

A Comparison of Performance Prediction Methods for GPU-Based Deep Learning

Begüm Balat

Technical University of Munich

School of Computation, Information and Technology

Munich, Germany

begum.balat@tum.de

Abstract—Training and deploying deep neural networks is computationally expensive, and the choice of GPU hardware strongly influences both runtime and cost. Estimating how long a deep learning workload will run on a particular GPU is difficult without access to that GPU, because performance is determined by low-level interactions between the workload, the kernel libraries, and the hardware. This paper compares three families of performance prediction methods, namely analytical models, profiling-based models, and machine-learning-based predictors, and concludes that the most accurate current approaches are hybrids that combine learned components with explicit hardware knowledge. The comparison is carried out by examining popular representative systems per family, describing their prediction mechanisms and required inputs, and contrasting their reported accuracy, generality across GPUs and workloads, and setup cost. Across the examined systems, prediction errors range from roughly two percent to well above one hundred percent depending on how far the target hardware and model lie outside the assumptions or training data of the predictor. The resulting comparison offers guidance on which class of method is appropriate for a given use case, such as hardware selection, scalability estimation, or optimization planning.

Index Terms—GPU, deep learning, performance prediction, performance modelling, roofline model, profiling, machine learning

I. INTRODUCTION

Deep learning has become one of the dominant workloads in high-performance computing. Model sizes have grown from millions to hundreds of billions of parameters within a decade, and training such models consumes thousands of GPU-hours of compute time, with corresponding monetary and energy costs. At the same time, the GPUs themselves have become a scarce resource: the most recent generations are expensive to purchase, and their availability is limited, with lead times of up to a year reported for flagship data-center GPUs [8].

In this setting, practitioners repeatedly face questions whose answers depend on knowing the runtime of a workload *before* running it. Which of several candidate GPUs should be rented in the cloud for a given training job? Is an upgrade from an existing GPU to a more expensive one worth the price? How will the training time change when the model is scaled up or distributed across more devices? Which software optimization should be implemented first? Answering these questions by direct measurement requires access to every hardware configuration under consideration, which is exactly what is expensive or impossible to obtain. Performance prediction addresses this

problem: it estimates the execution time of a deep learning workload on a target GPU without executing the workload on that GPU.

Such prediction is difficult because the runtime of a deep learning workload emerges at the level of individual GPU kernels. It depends on which kernel implementation a library such as cuDNN selects for a given operation and architecture, on how the kernel’s work is tiled and scheduled across the streaming multiprocessors, on the behavior of the memory hierarchy, and on how far the achieved throughput falls short of the theoretical peak. Hardware specification sheets alone are therefore a poor basis for estimation; in practice, deep learning workloads rarely exhaust the computational resources even of high-end GPUs [10].

Existing prediction approaches can be grouped into three families that differ in where their knowledge of the system originates. *Analytical models* derive execution time mathematically from a description of the workload and the hardware capabilities. *Profiling-based models* measure the workload, or a part of it, on available hardware and extrapolate from the measurements. *Machine-learning-based predictors* learn the mapping from workload and hardware features to runtime from collected data. These families form a spectrum that ranges from fully explicit modelling to fully learned behaviour.

This paper compares the three families through representative systems: the Hong and Kim model [4] and Paleo [5] for analytical models, Habitat [6] and Daydream [7] for profiling-based models, and NeuSight [8] for machine-learning-based predictors. For each system, the underlying mechanism, the required inputs, the reported accuracy, and the limitations are described. The methods are then compared across accuracy, generality, and setup cost. A recurring observation of this comparison is that no competitive method is pure: analytical models require measured calibration constants, profiling-based models embed learned components, and the most accurate learned predictor constrains its outputs with analytical performance laws. The three paradigms are therefore converging toward hybrid designs.

These five systems were chosen because each is a widely cited representative of its paradigm, and because they are historically connected rather than arbitrary: Paleo extends Hong and Kim’s kernel-level model to the scale of whole DNN training, Habitat and Daydream apply the same profiling-based

approach to two different predictive questions, and NeuSight explicitly benchmarks itself against both Habitat and Li et al.’s method, allowing this comparison to draw on evaluations the field has already validated. Spanning 2009 to 2025, the five systems also trace the evolution of each paradigm over roughly the last two decades.

The remainder of this paper is structured as follows. Section II introduces the necessary background on deep neural network training, GPU kernels, the roofline model, and the CUPTI profiling interface. Sections III, IV, and V present the analytical, profiling-based, and machine-learning-based methods, respectively. Section VI compares the methods across paradigms, and Section VII concludes with an outlook.

II. BACKGROUND

A. Training iterations in deep neural networks

A deep neural network (DNN) consists of layers of parameters, called weights, that are adjusted through training to minimize the loss function [2]. Training is done in iterations on a batch of input data. In the initial forward pass, the input is propagated through the layers of the network to produce a prediction. The loss value is then calculated by comparing the expected output with the generated prediction. Secondly, in the backward pass, the chain rule is applied to compute the gradient of loss with respect to every weight in the network, working backward from the output layer to the input layer. Lastly, an optimizer uses these gradients to update each weight. The three steps are repeated for many batches during the course of training.

B. GPU Kernels

Each operation that occurs during the training iterations is executed on the GPU as a separate kernel. A kernel is a function executed in parallel by many threads inside the GPU, each of which operates on a separate portion of the data [3]. A single training iteration of a modern DNN therefore launches a long sequence of kernels, one for each operation in the forward and backward pass, as well as for the weight update step. Since GPUs are designed to execute many identical operations on different data simultaneously, this kernel-based execution model maps well onto the parallel structure of DNN computations over the more commonly used CPUs.

C. The Roofline model and arithmetic intensity

The performance of a GPU kernel is constrained by two hardware limits: the peak computational throughput and the peak memory bandwidth. Peak computational throughput (P_{peak}) is measured in FLOPs/sec, and the peak memory bandwidth (BW_{peak}) in bytes/sec. To determine which constraint is the main limiting factor, we first calculate the operational intensity (OI), also known as arithmetic intensity, defined as the number of floating-point operations performed per byte of data transferred. A low OI means the kernel moves a lot of data relative to the computation done on it, making it memory-bound; a high OI means it performs more calculations per byte moved, making it compute-bound. The value separating these

two regimes, $OI = P_{\text{peak}}/BW_{\text{peak}}$, is called the ridge point: kernels with intensity below it are memory-bound and those above it are compute-bound.

These two regimes together determine the attainable performance P of a kernel, expressed by the roofline model [1] as

$$P = \min(P_{\text{peak}}, BW_{\text{peak}} \times OI). \quad (1)$$

D. CUPTI

The CUDA Profiling Tools Interface (CUPTI) is a set of APIs shipped with the CUDA toolkit that enables the construction of profiling and tracing tools for CUDA applications [11]. Through its activity interface, CUPTI records every kernel launch and memory transfer together with timestamps, durations, and stream identifiers, which allows a complete kernel-level timeline of an application to be reconstructed. Through its event and metric interfaces, CUPTI exposes hardware performance counters, from which quantities such as the number of executed floating-point operations and the number of bytes transferred from device memory can be obtained. Combining these two counters yields the measured operational intensity of a kernel, so the roofline classification of Section II-C can be determined for a running kernel rather than estimated from source code. CUPTI underlies NVIDIA’s own profilers and serves as the measurement layer for several of the performance prediction systems examined in this work.

E. Motivation for performance prediction

Training large DNNs is computationally expensive, and the cost depends heavily on the choice of GPU hardware, with different GPU generations offering very different trade-offs between compute throughput, memory bandwidth, and price. Selecting a suitable GPU for a given training workload is therefore a non-trivial decision, particularly as the number of available GPU types continues to grow. Currently, before selecting a GPU, two factors are considered: measuring the throughput of the GPU and consulting existing GPU performance benchmarks. However, the throughput cannot be measured without acquiring access to the GPU, and existing benchmarks are only available for a subset of GPUs. Even if financial motivations are overlooked and a GPU is selected solely on heuristics, assuming the most expensive GPU would work the best, benchmarking studies show that DNN training workloads often do not exhaust the computational resources of the GPU [10]. Being able to predict the training iteration time of a DNN on a GPU that has not yet been used for that workload allows practitioners to compare hardware options without incurring the cost of running the workload on each candidate GPU.

III. ANALYTICAL MODELS

Analytical models aim to estimate the performance of GPU workloads from a mathematical approach, evaluating the computational workload and the hardware capabilities without executing the model. The computational workload, derived from the network itself, is the sum of the calculations and

data movement within a program, measured through FLOP counts, layer counts, instruction counts, and memory accesses. Hardware capabilities are specified by parameters of the GPU, such as peak FLOPS, memory bandwidth, and communication bandwidth. Early analytical models targeted the execution of individual GPU kernels, whereas with the growing use of DNNs, later models specifically target the forward and backward propagation of network training.

Hong and Kim. Previously, Hong and Kim suggested an analytical model that can calculate the kernel execution time of a GPU before execution, by calculating memory and compute costs on parallelized kernels in cycles per instruction (CPI) [4]. The model is built on two metrics: Memory Warp Parallelism (MWP) and Computation Warp Parallelism (CWP). A warp is a group of threads that the GPU executes together, issuing the same instruction across all of them simultaneously. MWP is an estimate of how many memory requests can be executed concurrently, calculated using the number of concurrently running threads and the memory bandwidth consumption. CWP, in contrast, represents how much computation can be done by other warps while one warp is waiting for a memory fetch. The model compares the two by considering hardware parameters such as memory bandwidth, memory latency, number of cores, and clock frequency, as well as kernel parameters retrieved from the program such as the number of warps, the count of computation versus memory instructions per thread, and the memory access pattern. By comparing MWP and CWP, the model concludes whether the kernel is memory- or compute-bound.

While the model was accurate, with a geometric mean absolute error of 5.4% on micro-benchmarks (simple, isolated kernels), the error rate increases to 13.3% on the Merge benchmarks (more complex GPU applications) [4], [12]. The model also assumes no hardware caches for global memory, which is unrealistic on modern cached GPUs. These kernel-level and hardware-specific limits motivated higher-level performance models built for modern hardware and deep-learning workloads, such as Paleo.

Paleo. Paleo is designed in part to answer the question of how to create a neural network architecture that can be efficiently trained and evaluated under common hardware setups [5]. Paleo decomposes execution time into computation and communication time. The architecture declaratively tells how much computation forward and backward propagation will require, hence the operations can be counted from the structure alone. Paleo can therefore predict the execution time of training a DNN before execution, and because it can estimate time across different setups (e.g., different GPU counts), it is also used for predicting scalability. For each layer of the DNN, the required FLOPs and memory are counted and then divided by the hardware capabilities (peak FLOPS, bandwidth) to get an estimated execution time. This calculation mirrors the roofline principle introduced in Section II-C.

Paleo takes sequential layers as the sum of each layer’s time. For parallel branches, the synchronization barriers before and after the parallel structure are grouped into one supernode

(U). The supernode’s execution time $T(U)$ is calculated as a range over the parallel computation sequence, assuming perfect parallelization as the lower bound and fully sequential execution as the upper bound. When training across multiple GPUs, the gradients must be synchronized; to model this communication cost, Paleo uses AllReduce schemes (OneToAll, Tree, and Butterfly), each with their own cost, and calculates communication time by dividing the communicated data size by the bandwidth of the communication channel. Finally, two implementations for convolutional operations are considered — matrix multiplication and Fast Fourier Transform — and since these have different FLOP counts, Paleo uses cuDNN’s heuristics to predict which method the real system would favor. This is necessary because Paleo must also model software behavior, not just hardware.

Paleo does not report aggregate error metrics; instead it provides tables comparing predicted and measured times. The layer-wise comparison shows that Paleo’s per-layer estimates closely track measured TensorFlow times across both AlexNet and VGG-16, with the fc6 layer being the notable exception, where Paleo substantially underestimates runtime because the real system spends much of that layer’s time on data-layout transformations that the analytical model does not capture [5]. One point to stress is that Paleo assumes peak hardware conditions; to close this gap, the authors run a small benchmark on a real GPU and fit a Platform Percent of Peak (PPP) correction factor that captures how far real hardware falls short of its theoretical peak. This means that even an architecture-aware analytical model cannot remain purely analytical, as it still depends on a measured constant. The prediction error also varies with scale, from as low as roughly 1.3% at two workers up to roughly 14% at eight workers in the AlexNet hybrid-parallelism case (derived from [5], Tab. 4), with no consistent monotonic trend across intermediate worker counts. Finally, the model considers only certain communication schemes, excluding butterfly mixing and asynchronous parameter servers.

Limitations. Analytical models aim to predict performance without running the code, making them fast and cheap. Nonetheless, they rely on assumptions that limit accuracy on real systems. If one can afford to run the workload, even partially, profiling-based methods become attractive by trading higher up-front cost for greater fidelity. The need for empirical calibration in even the most architecture-aware analytical models motivates approaches that measure execution directly, as discussed in the following section.

IV. PROFILING-BASED MODELS

In the previous section, analytical models predicted GPU performance for DNN training from a description of the workload and the hardware, without executing the network itself. Even so, the analytical model of Paleo required a measured constant, PPP, to account for how far real hardware falls short of its theoretical peak. Profiling-based models, on the other hand, rely directly on executed measurements: rather than estimating performance from FLOP counts and hardware specifications, they execute the workload, or part of it, on a

real GPU and use the collected timing data as the basis for prediction. The advantage is fidelity, as a real measurement captures implementation-level behavior, such as which kernels a library actually selects, which an analytical model can only approximate through assumptions like peak hardware conditions or a fitted correction factor. This improvement comes at a cost: profiling-based methods require access to a GPU that can run the workload at least once, whereas an analytical model needs at most a small calibration benchmark.

Habitat. Habitat is a profiling-based model that utilizes both the GPU the DNN developer already has (the origin GPU) and the hardware specifications of a target GPU, to predict the training iteration time of the DNN on a GPU the developer does not have access to (the destination GPU) [6]. This ability resolves the hardware-access aspect of the GPU-selection problem, as candidate GPUs can be compared without first obtaining them. As discussed under training in Section II, the authors of Habitat observed that a DNN training session consists of repeated identical iterations, so its performance can be estimated by measuring a single iteration on the origin GPU and scaling that measurement to the destination GPU. This scaling is carried out by two mechanisms: wave scaling and pre-trained multilayer perceptrons (MLPs).

Wave Scaling. A GPU contains multiple processors called Streaming Multiprocessors (SMs). A kernel’s work is divided into thread blocks, which are assigned to SMs for execution. Because the number of SMs is limited, the thread blocks are processed in successive batches called waves. The wave size is the number of thread blocks that run simultaneously, and the number of waves is the total number of thread blocks divided by the wave size. If a time per wave can be assigned, then multiplying it by the number of waves estimates the total kernel execution time. When scaling a measurement from one GPU to another, two adjustments are required: the number of waves changes, since a destination GPU with more SMs processes more thread blocks per wave and therefore needs fewer waves, and the speed of each wave changes, since memory and compute operations scale differently across hardware. A GPU with a higher clock frequency executes compute operations faster, whereas a GPU with greater memory bandwidth executes memory operations faster. Wave scaling therefore applies both a memory-bandwidth ratio and a clock-frequency ratio simultaneously, and uses the roofline model to weight them: the relative contribution of each ratio is controlled by a blending factor γ :

- if γ is near 1, the kernel is mostly memory-bound;
- if γ is near 0, the kernel is mostly compute-bound;
- if $\gamma = 0.5$, an even blend is assumed.

Habitat computes γ for each kernel by measuring its arithmetic intensity using CUPTI and comparing it against the destination GPU’s ridge point in the roofline model. Combining these factors, the authors of Habitat arrive at the

following equation [6]:

$$T_d = T_o \cdot \frac{\left\lceil \frac{B}{W_d} \right\rceil}{\left\lceil \frac{B}{W_o} \right\rceil} \cdot \left(\frac{D_o}{D_d} \cdot \frac{W_d}{W_o} \right)^\gamma \cdot \left(\frac{C_o}{C_d} \right)^{1-\gamma}. \quad (2)$$

where T_d is the predicted execution time on the destination GPU, T_o is the measured execution time on the origin GPU, B is the total number of thread blocks in the kernel, W_o and W_d are the number of thread blocks per wave on the origin and destination GPU respectively, D_o and D_d are the memory bandwidths, C_o and C_d are the GPUs’ clock frequencies, and $\gamma \in [0, 1]$ represents the memory-bandwidth boundedness of the kernel as determined by the roofline model. The term $\left(\frac{D_o}{D_d} \cdot \frac{W_d}{W_o} \right)^\gamma$ captures the combined effect of memory bandwidth and wave size on memory-bound execution, while $\left(\frac{C_o}{C_d} \right)^{1-\gamma}$ models the scaling of compute-bound execution.

Wave scaling assumes that the same kernel runs on both GPUs, so that the number of thread blocks B and the arithmetic intensity (and therefore γ) remain unchanged between origin and destination. This assumption fails for operations whose kernels differ across GPU architectures, where the code, the number of thread blocks, and the arithmetic intensity all change. Habitat handles these operations with a separate mechanism, described next.

MLP Predictors. While wave scaling assumes that both GPUs run the exact same kernel, libraries like cuDNN (for convolutions) and cuBLAS (for matrix operations) do not use a fixed kernel. They select a kernel based on which is fastest for the GPU architecture they are running on. It is therefore possible that the origin GPU runs a convolution using one algorithm, whereas the destination GPU runs the same convolution with a different one. As the algorithm changes, the number of thread blocks and the arithmetic intensity also change, so wave scaling no longer has a valid anchor. These operations are called kernel-varying operations. Habitat supports four kernel-varying operations: 2D convolution, LSTM, batched matrix multiply, and linear layers. Operations that use the same kernel across architectures, such as ReLU, addition, and element-wise operations, are referred to as kernel-alike operations.

For kernel-varying operations, since no measured kernel time can be reliably scaled, Habitat predicts their execution time directly using a multilayer perceptron (MLP). To produce a prediction, the MLP requires the operation’s specifications (batch size, number of input and output channels, kernel size, stride, image size, and so on) together with the hardware features of the target GPU (memory capacity, memory bandwidth, SM count, and peak FLOPS). The hardware features are what allow a single trained MLP to make predictions for many different GPUs. Habitat uses four separate MLPs, one per kernel-varying operation. Training a dedicated MLP per operation is feasible precisely because only a small number of operations are kernel-varying, so a handful of specialized models suffices to cover them [6]. Each MLP is pre-trained on a large dataset of configurations of its operation, measured

across the six GPUs; for the convolution MLP this amounts to approximately 91,000 configurations. The trained MLPs are then shipped inside Habitat.

Habitat combines the two mechanisms by selecting one per operation and summing the predicted times to obtain the full iteration prediction: kernel-alike operations are estimated through wave scaling, while kernel-varying operations are estimated using their pre-trained MLP.

Accuracy and Evaluation. To evaluate Habitat’s prediction accuracy, the authors ran training iterations across six GPUs (Pascal P4000 and P100, Volta V100, and Turing 2070, 2080Ti, and T4) and five DNNs (ResNet-50, Inception v3, Transformer, GNMT, and DCGAN), covering image classification, machine translation, and image generation. Across all of these, the predictions resulted in an average error of 11.8% [6].

The evaluation was set up for predictions across all 30 possible origin-destination GPU pairs. Every GPU was used both as source and target, confirming that the method works regardless of which GPU is the origin, in both the weak-to-strong and strong-to-weak direction. While the error rate is low, the main intent of assisting the GPU selection process is demonstrated through two case studies.

In case study 1, a user is assumed to own a P4000 (the origin GPU) and wishes to train GNMT. They want to decide whether to rent a P100, T4, or V100 in the cloud. The user receives two pieces of information:

- The V100 is fastest (up to $4\times$ the P4000), the P100 is in the middle (up to $2.3\times$), and the T4 gives only a marginal boost (up to $1.4\times$).
- However, for cost-efficiency the best option is the T4, despite being the slowest of the three.

The “best” GPU therefore depends on what the user wishes to optimize for, and Habitat surfaces both options with the supporting reasoning. While the absolute numbers carried a 10.7% average error, Habitat correctly ranked all three GPUs on both throughput and cost-efficiency, accomplishing its goal of decision assistance.

In case study 2, the user is assumed to own a 2080Ti (the origin GPU) and wishes to train DCGAN, deciding whether to upgrade to a V100, a more expensive server GPU. Intuitively, the V100 should perform better since it is the more powerful card; however, Habitat predicts that it gives essentially no benefit, with a speedup of only $1.1\times$ over the 2080Ti. This is because DCGAN is a computationally light model and does not generate enough computation to utilize the V100’s additional resources. This case study showed a 7.7% average error, yet was again able to answer the user’s question correctly.

While the duration estimations are not perfect and carry around 7–11% absolute error, they correctly preserve the relative GPU rankings, which is what matters for the selection decision.

Limitations. While Habitat has relatively low error rates and achieves this accuracy by relying on direct measurement, it still carries certain limitations. Wave scaling requires that

an operation use the same kernel on both GPUs, and as covered earlier, the MLPs exist to patch the kernel-varying operations. However, even the MLPs cover only four kernel-varying operations; a new kernel-varying operation would require a newly trained MLP from scratch. Habitat is therefore not fully versatile, as it is general only for the operations already covered. A second limitation is that Habitat profiles at a chosen batch size, so that batch size must physically fit in the origin GPU’s memory. Predicting for batches larger than the local GPU can hold is suggested to be resolved through linear extrapolation as a workaround, but the case where only a single batch size fits is left unresolved. Finally, Habitat models only a single GPU’s computation. It cannot predict distributed training, which requires communication between multiple GPUs; the authors note that Habitat’s predictions could feed into separate communication models, although this is not yet implemented.

Daydream. Habitat scales a measured iteration from one GPU to another; profiling data can, however, also answer a different predictive question on the *same* hardware: how much would a given software optimization improve training time, without implementing the optimization. Daydream addresses this what-if question [7]. In a first phase, Daydream uses CUPTI to collect a low-level trace of one training iteration, containing every CPU and GPU task together with its name, start time, duration, and stream. From this trace it constructs a kernel-granularity dependency graph of the iteration, exploiting the observation that the fine-grained tasks in DNN training are highly sequential, so only a limited number of inter-thread dependencies must be identified. Using knowledge of the training procedure, Daydream then maps the thousands of low-level tasks back to the DNN layers they belong to. Optimizations are modelled as simple graph transformations on this dependency graph, such as shrinking task durations (e.g., for mixed precision arithmetic, which performs computations in lower-precision formats such as 16-bit floats to speed up memory-bound kernels), fusing nodes (kernel fusion), or reordering communication; the runtime of the transformed graph is then predicted by simulating its execution. The authors evaluate Daydream with five optimizations on five DNN models across three applications and show that it reliably distinguishes optimizations that yield significant improvements from those that do not; for example, it estimates that mixed precision training improves the iteration time of BERT_{LARGE} by 17.2%, with less than 3% prediction error [7]. Daydream thus demonstrates the breadth of the profiling-based paradigm: the same kernel-level measurements that Habitat scales across hardware can also be restructured to explore hypothetical software configurations. Its limitation is the mirror image of its strength: since every prediction is anchored in a trace of the actual workload, the workload must be executable on the profiled hardware, and effects that change kernel behaviour in ways not expressible as graph transformations remain out of reach.

Notably, Habitat already relies on an MLP component for part of its predictions. This component is complementary

within Habitat’s hybrid design, but it raises the question of whether wave scaling itself could be replaced by a machine learning model. The following section explores performance prediction carried out entirely through machine learning, without the hybrid structure used by Habitat.

V. ML-BASED PREDICTORS

Machine-learning-based predictors treat performance prediction as a supervised learning problem: a model is trained on measured samples of workload configurations, hardware features, and the resulting latencies, and is then queried for combinations it has not seen. The MLPs inside Habitat are already an instance of this idea, restricted to four operation types. The decisive question for this paradigm is not whether such a model can fit its training data, but whether it *extrapolates*: to larger operator dimensions, to newer network architectures, and to GPUs that did not exist when the training data was collected.

Limits of direct latency prediction. The authors of NeuSight examine this question empirically for two representative prior predictors [8]. Habitat’s MLPs, which map operator dimensions and GPU features directly to a latency, were trained on older-generation GPUs (V100, 2080Ti, T4, RTX2070, P100, and P4000) and on matrix dimensions up to 1024. When queried outside this range, their error grows to 38% for larger matrix dimensions and up to 127% on the unseen A100 GPU. The second baseline, by Li et al., fits a linear regression between the FLOP count of a kernel and its latency for each GPU, and extrapolates to unseen GPUs through a linear relationship with memory bandwidth [9]. This approach fails for small operator dimensions, where the GPU is underutilized and latency does not scale linearly, and for unseen GPUs, where the bandwidth-based extrapolation is too simplistic [8]. Notably, increasing the capacity of the learned model does not resolve the problem: the NeuSight authors show that deeper MLPs and transformer-based predictors still exceed 70% error on out-of-distribution operator dimensions [8]. The difficulty is therefore structural rather than a matter of model size, which motivates a predictor that embeds knowledge of how GPUs actually execute kernels.

NeuSight. NeuSight is a forecasting framework that predicts the latency of deep learning models, for both training and inference, on GPUs that are unavailable to the user, without executing the workload on the target GPU [8]. Its central insight concerns how GPU libraries execute deep learning kernels: the output of a kernel such as a General Matrix Multiplication (GEMM) is partitioned into multiple identical *tiles*, which are dispatched to the SMs and executed independently in successive waves — the same wave structure that Habitat’s wave scaling exploits. Rather than predicting the latency of the whole kernel directly, NeuSight predicts the latency of a single tile and reconstructs the kernel latency from it. This decomposition is what restores generalization: typical tile dimensions used by GEMM libraries range only from 32 to 256, so even when a kernel’s dimensions lie far

outside anything seen during training, the tiles it decomposes into remain within a small, familiar range.

The second design decision is that the machine learning model does not output a latency at all. Instead, a small MLP predicts the *utilization* of the device for a tile, that is, the fraction of the roofline throughput of Equation (1) that the tile actually achieves. The MLP receives the operator dimensions and derived per-tile features together with the architectural parameters of the target GPU — memory capacity, memory bandwidth, number of SMs, L2 cache size, and peak FLOPS — all of which are available from public specification sheets. The predicted utilization then scales the analytically computed roofline bound to obtain the tile latency. Because the prediction is expressed relative to the roofline, it can never violate the fundamental performance limits of the hardware; the learned component only captures the non-linear gap between peak and achieved performance. NeuSight trains separate predictors for the main kernel categories (matrix-multiplication-based, element-wise, and reduction-based operators) and also supports operator fusion, which merges the memory traffic of fused operators.

Given the per-tile prediction, the kernel latency is assembled from the tiling structure. For a kernel with output dimensions x_i and tile dimensions t_i , the number of tiles and waves follows as

$$n_{\text{tiles}} = \prod_{i=1}^N \left\lceil \frac{x_i}{t_i} \right\rceil, \quad n_{\text{waves}} = \left\lceil \frac{n_{\text{tiles}}}{n_{\text{SM}}} \right\rceil, \quad (3)$$

and the kernel latency is estimated as

$$T_{\text{kernel}} = T_{\text{tile}} \times n_{\text{waves}}. \quad (4)$$

The parallel to Equation (2) is apparent: where Habitat rescales a wave time measured on an origin GPU, NeuSight predicts the wave time from scratch, which removes the need for an origin measurement altogether.

On top of the kernel predictions, NeuSight operates in three stages. First, the latency of each kernel in the workload is predicted as described. Second, the per-kernel latencies are aggregated over the dataflow graph of the model to obtain the per-GPU latency, under the assumption that kernels execute sequentially on the device — an assumption that was empirically validated for mainstream frameworks by the Daydream authors [7]. Third, for models that are distributed across the GPUs of a server, NeuSight predicts the latency of the required communication operators, such as ring-based all-reduce and point-to-point send/receive, by estimating link utilization and scaling with the peak interconnect bandwidth of the target system. Parallelization strategies split a model or its data across GPUs in different ways: data parallelism replicates the model and divides the input batch, tensor parallelism splits individual operators across devices, and pipeline parallelism assigns different layers to different devices in stages. Pipeline bubbles are the idle periods during which some devices wait for others to finish their stage. These communication

latencies are inserted into the execution according to the user-provided parallelization strategy and combined with the per-GPU latencies to forecast end-to-end distributed performance.

Accuracy and Evaluation. NeuSight is evaluated on seven GPUs spanning six architecture generations (P100, T4, V100, A100-40GB, A100-80GB, L4, and H100) and five transformer-based workloads (BERT, GPT-2, GPT-3, OPT, and Switch Transformer), for both training and inference [8]. Crucially, the H100, L4, and A100-80GB GPUs are excluded from the training data entirely, while for GPT-3 and OPT only a subset of model dimensions is seen during training, so the headline results still measure genuine extrapolation. Across this suite, NeuSight achieves an average percentage error of 8.9%, compared to 140% for the MLP-based approach of Habitat and 60.8% for the linear-regression approach of Li et al. The most striking single result concerns GPT-3 on the H100: the H100 was entirely absent from training, and the GPT-3 configuration tested uses a matrix dimension of 2048, beyond the maximum of 1024 seen during training, making it an out-of-distribution operator dimension. Under this combined GPU- and model-level extrapolation, NeuSight reduces the percentage error from 121.4% (training, versus Habitat) and 30.8% (inference, versus Li et al.) to 2.3%. For distributed execution on 4-GPU servers (an H100 DGX system and an A100 NVLink system, where DGX denotes NVIDIA’s integrated multi-GPU server and NVLink its high-bandwidth GPU-to-GPU interconnect), NeuSight predicts the latency of GPT-2 and GPT-3 with an average error of 5.4%.

Limitations. NeuSight’s accuracy rests on the assumption that libraries execute kernels through the tiling scheme it models; kernel implementations that deviate substantially from this convention, such as hand-optimized fused kernels, may not be captured. The sequential-execution assumption for aggregation excludes workloads with significant inter-kernel overlap. Distributed prediction currently covers only GPUs within a single server, so cross-server network effects are out of scope, although the authors note that NeuSight could be combined with network simulators. Finally, although no measurement on the target GPU is needed at prediction time, the framework does require an up-front dataset of kernel measurements collected across a set of available older GPUs, which constitutes a one-time but non-trivial setup cost.

VI. DISCUSSION

Table I summarizes the five examined systems. The comparison across the three paradigms can be organized along four questions: how accurate the predictions are, how well they generalize, what they cost to set up and use, and what each method is actually for.

Accuracy versus generality. The three paradigms trade accuracy against generality in different ways. Analytical models generalize by construction: the Hong and Kim model or Paleo can, in principle, be applied to any kernel or network for which the inputs can be derived, without any training data. Their fidelity, however, is bounded by their assumptions, and the examined cases show precisely where these assumptions

break: the missing cache model of Hong and Kim, the peak-hardware assumption of Paleo that required the PPP correction, and the fc6 layer whose data-layout transformations no FLOP count reveals. Profiling-based methods sit at the opposite end. A measurement captures everything, including library kernel selection and achieved rather than theoretical throughput, which is why Habitat reaches 11.8% error across thirty GPU pairs and Daydream predicts optimization effects with errors mostly in the single digits to low teens, reaching up to roughly 16% in its most difficult configurations. The price is that the fidelity is anchored to hardware that must be available. Purely learned predictors, finally, are only as general as their training distribution: the same Habitat MLPs that perform well within their training range degrade to 127% error on an unseen GPU. NeuSight shows that this is not an inherent property of the paradigm but a consequence of predictor design; by restricting the learned component to a bounded sub-problem (per-tile utilization) and clamping it with the roofline, it generalizes to an unseen model on an unseen GPU with 2.3% error.

Convergence toward hybrids. A second observation is that none of the competitive methods is a pure representative of its paradigm. Paleo, the archetypal analytical model, requires the measured PPP constant. Habitat, the archetypal profiling approach, ships four pre-trained MLPs because scaling breaks for kernel-varying operations. NeuSight, the archetypal learned predictor, embeds the roofline model and the wave-based execution structure — both analytical artifacts — directly into its architecture. The spectrum from pure hardware modelling to pure learning is therefore largely unoccupied at its endpoints. What distinguishes the methods is not whether they combine paradigms, but which component carries the main predictive burden and which components correct its blind spots. Viewed this way, the historical progression of the examined systems is a migration of the learned component from the periphery (a patch for four operations in Habitat) to the center (the utilization predictor in NeuSight), accompanied by a migration of the analytical knowledge from the center (the CPI equations of Hong and Kim) to the periphery (bounds and structural priors in NeuSight).

Setup cost and cost per prediction. The paradigms also differ in where their costs fall. Analytical models are essentially free per prediction but expensive per *architecture*: each hardware generation may invalidate modelling assumptions, as the cache assumption of Hong and Kim illustrates, and updating an analytical model is manual expert work. Profiling-based methods are expensive per *workload*: every new model requires at least one profiled run, and Daydream additionally requires framework instrumentation. Learned predictors are expensive *up front*: NeuSight requires a measurement campaign across a fleet of (older) GPUs to build its training set, but afterwards a prediction for a new model or a new GPU requires only the operator graph and a specification sheet. For a practitioner, the relevant cost is therefore determined by how often the workload, the hardware, or both change. It is also worth noting that absolute accuracy is not always the metric that matters: Habitat’s case studies demonstrate that a

TABLE I
COMPARISON OF THE EXAMINED PERFORMANCE PREDICTION METHODS.

Method	Paradigm	Prediction target	Required inputs	Target GPU access	Reported error
Hong & Kim [4]	Analytical	Single kernel execution time	Kernel instruction and warp analysis, microarchitectural parameters	Not required	5.4% (micro-bench.), 13.3% (Merge bench.)
Paleo [5]	Analytical with empirical calibration	DNN training time and scalability	Network architecture, peak hardware specifications, PPP calibration benchmark	Short calibration benchmark	<3% (1 worker) up to ~14% (8 workers), derived from [5], Tab. 4
Habitat [6]	Profiling + ML hybrid	Training iteration time on a destination GPU	One measured iteration on an origin GPU, destination specifications	Not required (origin GPU required)	11.8% average
Daydream [7]	Profiling + simulation	Runtime under hypothetical optimizations	CUPTI kernel-level trace of a baseline iteration	Required (baseline run)	e.g. <3% for mixed precision on BERT _{LARGE}
NeuSight [8]	ML bounded by performance laws	Training and inference latency, single- and multi-GPU (per server)	Operator graph of the model, GPU specification sheet	Not required	8.9% average; 2.3% for GPT-3 on H100

predictor with 7–11% absolute error can still rank candidate GPUs perfectly, which is sufficient for a selection decision.

Scope and practical guidance. Finally, the methods answer different questions and are partly complementary rather than competing. For a quick feasibility estimate of a new architecture, or when designing hardware that does not exist yet, analytical models remain the only option, since there is nothing to profile and no data to learn from. When an origin GPU is available and the question is which cloud GPU to rent for a conventional workload, Habitat answers it with a single profiled iteration. When the hardware is fixed and the question is which software optimization to implement first, Daydream’s what-if analysis applies. When neither the target model nor the target GPU is accessible — the situation of practically everyone evaluating a flagship GPU at launch — NeuSight currently offers the best combination of accuracy and generality, and its per-server communication model even covers small-scale distributed execution. Combinations are natural: Paleo-style communication modelling or dedicated network simulators can extend per-device predictors such as Habitat or NeuSight to the cluster scale, and profiled traces of the kind Daydream collects could serve as training data for learned predictors.

VII. CONCLUSION AND OUTLOOK

This paper compared three families of performance prediction methods for GPU-based deep learning through five representative systems. Analytical models, exemplified by the Hong and Kim model and Paleo, predict from mathematical descriptions of workload and hardware; they are cheap and universally applicable but limited by their assumptions and require empirical calibration in practice. Profiling-based methods, exemplified by Habitat and Daydream, achieve high fidelity by anchoring predictions in real measurements, at the cost of requiring hardware access, and already incorporate learned components where measurement-based scaling breaks down. Fully learned predictors that map workload and hardware features directly to latency fail to extrapolate, but NeuSight demonstrates that a learned predictor which is

decomposed along the GPU’s tiled execution structure and bounded by the roofline model generalizes to unseen models on unseen GPUs with errors of a few percent. The main takeaway of the comparison is that the three paradigms are not alternatives but ingredients: the most capable current systems combine analytical structure, measured data, and learned components, and differ mainly in which ingredient dominates.

It should be noted that this comparison rests on one or two representative systems per paradigm; a broader survey covering more systems, such as Calculon or graph-neural-network-based predictors, could reveal additional trade-offs not captured here. Several directions for future work also remain open. Current predictors cover a single GPU or a single multi-GPU server, whereas frontier models are trained on clusters of thousands of GPUs, where network topology, congestion, and parallelization strategy dominate runtime; integrating per-device predictors with network models is a natural next step. Compiler-driven optimizations such as aggressive operator fusion and hand-written attention kernels increasingly blur the kernel abstraction on which all examined methods rely, and predictors will need to track these software changes as much as hardware changes. Extending prediction beyond NVIDIA hardware to other vendors and accelerator classes, and predicting energy consumption alongside runtime, would further broaden the practical value of this line of work, since energy is becoming as significant a constraint as time in large-scale deep learning.

Finally, this comparison captures a fast-moving field. NeuSight, the system examined here, has itself already begun to be superseded: hybrid analytical–learning successors such as PipeWeave [13] target the same unseen-GPU forecasting problem and report substantially lower error, particularly on the low-precision transformer kernels that dominate modern large-language-model workloads. This reinforces the paper’s central observation that the field is converging on hybrid designs, while underlining that any snapshot of the state of the art is necessarily provisional.

REFERENCES

- [1] S. Williams, A. Waterman, and D. Patterson, “Roofline: An insightful visual performance model for multicore architectures,” *Commun. ACM*, vol. 52, no. 4, pp. 65–76, Apr. 2009.
- [2] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. Cambridge, MA, USA: MIT Press, 2016.
- [3] NVIDIA Corporation, *CUDA C++ Programming Guide*, 2024.
- [4] S. Hong and H. Kim, “An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness,” in *Proc. 36th Annu. Int. Symp. Computer Architecture (ISCA)*, Austin, TX, USA, 2009, pp. 152–163.
- [5] H. Qi, E. R. Sparks, and A. Talwalkar, “Paleo: A performance model for deep neural networks,” in *Proc. Int. Conf. Learning Representations (ICLR)*, Toulon, France, 2017.
- [6] G. X. Yu, Y. Gao, P. Golikov, and G. Pekhimenko, “Habitat: A runtime-based computational performance predictor for deep neural network training,” in *Proc. USENIX Annu. Technical Conf. (ATC)*, 2021, pp. 503–521.
- [7] H. Zhu, A. Phanishayee, and G. Pekhimenko, “Daydream: Accurately estimating the efficacy of optimizations for DNN training,” in *Proc. USENIX Annu. Technical Conf. (ATC)*, 2020, pp. 337–352.
- [8] S. Lee, A. Phanishayee, and D. Mahajan, “Forecasting GPU performance for deep learning training and inference,” in *Proc. 30th ACM Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Rotterdam, Netherlands, 2025, pp. 493–508.
- [9] Y. Li, Y. Sun, and A. Jog, “Path forward beyond simulators: Fast and accurate GPU execution time prediction for DNN workloads,” in *Proc. 56th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Toronto, ON, Canada, 2023.
- [10] H. Zhu, M. Akrouf, B. Zheng, A. Pelegrini, A. Jayarajan, A. Phanishayee, B. Schroeder, and G. Pekhimenko, “Benchmarking and analyzing deep neural network training,” in *Proc. IEEE Int. Symp. Workload Characterization (IISWC)*, Raleigh, NC, USA, 2018.
- [11] NVIDIA Corporation, *CUPTI: CUDA Profiling Tools Interface*, 2024. [Online]. Available: <https://docs.nvidia.com/cupti/>
- [12] M. D. Linderman, J. D. Collins, H. Wang, and T. H. Meng, “Merge: A programming model for heterogeneous multi-core systems,” in *Proc. 13th Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Seattle, WA, USA, 2008, pp. 287–296.
- [13] K. Zhang, Y. Cui, S. Zhang, C. Ding, S. Qian, L. Wang, J. Cao, G. Xue, C. Huang, G. Yang, and L. Zhang, “PipeWeave: Synergizing analytical and learning models for unified GPU performance prediction,” in *Proc. 53rd Annu. Int. Symp. Computer Architecture (ISCA)*, 2026.