accuracy. Analytical models
 1229 sacrifice accuracy for transparency; deep learning models sacri-
 1230 fice transparency for accuracy. For production deployment, hybrid
 1231 approaches that combine interpretable structure with learned com-
 1232 ponents increasingly represent the best of both worlds.

1233 6 Open Challenges and Future Directions

1235 Despite remarkable progress, significant challenges remain in ML-
 1236 based performance modeling. This section identifies key open prob-
 1237 lems and promising research directions that will shape the field’s
 1238 evolution.

1241 6.1 Data Availability and Quality

1243 The effectiveness of ML-based performance models fundamentally
 1244 depends on training data quality and availability. Several challenges
 1245 persist in this dimension.

1247 *6.1.1 Benchmark Diversity.* Existing datasets predominantly cover
 1248 CNN architectures optimized for image classification. TenSet [20]
 1249 provides 52 million tensor program records but focuses on oper-
 1250 ators from ResNet, MobileNet, and similar architectures. Modern
 1251 workloads—transformers, mixture-of-experts models, graph neural
 1252 networks, diffusion models—remain underrepresented.

1253 The rapid evolution of model architectures exacerbates this gap.
 1254 Models trained on 2022-era workloads may poorly predict perfor-
 1255 mance of 2025 architectures featuring sparse attention, conditional
 1256 computation, or novel activation functions. Continuously updated,
 1257 community-maintained benchmark suites could address this chal-
 1258 lenge.

1260 *6.1.2 Hardware Coverage.* Hardware diversity creates data collec-
 1261 tion bottlenecks. LitePred [6] covers 85 edge devices, but the mobile
 1262 hardware landscape spans hundreds of distinct SoC configura-
 1263 tions. Data center hardware (H100, TPU v5, custom accelerators) often
 1264 has restricted access, limiting public dataset creation.

1265 Simulation-based data generation offers a partial solution: Arch-
 1266 Gym [8] trains on Timeloop outputs, avoiding hardware access
 1267 requirements. However, simulation accuracy itself requires valida-
 1268 tion against real hardware, creating a chicken-and-egg problem.

1270 *6.1.3 Measurement Noise and Reproducibility.* Performance mea-
 1271 surements exhibit variance from thermal throttling, OS scheduling,
 1272 memory allocation, and caching effects. Industrial-strength profil-
 1273 ing requires careful warm-up periods, multiple runs, and statistical
 1274 aggregation. Many published models train on single-run measure-
 1275 ments, potentially learning noise rather than signal.

1277 Standardized measurement protocols—specifying warm-up it-
 1278 erations, cooling periods, statistical aggregation methods—would
 1279 improve cross-study comparability and model reliability.

1280 6.2 Model Generalization

1282 Generalization remains the central challenge: models that excel on
 1283 training distributions often fail on realistic deployment scenarios.

1285 *6.2.1 Cross-Workload Generalization.* Models struggle to general-
 1286 ize across workload types. A predictor trained on CNNs may fail
 1287 on transformers due to different computational patterns: CNNs are
 1288 compute-dominated by convolutions with high data reuse, while
 1289 transformers feature attention mechanisms with sequence-length-
 1290 dependent memory access patterns.

1291 Promising directions include workload-agnostic representations
 1292 (learning from computation graphs rather than architecture-specific
 1293 features) and multi-task learning across workload families.

1294 *6.2.2 Cross-Hardware Generalization.* Hardware generalization
 1295 faces fundamental obstacles. Different hardware families (CPUs,
 1296 GPUs, TPUs, FPGAs) employ distinct execution models, memory
 1297 hierarchies, and parallelism patterns. Even within GPU families,
 1298 architectural changes (Volta to Ampere to Hopper) introduce new
 1299 features (tensor cores, TMA, FP8) that alter performance character-
 1300 istics.

1301 Transfer learning approaches [6, 10] show promise for related
 1302 hardware, but truly cross-family prediction (e.g., GPU to TPU)
 1303 remains elusive. Hardware-agnostic intermediate representations
 1304 that capture essential computational patterns while abstracting
 1305 platform details could enable broader transfer.

1307 *6.2.3 Distribution Shift.* Performance models face distribution shift
 1308 as software stacks evolve. Compiler optimizations, framework up-
 1309 dates, and driver changes alter the workload-to-hardware mapping,
 1310 invalidating models trained on older configurations.

1311 Online adaptation and continual learning techniques could ad-
 1312 dress distribution shift, but few studies systematically evaluate tem-
 1313 poral generalization. Developing benchmarks that explicitly mea-
 1314 sure robustness to software evolution would accelerate progress.

1315 6.3 Integration with Design Flows

1317 For ML-based performance models to impact practice, they must
 1318 integrate seamlessly with existing design and optimization work-
 1319 flows.

1320 *6.3.1 Compiler Integration.* Compiler autotuning represents a nat-
 1321 ural application: ML models guide the search for optimal tensor
 1322 program configurations. TVM [4] and Ansor [19] demonstrate this
 1323 integration, but challenges remain.

1325 Cost model accuracy directly affects autotuning efficiency. Mis-
 1326 predictions cause the search to explore suboptimal regions, wast-
 1327 ing compilation time. Uncertainty quantification—knowing when
 1328 predictions are unreliable—could enable more efficient exploration-
 1329 exploitation trade-offs. Recent uncertainty-aware cost models can
 1330 provide calibrated uncertainty estimates, but such techniques are
 1331 not yet standard.

1332 *6.3.2 Architecture Exploration.* Hardware design space exploration
 1333 requires evaluating millions of configurations. ML surrogate models

1335 can accelerate this process, as demonstrated by ArchGym [8], but
 1336 integration challenges persist.

1337 The design space is often too large for exhaustive surrogate train-
 1338 ing. Active learning strategies that intelligently select which config-
 1339 urations to simulate could improve sample efficiency. Additionally,
 1340 surrogate models must provide reliable uncertainty estimates to
 1341 avoid overconfident predictions that mislead designers.

1342 *6.3.3 Serving System Optimization.* LLM serving systems require
 1343 real-time performance prediction for scheduling decisions. VIDUR [1]
 1344 provides offline simulation, but online serving requires predictions
 1345 within microseconds.

1346 Lightweight models suitable for real-time inference, combined
 1347 with periodic retraining on observed performance, could enable
 1348 adaptive serving optimization. The challenge is maintaining accu-
 1349 racy while meeting strict latency requirements.

1350 6.4 Emerging Opportunities

1353 Several emerging trends create new opportunities for ML-based
 1354 performance modeling.

1355 *6.4.1 Foundation Models for Performance Prediction.* The success
 1356 of foundation models in NLP and vision suggests potential for per-
 1357 formance modeling. A foundation model pre-trained on diverse
 1358 workload-hardware-performance tuples could provide strong ini-
 1359 tialization for downstream tasks.

1360 Key requirements include: (1) massive, diverse training datasets
 1361 spanning hardware platforms and workload types; (2) representa-
 1362 tion learning that captures transferable performance patterns; and
 1363 (3) efficient adaptation mechanisms for new domains. The TenSet
 1364 dataset [20] provides a starting point, but orders-of-magnitude
 1365 more data may be needed for true foundation model capabilities.

1366 *6.4.2 LLM-Assisted Performance Analysis.* Large language models
 1367 offer new modalities for performance understanding. Recent work
 1368 explores using code LLMs to extract performance-relevant features,
 1369 explain performance anomalies, and suggest optimizations.

1370 Challenges include hallucination (LLMs may generate plausible-
 1371 sounding but incorrect performance estimates) and limited nu-
 1372 matical reasoning. Hybrid approaches combining LLM-based code
 1373 understanding with principled performance models could leverage
 1374 the strengths of both paradigms.

1375 *6.4.3 Hardware-Software Co-Design.* ML performance models can
 1376 enable tighter hardware-software co-design by rapidly evaluating
 1377 how software changes affect hardware utilization and vice versa.

1378 ArchGym [8] demonstrates ML-guided accelerator design, but
 1379 joint optimization of hardware architecture, compiler mappings,
 1380 and model structure remains underexplored. Differentiable perfor-
 1381 mance models could enable gradient-based co-optimization across
 1382 the full stack.

1383 *6.4.4 Emerging Hardware Paradigms.* New computing paradigms—
 1384 processing-in-memory, neuromorphic computing, analog accelerators,
 1385 quantum-classical hybrids—require new performance model-
 1386 ing approaches. These platforms exhibit fundamentally different
 1387 performance characteristics (energy-dominated costs, stochastic
 1388 execution, analog noise) that existing frameworks do not address.

Table 4: Summary of reproducibility evaluation for representative performance modeling tools. Scores reflect practical reproducibility on a 10-point scale.

Tool	Target	Score	Key Issue
Timeloop	Accelerators	9/10	Docker recommended
FlashAttention	GPUs	9/10	GPU required
ASTRA-sim	Distributed	8/10	Complex build
VIDUR	LLM serving	7/10	Python 3.10 only
nn-Meter	Edge devices	5/10	Pickle incompatibility

Early-stage performance modeling for emerging hardware could accelerate adoption by enabling software optimization before hardware availability. Transfer learning from related platforms (e.g., digital accelerators to analog) represents a promising direction.

6.4.5 Multi-Objective and Constraint-Aware Prediction. Practical deployment involves multiple objectives: latency, throughput, energy, memory, cost. Most current models predict single metrics independently. Joint multi-objective prediction could enable Pareto-optimal design selection.

Additionally, constraint-aware prediction—determining whether a workload *fits* on target hardware given memory limits—is often more valuable than precise latency estimates. Models that directly predict feasibility and constraint violations could better support deployment decisions.

7 Experimental Evaluation

To validate the practical applicability of surveyed performance modeling tools, we conducted hands-on reproducibility evaluations of five representative systems spanning different hardware targets and modeling approaches. This section presents our methodology, tool-by-tool findings, and synthesizes key lessons for practitioners.

7.1 Evaluation Methodology

We evaluated each tool along four dimensions:

Setup Complexity. We assessed installation difficulty, dependency management, documentation quality, and time to first result. Tools were tested in clean environments following their documented procedures.

Reproducibility. We verified whether example configurations produce consistent results and whether reference outputs are provided for validation.

Practical Usability. We examined API design, configuration flexibility, output interpretability, and integration with existing workflows.

Accuracy Validation. Where possible, we compared tool outputs against published accuracy claims or reference implementations.

Table 4 summarizes our findings across all evaluated tools.

7.2 Tool-by-Tool Results

7.2.1 Timeloop: DNN Accelerator Modeling. Timeloop [12] provides analytical performance and energy modeling for DNN accelerators through loop-nest analysis.

Setup. Docker-based installation succeeds in 10–15 minutes with pre-built images for both x86 and ARM platforms. Native installation requires 1–2 hours due to complex dependencies (Barvinok, NTL libraries).

Reproducibility. Excellent—reference outputs are provided for all example architectures (Eyeriss, Simba), and results are deterministic. Tutorials with Jupyter notebooks enable systematic learning.

Key Finding. Energy breakdown analysis reveals DRAM dominates (>60%) for typical configurations, validating the importance of dataflow optimization. The mapper may not find globally optimal solutions but provides interpretable trade-off analysis.

7.2.2 ASTRA-sim: Distributed Training Simulation. ASTRA-sim [15] simulates distributed DNN training with configurable network backends.

Setup. Docker recommended due to Protobuf version sensitivity. Native build requires 1–2 hours with careful dependency management.

Reproducibility. Good—validated configurations for HGX-H100 and DGX-V100 are included. However, reference timing outputs are not provided, requiring trust in published accuracy claims.

Key Finding. The Chakra trace format has a learning curve, but enables detailed collective communication modeling. Multiple network backends (analytical, NS-3) allow accuracy-speed trade-offs.

7.2.3 VIDUR: LLM Inference Simulation. VIDUR [1] provides discrete-event simulation for LLM serving systems.

Setup. Python-only installation, but strict Python 3.10 requirement creates compatibility issues—Python 3.14 fails due to argparse API changes.

Reproducibility. Good for supported configurations. Pre-profiled data covers A100, H100, and A40 GPUs for Llama-family models. Adding new models requires GPU access for profiling.

Key Finding. Rich scheduler implementations (vLLM, Orca, Sarathi) enable direct algorithm comparison. Metrics include time-to-first-token, time-per-output-token, and memory utilization—essential for SLO-driven capacity planning.

7.2.4 nn-Meter: Edge Device Latency Prediction. nn-Meter [18] predicts DNN latency on edge devices through kernel-level decomposition.

Setup. Simple pip installation, but critical compatibility issue: pre-trained predictors fail to load with current scikit-learn versions due to pickle format changes.

Reproducibility. Poor in current state—the core functionality (pre-trained predictors) is broken without pinning scikit-learn to version 1.0.x.

Key Finding. This case highlights a critical reproducibility anti-pattern: ML models serialized with pickle are fragile across library versions. Researchers should prefer version-agnostic serialization formats (ONNX, SavedModel) or pin exact dependency versions.

7.2.5 FlashAttention: GPU Attention Optimization. FlashAttention provides IO-aware attention kernels achieving 2–4× speedup over standard implementations.

Setup. PyPI package available; compilation from source requires 3–5 minutes with ninja. GPU (Ampere or newer) required for both building and running.

1509 **Table 5: Reproducibility best practices derived from tool eval-
1510 uation.**

1512 Practice	1513 Rationale
1514 Provide Docker images	1515 Isolates dependencies
1516 Document Python version	1517 Prevents API incompatibilities
1518 Include reference outputs	1519 Enables result verification
1520 Use portable model formats	1521 Avoids pickle versioning issues
1522 Pin dependency versions	1523 Ensures reproducible environments

1521 **Reproducibility.** Excellent—widely adopted in major frame-
1522 works (HuggingFace, vLLM), comprehensive test suite, and bench-
1523 mark scripts for validation.

1524 **Key Finding.** FlashAttention demonstrates successful repro-
1525 ductibility through framework integration rather than standalone
1526 distribution. Native integration into PyTorch’s scaled_dot_product_attention
1527 ensures continued maintenance and compatibility.

1528 7.3 Synthesis and Recommendations

1529 Our evaluation reveals systematic patterns affecting reproducibility:

1530 Containerization dramatically improves reproducibility.

1531 Tools providing Docker images (Timeloop, ASTRA-sim) achieve
1532 higher reproducibility scores by isolating complex dependency
1533 chains. Native builds consistently encounter platform-specific is-
1534 sues.

1535 **Python version sensitivity is a major concern.** VIDUR re-
1536 quires Python 3.10 specifically; nn-Meter’s pickle files are incom-
1537 patible with current scikit-learn. Projects should document version
1538 constraints prominently and consider providing locked dependency
1539 specifications.

1540 **Pre-trained models age poorly.** nn-Meter’s reliance on pickled
1541 scikit-learn models created a time bomb. FlashAttention avoids this
1542 by focusing on kernel optimization rather than learned models.
1543 For projects distributing trained models, ONNX or similar portable
1544 formats are preferable.

1545 **Reference outputs enable validation.** Timeloop’s inclusion
1546 of expected outputs for all examples enables immediate verification.
1547 ASTRA-sim and VIDUR lack this, requiring users to trust published
1548 accuracy claims.

1549 Table 5 summarizes best practices derived from our evaluation.

1552 8 Conclusion

1553 This survey has provided a comprehensive analysis of machine
1554 learning approaches for computer architecture performance mod-
1555 eling. We have examined over 60 papers spanning traditional ana-
1556 lytical models, simulation-based approaches, and modern ML tech-
1557 niques including classical machine learning, deep learning, graph
1558 neural networks, and hybrid methods.

1561 8.1 Key Findings

1562 Our analysis reveals several key findings that characterize the cur-
1563 rent state of the field:

1564 **ML-based models achieve remarkable accuracy.** State-of-
1565 the-art approaches achieve prediction errors below 5% for their

1566 target domains. NeuSight [11] reaches 2.3% error on GPU infer-
1567 ence through physics-informed tile-based prediction. LitePred [6]
1568 achieves 0.7% error across 85 edge platforms through transfer learn-
1569 ing. These accuracy levels are sufficient for production deployment
1570 in neural architecture search, autotuning, and hardware-aware op-
1571 timization.

1572 **Hybrid approaches dominate recent work.** The most suc-
1573 cessful models combine analytical structure with learned compo-
1574 nents. Analytical decomposition provides interpretable baselines
1575 and physics-based inductive bias, while ML captures complex ef-
1576 fects that elude closed-form analysis. This hybrid philosophy—
1577 exemplified by NeuSight’s tile-based prediction and VIDUR’s [1]
1578 simulation-based framework—consistently outperforms pure ana-
1579 lytical or pure ML approaches.

1580 **Transfer learning is essential for scalability.** The prolifera-
1581 tion of hardware platforms makes per-device training impractical.
1582 Meta-learning (HELP [10]) and VAE-based sampling (LitePred [6])
1583 enable adaptation to new devices with 10–100 samples, demon-
1584 strating that cross-platform generalization is tractable.

1585 **Kernel-level decomposition improves accuracy.** nn-Meter’s [18]
1586 insight that end-to-end latency decomposes into kernel latencies
1587 has become standard practice. By modeling at the kernel level and
1588 capturing framework fusion behavior, models achieve composi-
1589 tional predictions that generalize across architectures.

1590 **LLM inference presents unique challenges.** Large language
1591 model serving has distinct characteristics—autoregressive gener-
1592 ation, KV cache growth, prefill-decode phase separation—that re-
1593 quire specialized modeling. VIDUR [1] and similar frameworks
1594 provide discrete-event simulation capturing these dynamics with
1595 <5% accuracy.

1599 8.2 Promising Research Directions

1600 Looking forward, we identify the most promising directions for
1601 advancing the field:

1602 **Foundation models for performance prediction.** Pre-trained
1603 models that transfer across workloads and hardware could dramat-
1604 ically reduce data requirements for new prediction tasks. Creating
1605 the large-scale, diverse datasets needed to train such models rep-
1606 presents a key community challenge.

1607 **Uncertainty quantification.** Knowing when predictions are
1608 reliable enables better decision-making in autotuning, design space
1609 exploration, and serving optimization. Calibrated uncertainty es-
1610 timates remain underexplored despite their practical importance.

1611 **Temporal generalization.** As software stacks evolve, perfor-
1612 mance models must adapt. Continual learning approaches and
1613 benchmarks measuring robustness to software evolution deserve
1614 increased attention.

1615 **Multi-objective prediction.** Practical deployment involves la-
1616 tency, throughput, energy, memory, and cost trade-offs. Joint multi-
1617 objective prediction could enable Pareto-optimal design selection
1618 across these dimensions.

1619 **Emerging hardware support.** Processing-in-memory, neuro-
1620 morphic computing, and analog accelerators require new modeling
1621 paradigms. Early-stage performance modeling for emerging hard-
1622 ware could accelerate adoption.

8.3 Concluding Remarks

Machine learning has transformed performance modeling from an art requiring deep architectural intuition to an increasingly systematic discipline. The surveyed approaches demonstrate that learned models can capture complex hardware-software interactions while enabling millisecond-scale prediction. As ML workloads continue to grow in importance and hardware diversity expands, accurate, generalizable performance models will become ever more critical for efficient system design and deployment.

We hope this survey serves as both a comprehensive reference for practitioners selecting performance modeling approaches and a roadmap for researchers identifying impactful open problems. The field's rapid progress suggests that the coming years will bring continued advances in accuracy, generalization, and practical deployment of ML-based performance models.

References

- [1] Amey Agrawal, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S. Gulavani, and Ramachandran Ramachandran. 2024. VIDUR: A Large-Scale Simulation Framework for LLM Inference. In *Proceedings of Machine Learning and Systems (MLSys)*. 1–15.
- [2] Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt. 2009. Analyzing CUDA Workloads Using a Detailed GPU Simulator. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 163–174. <https://doi.org/10.1109/ISPASS.2009.4919648>
- [3] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The gem5 Simulator. *ACM SIGARCH Computer Architecture News* 39, 2 (2011), 1–7. <https://doi.org/10.1145/2024716.2024718>
- [4] 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. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 578–594.
- [5] Lukasz Dudziak, Thomas Chau, Mohamed S. Abdelfattah, Royson Lee, Hyeji Kim, and Nicholas D. Lane. 2024. Latency Predictors for Neural Architecture Search. In *Proceedings of Machine Learning and Systems (MLSys)*. 1–14.
- [6] Yang Feng, Zhehao Li, Jiacheng Yang, and Yunxin Liu. 2024. LitePred: Transferable and Scalable Latency Prediction for Hardware-Aware Neural Architecture Search. In *Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 1–18.
- [7] Mahmoud Khairy, Zhesheng Shen, Tor M. Aamodt, and Timothy G. Rogers. 2020. Accel-Sim: An Extensible Simulation Framework for Validated GPU Modeling. In *Proceedings of the 47th International Symposium on Computer Architecture (ISCA)*. 473–486. <https://doi.org/10.1109/ISCA45697.2020.00047>
- [8] Srivatsan Krishnan, Amir Yazdanbakhsh, Shvetank Prakash, Norman P. Jouppi, Jignesh Parmar, Hyoukjun Kim, James Laudon, and Chandrakant Narayanaswami. 2023. ArchGym: An Open-Source Gymnasium for Machine Learning Assisted Architecture Design. In *Proceedings of the 50th International Symposium on Computer Architecture (ISCA)*. 1–16. <https://doi.org/10.1145/3579371.3589049>
- [9] Hyoukjun Kwon, Prasanth Chatarasi, Michael Sarber, Michael Pellauer, Angshuman Parashar, and Tushar Krishna. 2019. MAESTRO: A Data-Centric Approach to Understand Reuse, Performance, and Hardware Cost of DNN Mappings. In *Proceedings of the 52nd IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–14. <https://doi.org/10.1145/3352460.3358292>
- [10] Hayeon Lee, Sewooong Lee, Song Chong, and Sung Ju Hwang. 2021. HELP: Hardware-Adaptive Efficient Latency Prediction for NAS via Meta-Learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, Vol. 34. 27016–27028.
- [11] Seunghyun Lee, Amar Phanishayee, and Divya Mahajan. 2025. NeuSight: GPU Performance Forecasting via Tile-Based Execution Analysis. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 1–15.
- [12] Angshuman Parashar, Priyanka Raina, Yakun Sophia Shao, Yu-Hsin Chen, Victor A. Ying, Anurag Muber, Rangharajan Venkatesan, Brucek Khailany, Stephen W. Keckler, and Joel Emer. 2019. Timeloop: A Systematic Approach to DNN Accelerator Evaluation. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 304–315.
- [13] Ondrej Sykora, Alexis Rucker, Charith Mendis, Rajkishore Barik, Phitchaya Mangpo Phothilimthana, and Saman Amarasinghe. 2022. GRANITE: A Graph Neural Network Model for Basic Block Throughput Estimation. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*. 1–13. <https://doi.org/10.1109/IISWC55918.2022.00014>
- [14] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Commun. ACM* 52, 4 (2009), 65–76. <https://doi.org/10.1145/1498765.1498785>
- [15] William Won, Taekyung Heo, Saeed Rashidi, Saeed Talati, Srinivas Srinivasan, and Tushar Krishna. 2023. ASTRA-sm2.0: Modeling Hierarchical Networks and Disaggregated Systems for Large-Model Training at Scale. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 283–294. <https://doi.org/10.1109/ISPASS57527.2023.00035>
- [16] Yannan Nelli Wu, Joel Emer, and Vivienne Sze. 2022. SparseLoop: An Analytical Approach to Sparse Tensor Accelerator Modeling. In *Proceedings of the 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–15. <https://doi.org/10.1109/MICRO56248.2022.00078>
- [17] Geoffrey X. Yu, Yubo Gao, Pavel Golber, and Asaf Cidon. 2021. Habitat: A Runtime-Based Computational Performance Predictor for Deep Neural Network Training. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 503–521.
- [18] Li Lyra Zhang, Shihao Han, Jianyu Wei, Ningxin Zheng, Ting Cao, Yuqing Yang, and Yunxin Liu. 2021. nn-Meter: Towards Accurate Latency Prediction of Deep-Learning Model Inference on Diverse Edge Devices. In *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*. 81–93. <https://doi.org/10.1145/3458864.3467882> Best Paper Award.
- [19] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Amer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. 2020. Ansor: Generating High-Performance Tensor Programs for Deep Learning. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 863–879.
- [20] Lianmin Zheng, Ruochen Liu, Junru Shao, Tianqi Chen, Joseph E. Gonzalez, Ion Stoica, and Zhihao Zhang. 2021. TenSet: A Large-scale Program Performance Dataset for Learned Tensor Compilers. In *Advances in Neural Information Processing Systems (NeurIPS)*, Vol. 34. 29876–29888.