xAMM: “Attention” to Details Improves Cross-Platform Prediction Accuracy

Aakash Raj Dhakal*, Tanzima Z. Islam*, Arunavo Dey*, Daniel Nichols†, Abhinav Bhatele†,
Tapasya Patki‡, Tom Scogland‡ and Jae-Seung Yeom‡
*Texas State University †University of Maryland ‡Lawrence Livermore National Laboratory
{nvc22,tanzima,hcs77}@txstate.edu {dnicho,bhatele}@umd.edu {patki1,scogland1,yeom2}@llnl.gov

Abstract—**As computing becomes the major enabler in more and more fields, computing platforms also have become more heterogeneous than ever before to support different needs. Inevitably, high performance computing (HPC) centers and cloud vendors offer a diverse array of computing platforms to the user, often to a point where it overwhelms users as well as system managers. Therefore, a cross-platform performance prediction model, which leverages observations from one platform to predict performance on another, can be extremely valuable. However, building such a model for numerous platforms requires an enormous amount of effort to collect training data, which is often prohibitively expensive. To overcome this challenge, we propose xAMM[1], an end-to-end Machine Learning (ML) pipeline that uses the attention mechanism, a transformative concept in generative AI, for two purposes: learning smart embeddings from raw application performance samples and constructing Abstract Machine Models (AMMs)–compact representations of machine properties. By integrating performance sample embeddings with AMMs where available, xAMM improves the accuracy of the state-of-the-art XGBoost model by 49.64% for CPU $\rightarrow$ CPU and 99.07% for CPU $\rightarrow$ GPU prediction compared to building the model using raw data, a common approach in the existing literature.**

Index Terms—**Cross-Platform Performance Prediction, Attention, Embedding Learning, Abstract Machine Model.**

I. Introduction

As more and more scientific discoveries and engineering campaigns rely on interdisciplinary studies, a single workflow tends to consist of mixed tasks that are better supported by different computing platforms. Cross-platform performance prediction for heterogeneous High Performance Computing (HPC) systems has become critical in today’s rapidly evolving technological landscape. As new architectures are released approximately every six months, scientists and engineers face a significant challenge in understanding how their set of applications will perform on these platforms. The delay in answering these questions can be prohibitive, as traditional performance measurement methods are time-consuming and require days or weeks. Performance prediction across platforms without extensive benchmarking enables users, software systems, and facilities to make informed decisions about several HPC tasks, including code optimization, and machine procurement.

There exist prior studies on cross-platform performance prediction. Some rely on analytical models or simulations [1], [2]. Nevertheless, adapting to rapidly evolving architectures at a reasonable computational cost to capture the behavior of representative HPC applications remains exceedingly challenging. Others use a categorical feature to distinguish platforms, static hardware descriptions, such as core counts and clock speeds, or application data from target platforms [3], [4]. These methods lack a systematic approach to represent machines dynamically, relying instead on ad hoc, oversimplified static information, resulting in an inextensible model even for similar but unseen platforms. Without dynamic signatures–capturing key properties such as memory bandwidth, computational intensity, and scalability–such models risk poor generalizability as hardware evolves. **We fill this gap by proposing a principled approach for constructing such dynamic machine “signatures” called Abstract Machine Model (AMM).**

Moreover, most ML-based performance prediction studies [3], [4], [5], [6], [7], [8] use raw data to build downstream prediction models, which can obscure critical correlations across features and samples needed for accurate predictions. As downstream predictive models, many leverage tree-based models such as XGBoost [9] and neural networks [5], [10], [11]. In contrast, we propose to transform raw data into embeddings using “attention”, a genAI mechanism, that calculates weights based on feature correlations and sample importance to transform raw data into a compact representation before data is input to a downstream model. **This approach captures discriminatory information that improves the accuracy of downstream models regardless of the model architecture, making our methodology complementary to existing ones.**

To address both gaps, we propose a unified data transformation methodology that: ① Transforms raw performance samples into embeddings using attention. ② Constructs machine-specific dynamic signatures by capturing key properties such as memory bandwidth, computational intensity, and scalability, derived from running benchmark applications such as RAJAPerfSuite [12], [13] under various configurations (e.g., number of threads). Specifically, we propose to calculate attention weights for each sample in the machine benchmark dataset based on its feature correlations. These weights are then used to compute a weighted average of the samples, where each sample is multiplied by its corresponding weight. This process produces a single embedding vector that serves as the machine’s signature, effectively summarizing its dynamic performance characteristics in a compact representation. ③

[1]Pronounced as “Exam”

Creates a unified application sample representation by concatenating the application sample embeddings learned in (1) with the source and target machines' AMMs, if available.

We implement our proposed data transformation methodology into an automated end-to-end data transformation and modeling framework–`xAMM`–where users can deploy their downstream modeling architecture while `xAMM` constructs unified input to these models. Our extensive evaluations demonstrate the benefit of each step and their combined impact using several scientific applications on both CPU and GPU platforms. While (1) is mandatory, (2) is optional and is calculated when machine benchmark data or pre-constructed AMMs are available[2]. Whenever a new machine is considered, if a machine benchmark dataset is available, `xAMM` can construct its AMM offline and save it for future use.

During the training of the downstream predictive model, which is an offline process, users provide performance profiles of scientific applications collected on various machines. If a training machine's AMM is available, `xAMM` uses it. Otherwise, either users must provide a machine benchmark dataset or `xAMM` skips steps (2) and (3). Figure 1 presents a high-level overview of `xAMM`'s methodology.

During inference, users provide performance measurements of a potentially new application on a reference machine. To take advantage of AMMs, one of two conditions must be met: (1) if both the source and target machines were included during training, users need only input their names and `xAMM` reuses their precomputed AMMs; or (2) if the target machine was not part of training but its benchmark dataset is available, `xAMM` constructs its AMM. However, the reference machine must always be included in the training dataset to utilize AMMs effectively. If AMMs for either machine are unavailable, `xAMM` skips steps (2) and (3) and proceeds with step (1).

To summarize, the technical contributions of this paper are:

- Attention-based sample embedding learning;
- Novel principled approach to construct "abstract machine model" that captures dynamic machine characteristics;
- An end-to-end data transformation pipeline, `xAMM` that integrates sample embedding learning and optional AMM construction, enabling effective data transformation for improved cross-platform prediction.
- A unique machine benchmark dataset and their corresponding performance signatures (AMMs) from 16 platforms.

II. Background

II-A Cross-platform performance prediction

Cross-platform performance prediction estimates how computational tasks will perform across different hardware architectures, including various CPUs and GPUs. It is commonly used in performance-aware scheduling within multi-resource environments. Several studies have addressed this problem by exploring diverse methodologies such as transfer learning [10], neural networks [5], and tree-based methods [3], [4]. These works primarily use raw data to predict absolute execution times or build increasingly complex downstream models. While recent trends emphasize tree-based methods for their interpretability over neural network-based black-box architectures [4], none have highlighted data transformation as a critical factor in enhancing predictive performance.

II-B Embedding

Embeddings are a powerful mathematical concept in machine learning that transforms high-dimensional data into more compact, low-dimensional representations while preserving essential relationships and structures. Mathematically, an embedding is a function $f : X \to X'$ that maps elements from a high-dimensional space X to a lower-dimensional space X'. The embedding function is typically learned through optimization processes that minimize a loss function, ensuring that similar items in the input space remain close in the embedded space. Physically, embeddings can be visualized as points in a lower-dimensional space. In this paper, we use t-SNE plots to visually validate that embeddings from samples of similar architectures remain closer than different ones. Since embeddings compress high-dimensional data, it mitigates the curse of dimensionality for the downstream models [14], [15].

III. Dataset Description

Figure 1 presents an overview of the `xAMM` system, showing that two datasets are input to `xAMM` during training.

III-A Machine Benchmarking Dataset

A set of quantitative benchmarking results provides more insight into a machine's performance characteristics than a textual specification from a machine catalog. Each sample includes the machine name, core counts or thread counts, and a set of benchmark results. The input features are represented as a vector $\mathbf{Z} = [z_1, z_2, \ldots, z_n]$, where $n = 64$ denotes the total number of characteristics measured by the benchmark. We leverage RAJAPerfSuite [16], which evaluates a variety of performance tests implemented on the RAJA portability framework for both CPUs and GPUs. Without such a standardized benchmark, comparing the performance across machines becomes a challenging task.

For example, the `Basic_DAXPY` kernel in RAJAPerfSuite performs a vector addition operation $(y = a * x + y)$, representing memory bandwidth-bound computations common in scientific applications. This kernel helps characterize a machine's memory subsystem performance. Another example is the `Basic_REDUCE` kernel, which performs a sum reduction across an array. This operation tests both the machine's ability to parallelize reductions efficiently and its memory bandwidth, providing insights into the balance between the computational and memory capabilities of the system. These are lightweight kernels, requiring minimal data collection effort. Such data are often collected during acceptance testing of a new system, and therefore are already available. Even if not, we only need to gather once per system. **Our work is the first to use these**

[2]We have collected such machine benchmark datasets using RAJAPerfSuite across 16 diverse platforms and will make the AMMs publicly available in https://github.com/LLNL/generalizable_modeling.git.

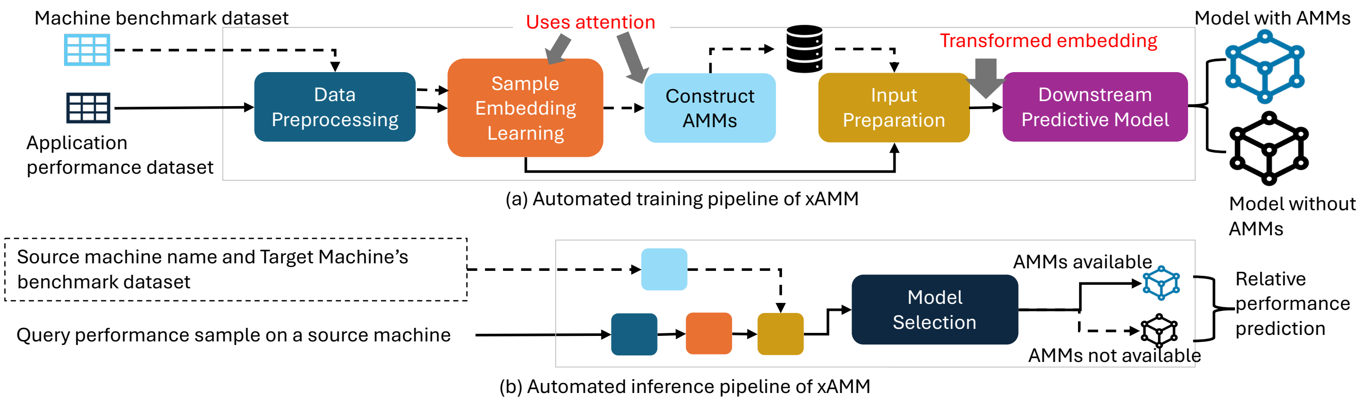

Machine benchmark dataset
Application performance dataset
Data Preprocessing
Uses attention
Sample Embedding Learning
Construct AMMs
Transformed embedding
Input Preparation
Downstream Predictive Model
Model with AMMs
Model without AMMs
(a) Automated training pipeline of xAMM
Source machine name and Target Machine's benchmark dataset
Query performance sample on a source machine
Model Selection
AMMs available
AMMs not available
Relative performance prediction
(b) Automated inference pipeline of xAMM

Fig. 1: Illustration of the `xAMM` system: Dotted lines and boxes represent optional components. In (a),`xAMM` calculates and saves AMMs from the machine benchmark dataset, using them during training to transform application sample embeddings. In (b), box colors correspond to functionalities labeled in the training pipeline. During inference, if the target machine was included during training, its AMM is reused; otherwise, `xAMM` constructs target's AMM using its machine's benchmark dataset (if provided) or skips this step. `Input Preparation` concatenates application sample embeddings with the source and target machines' AMMs, when available.

benchmark runs to build a signature for that system.

III-B Application Performance Dataset

In addition to machine benchmarking data, we collect hardware performance counter data for various HPC applications executed on different machines. Hardware performance counters capture dynamic interactions between applications and hardware. We measure the PAPI [17] counters using HPCToolKit [18]. Each sample includes the application parameters, the machine name, configuration parameters (e.g., the number of threads or tasks), and the measured hardware performance counters. The input features are represented as a vector $\mathbf{X} = [x_1, x_2, \ldots, x_m]$, where $m = 14$ is the number of counters in an application performance sample.

The methodology presented in this paper is generic and can be applied to any set of features chosen by a user. While collecting hardware performance counters can be time-consuming, they serve as low-level performance fingerprints, capturing critical insights into application behavior. Although this initial data collection introduces some overhead, it is performed only once during the construction of the source model. For an inference task, performance data can be collected on a reference machine by running a smaller pilot job, designed to replicate the computational characteristics of a larger ensemble job within the same scientific workflow.

III-C Dataset Homogeneity Assumption

In this work, we assume a homogeneous dataset where all source applications share the same feature names, differing only in their values. The problem of knowledge transfer when the feature names or numbers do not match across the samples is known as "heterogeneous knowledge transfer" and is complementary to this work and out of scope.

IV. Modeling Methodology

Figure 1 provides an overview of our modeling approach, implemented as an end-to-end Python-based ML pipeline called `xAMM`. As shown in Figure 1(a), `xAMM` precomputes and saves AMMs from a machine benchmark dataset. During training, it performs data preprocessing, learns sample embeddings, constructs AMMs, prepares inputs, and trains downstream models. During inference, `xAMM` executes a subset of these tasks as applicable to the prediction process.

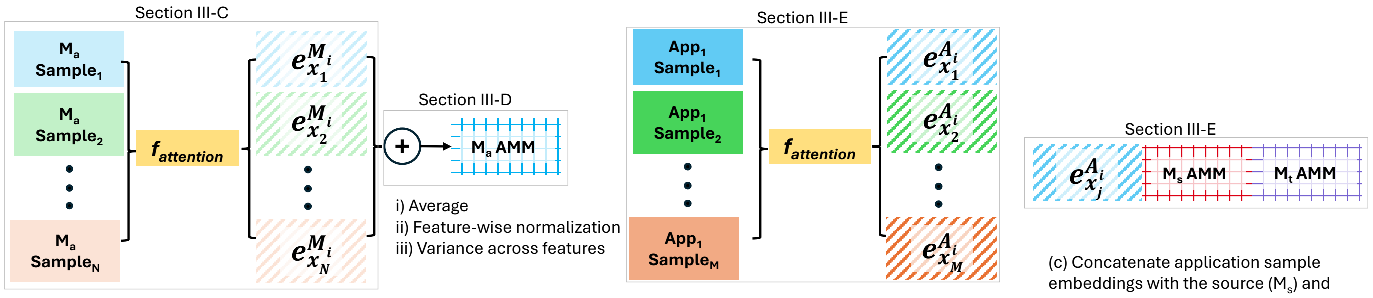
IV-A Problem Formulation: Predict Relative Performance

In this work, we formulate the problem of cross-platform performance prediction as predicting an application's relative performance, not absolute runtime. In contrast to traditional methods that require comprehensive data collection on every machine, using relative performance involves detailed measurements on a single reference machine. This upfront cost is offset by eliminating the need for extensive data collection on new platforms, making the process both scalable and efficient as platforms evolve rapidly. Mathematically, Equation 1 presents relative performance of an application A_i on machine M_2 compared to machine M_1 as:

$$\mathbf{relR}^{A_i}_{M_1 \to M_2} = \frac{R_{M_2}}{R_{M_1}} \tag{1}$$

Where R_{M_1} is the runtime on the source or reference machine, and R_{M_2} is the runtime on a target machine. During training, both runtimes are provided for the model to learn from. During inference, R_{M_1} is provided to the downstream model, while the model predicts $\mathbf{relR}^{A_i}_{M_1 \to M_2}$. A $\mathbf{relR} > 1$ indicates how much faster the application runs on the target machine, while $\mathbf{relR} < 1$ shows how much slower compared to the reference.

Relative performance abstracts away differences in absolute execution times caused by factors such as iteration counts. For instance, in an iterative simulation, total runtime depends on the number of iterations, which can vary even for the same workload. By using relative performance, we normalize these differences and focus on how efficiently each machine executes a single iteration, ensuring consistent and comparable



(a) Learn sample embeddings collected from a benchmark application using attention and combine them to construct an AMM for a particular machine M_a.

(b) Embedding learning for application samples

(c) Concatenate application sample embeddings with the source (M_s) and target (M_t) machines' AMMs to prepare input to the predictive model

Fig. 2: Illustration of different components of the xAMM system.

results. Since cross-platform performance prediction is already challenging, this study focuses on applications with minimal dynamism to further simplify the problem.

IV-B Data Preprocessing

To prepare the data for modeling, the `Data Preprocessing` module in xAMM (Figure 1) first removes incomplete or erroneous samples from the dataset and normalizes by subtracting the mean and dividing by the standard deviation for each feature to address differences in scale across machines. This process centers the data around zero and scales it to unit variance. During training (Figure 1(a)), the `Data Preprocessing` module also calculates the relative performance of each sample by comparing it to similar samples from the same application run on other machines with identical configurations (e.g., the same number of threads used).

IV-C Sample Embedding Learning Using Attention

Representation learning, widely used in ML pipelines, transforms raw data into embeddings—compact, structured representations in a lower-dimensional space that emphasize discriminative features while reducing noise. Studies show that high-quality embeddings enhance the predictive accuracy of downstream models [19]. These observations motivate xAMM to leverage representation learning to generate embeddings from performance samples. Common approaches for representation learning include deep neural networks, AutoEncoders, and Variational AutoEncoders [20], [21], [22].

Instead of using these traditional methods, in this work, we proposed to leverage "attention", a transformative concept in generative AI, to learn embeddings. Intuitively, the attention mechanism mimics how humans selectively focus on certain information while ignoring others. Physically, attention can be considered a focus distribution across input data, where the embedding learning model, typically a neural network architecture, assigns more weight to certain parts of the input based on their relevance to the current task.

Specifically, we leverage "sequential attention mechanism" that dynamically focuses on the most relevant input features across multiple iterations of the embedding learning process and enables the representation learning model to iteratively refine the learned embeddings to capture the most salient features of the data and their correlations. Using Equation 2, xAMM calculates the embedding of a sample x_i, $\mathbf{e}_{x_i}$:

$$\underset{\text{Final performance sample embedding}}{\mathbf{e}_{x_i}} = \underset{\text{Attention function computing sample significance}}{f_{\text{attention}}}\,(\overset{\text{Performance sample}}{\mathbf{x}_i})$$

$$f_{\text{attention}}(\mathbf{x}_i) = \text{Softmax}\left(\frac{\underset{\text{Relative performance}}{Q}\;\overset{\text{Relevant features for predicting relative performance}}{K^T}}{\sqrt{\underset{\text{Number of features}}{d_k}}}\right)\overset{\text{Feature importance}}{V} \tag{2}$$

Where $\mathbf{x}_i$ represents a single performance sample, $f_{\text{attention}}(\mathbf{x}_i)$ gives a quantitative score for a performance sample $\mathbf{x}_i$ denoting its significance in predicting relative performance. In this context, Q represents relative performance, K represents the features that are relevant for predicting relative performance, V represents how important each feature is for predicting relative performance and d_k represents the number of features in the performance samples.

The Softmax $\left(\frac{QK^T}{\sqrt{d_k}}\right)$ term calculates attention weights, determining how much focus to put on each feature of the input data. The final multiplication with V aggregates the values based on these attention weights. The resulting embedding $\mathbf{e}_{x_i}$ captures the most relevant aspects of the performance sample, emphasizing features that are most important for predicting cross-platform performance. The `Sample Embedding Learning` module transforms machine M's benchmark data (Z) into $e^M_{\mathbf{Z}}$ (Figure 2(a)) and application performance samples (X) into $e^A_{\mathbf{X}}$ (Figure 2(b)), for both training and inference.

Rationale for using attention The attention mechanism has been shown to enhance the quality of embeddings [23] by focusing dynamically on the most relevant features. While the idea of attention is not novel, to our knowledge, no one has used attention in performance modeling. In Section VI-B, our experiments demonstrate that using attention compared to not using it while learning embeddings improves the accuracy of a downstream supervised prediction model in transferring knowledge from one machine to another.

Algorithm 1 Computation of $\mathbf{AMM}_M$

```
Require: Input embeddings $\mathbf{e}^M_{x_i}$ for all samples $x_i$ collected from machine $M$
Ensure: Machine-specific Abstract Machine Model (AMM), $\mathbf{AMM}_M$
1: Option 1: Averaging: $\mathbf{AMM}_M = \frac{1}{N_M}\sum_{i=1}^{N_M}\mathbf{e}^M_{x_i}$ // Computes the average of all sample embeddings.
2: Option 2: $L2$-Norm: $\mathbf{AMM}_M = \|\mathbf{e}^M_{x_i}\|_2 = \sqrt{\sum_k(\mathbf{e}^M_{x_i})^2_k}$ // Computes the magnitude of the embeddings.
3: Option 3: Variance: $\mathbf{AMM}_M = \frac{1}{N_M}\sum_{i=1}^{N_M}\left((\mathbf{e}^M_{x_i})_k - \mu_M\right)^2$ // Measures the variance of embedding features.
4: Adjust: $\mathbf{AMM}_M = \mathrm{ReLU}(\mathbf{AMM}_M) + 0.01 * \mathrm{ReLU}(-\mathbf{AMM}_M)$ // Ensures non-negative values.
5: Log Transform: $\mathbf{AMM}_M = \log(1 + \mathbf{AMM}_M)$ // Reduces the impact of large values or outliers.
6: Normalize: $\mathbf{AMM}_M = \mathbf{AMM}_M - \max(\mathbf{AMM}_M)$ // Scales embeddings relative to the maximum value.
```

IV-D Combine Sample Embeddings Using Attention to Construct AMMs

Whenever a new machine M's benchmark data becomes available, xAMM constructs its AMM, denoted as $\mathbf{AMM}_M$, by aggregating the embeddings $\mathbf{e}^M_{z_i}$ collected from the benchmark runs on that machine. Intuitively, an AMM is a weighted average of the samples, which signifies a low-dimensional representation of M's performance characteristics. An AMM is calculated once for every new machine and saved for reuse.

Sample aggregation strategies As an AMM is the weighted average of the samples, we propose two different approaches for calculating the weights. This weight acts as an "attention score" while combining the samples into a unified embedding. (1) $L2$**-Norm**, where xAMM sums the squared embedding feature values and takes the square root, giving higher importance to features with larger values and capturing the overall intensity of the machine's performance. (2) **Variance**, where xAMM calculates the statistical spread of embedding features to quantify the variability in a machine's performance behavior across different configurations and workloads. In this work, we also compare our proposed attention-based aggregation methods against the traditional approach of **Averaging**, where xAMM computes the mean of all sample embeddings.

Algorithm 1 presents the steps for constructing AMM_M in detail. After constructing $\mathbf{AMM}_M$ using one of these methods, xAMM applies additional transformations to refine the representation. It ensures non-negative values by applying the ReLU function, adjusts small negative values with a scaled correction, and reduces the influence of large values through a logarithmic transformation. Finally, xAMM scales $\mathbf{AMM}_M$ relative to the maximum value by subtracting it.

Rationale for using AMM We hypothesize that providing the dynamic characteristics of source and target machines to a predictive model will make it generalize better and improve its accuracy in predicting cross-platform performance. The rationale is that oversimplified static features, such as core counts or clock speeds, are often similar across heterogeneous platforms, failing to provide downstream models with the necessary discriminative information. Without this, models will likely misclassify or conflate different hardware platforms, limiting the model's generalizability. Our detailed experiments in Section VI-D demonstrate that attention-based methods (Options 2 and 3 in Algorithm 1) significantly improve the quality of AMMs (Figure 6) compared to averaging.

IV-E Model Input Preparation

xAMM applies the attention-based representation learning method (Section IV-C) to transform raw performance samples X_j from applications A_i into embeddings $e^{A_i}x_j$ using $\mathbf{f_{attention}}$. Each sample $\mathbf{X^{A_i}} = [x_1^{A_i}, x_2^{A_i}, \ldots, x_n^{A_i}]$ from an application $A_i \in \mathbf{A}$, the set of training applications, is processed to generate its corresponding embedding by leveraging the sequential attention-based method described in Section IV-C. Figure 2(b) illustrates this process for learning sample embeddings from application data.

Afterwards, as shown in Figure 2(c), xAMM concatenates the application sample embeddings, $e^{A_i}_{x_j}$ with the AMMs of the source (M_s) and target (M_t) machines, if available:

$$\mathbf{I}_j^{A_i} = \mathbf{e^{A_i}_{x_j}} \oplus \mathbf{AMM}_{M_s} \oplus \mathbf{AMM}_{M_t}, \quad \forall x_j \in A_i \tag{3}$$

Where $\mathbf{I}_j^{A_i}$ is the output of the j^{th} sample of application A_i, and $\oplus$ represents concatenation, $\mathbf{e}^{A_i}_{x_j}$ represents the embedding of sample x_j for application A_i, and $\mathbf{AMM}_{M_s}$ and $\mathbf{AMM}_{M_t}$ are the AMMs of the source machine M_s and target machine M_t, respectively.

As shown in Figure 1(a), if users do not provide a machine benchmark dataset during training, xAMM can either utilize a pre-trained AMM for a closely related platform (selected by the user) or skip this step entirely. Similarly, during inference (Figure 1(b)), if users want xAMM to leverage AMMs, they must supply the source/reference machine's name and the target machine's benchmark dataset if the target machine was not part of the training dataset. Otherwise, the target machine's name suffices. It is important to note that while the source machine must be included in the training dataset, the test application does not need to be seen previously.

IV-F Training a Downstream Predictive Model

The training phase builds a source performance model from multiple applications run on multiple machines using various configurations. For training a predictive model, xAMM employs a supervised learning approach using $\mathbf{I}_j^{A_i}$ from Equation 3 as features and $\mathbf{relR^{A_i}_{M_s \to M_t}}$ from Equation 1 as the target,

$$\hat{y} = h(\mathbf{I_j^{A_i}})\ \forall x_j \in A_i \text{ and } \forall i \in |A| \tag{4}$$

Where $|A|$ is the number of training applications in the set of $\mathbf{A}$, $\hat{y}$ is the predicted relative performance, and h is the trained model function. Once the source model is built, it can be reused for knowledge transfer across various machines. Future work will explore few-shot learning [24] methods to continue to update the model incrementally.

IV-G Using `xAMM` for Inference

Figure 1(b) outlines the steps involved during inference using `xAMM`, which automates the process with minimal user input via a `config.yml` file. To use `xAMM`'s inference service, users provide performance measurements of a new application on a reference machine (M_s) and specify the names of the source (M_s) and target (M_t) machines in the configuration file. If AMMs are to be used, and M_t was included during training, its pre-trained AMM is reused. If M_t was not included, users may provide its benchmark data, which `xAMM` converts into an AMM before processing application samples. Alternatively, users can skip AMM usage altogether, and `xAMM` processes the application samples directly.

During training, `xAMM` builds two models using AMMs and without. Both use attention-based sample embeddings, ensuring functionality even if AMMs are unavailable during inference. To add a new machine, users need to collect its benchmark data (e.g., using RAJAPerfSuite) and update the `config.yml` file. Users can select `func: amm` to construct AMMs separately from the train and test tasks, `func: train` to train a downstream predictive model, or `func: test` for inference. If `source_model_path` is unspecified, the training module is triggered automatically. List 1 shows an example configuration file with comments explaining its fields:

```
config.yml
  machine_data_path: # Path to machine benchmark
    dataset (only required if AMM is used)
  app_train_data_path: # Path to application
    training dataset
  app_test_data_path: # Path to application testing
    dataset
  src_model_path: # Path to pre-trained source model
  src_machine_amm: # Path to source machine's saved
    AMM. If omitted, then AMM is not used during
    inference
  target_machine_amm: # Path to target machine's
    benchmark dataset or saved AMM. If omitted, then
     AMM is not used during inference
  func: amm train test # Specify functionality:
    construct AMMs, train, or test
```

Listing 1: Example configuration file for `xAMM`

V. Experimental Setup

V-A Applications

We leverage five HPC proxy applications on three CPU and GPU platforms. Table I describes the applications `Laghos`, `Kripke`, `miniVite`, `SW4lite`, and `TESTDFFT`, the number of samples collected per application, and their significance in many science and engineering domains.

V-B Application Performance Samples

Table II describes the set of hardware performance counters used as features. These include ratios of branch, load, store, and arithmetic instructions to total instructions, cache miss rates at L1 and L2 levels, floating-point operation intensities, I/O activity, memory stalls, and extended page table sizes.

TABLE I: Description of the applications

Application	Description	#-Samples
`Laghos`	Finite Element Method (FEM) for compressible gas dynamics; represents many applications in computational fluid dynamics and material science under extreme conditions.	154
`Kripke`	3D Sn Deterministic Particle Transport Code; represents a key computational method in nuclear engineering, radiation shielding, and medical physics applications.	106
`miniVite`	Graph Community Detection; represents a key computational method in social network analysis, bioinformatics, and cybersecurity.	289
`SW4lite`	Seismic Wave Simulation; represents a computation fundamental to geophysics and earthquake engineering.	15
`TESTDFFT`	Parallel 3D FFT; represents fundamental computation common in many scientific and engineering applications, including signal processing, computational physics, and image processing.	85

TABLE II: Collected hardware performance counters

Feature	Description
Branch Intensity	Ratio of branch instructions to total instructions
Load Intensity	Ratio of load instructions to total instructions
Store Intensity	Ratio of store instructions to total instructions
L1 Load Misses	Load misses from L1 cache
L1 Store Misses	Store misses from L1 cache
L2 Load Misses	Load misses from L2 cache
L2 Store Misses	Store misses from L2 cache
Single Floating Point Intensity	Ratio of single precision FP instructions to total instructions
Double Floating Point Intensity	Ratio of double precision FP instructions to total instructions
Arithmetic Intensity	Ratio of integer arithmetic instructions to total instructions
I/O Bytes Read	Bytes read from IO
I/O Bytes Written	Bytes written to IO
Extended Page Table	Extended page table size
Memory Stall	Memory Stalls

A challenge in comparing hardware performance counters across platforms is that the names of these counters can vary. To address this challenge, we assign high-level names to features based on their intended use, mapping them to the corresponding counters on a different platform. More details on the data collection process can be found here [4].

V-C Embedding Model

To implement sequential attention for learning sample embeddings, we select the default deep neural network architecture available in the TabNet [25] library, although any library can be used. To run TabNet, we used PyTorch version 2.1.2 and Python version 3.10.0. We used default hyperparameter values implemented in TabNet. The embedding model produces one vector for each sample. We also compare our $f_{attention}$ method to the state-of-the-art embedding learning methods used in the ML domain: Neural Network (NN), Auto Encoder (AE), Variational Auto Encoder (VAE).

V-D Downstream Models

We evaluate four machine learning techniques as the downstream predictive model: `ADABOOST` [26], `BAGGING` [27], `RANDOM FORREST` [28], and `XGBOOST` [9]. The choice of a downstream predictive model is not the primary focus of this

TABLE III: Machine Specifications

Metric	M_1	M_2	M_3
Machine Name	Quartz	Ruby	Corona
CPU Type	Intel Xeon E5-2695 v4	Intel Xeon CLX-8276L	AMD Rome
Cores/node	36	56	48
GPU Support	No	No	Yes
GPU Type	N/A	N/A	AMD MI50
GPUs/node	N/A	N/A	8

work. This work aims to demonstrate that the improved data representation generated by `xAMM` is universally beneficial, regardless of the prediction model used.

V-E Hardware platforms

Although we benchmarked 16 machines, we collected samples from 5 applications on 3 platforms to keep data collection time manageable. More details of the datasets are described in Section III. Table III summarizes the specifications of these three machines (Quartz [29] as M_1, Ruby [30] as M_2, and Corona [31] as M_3). Quartz and Ruby are different generations of CPU-based machines without a GPU. Quartz features 36 cores per node and Ruby features 56 cores per node. In contrast, Corona is a GPU-based system with 48 cores per node and equipped with 8 AMD MI50 GPUs per node. We leverage RAJAPerfSuite to collect machine benchmark data.

The full list of 16 machines includes various processor architectures: Intel Xeon Gold, Intel Xeon E5-2695 v2, Intel Xeon E5-2695 v4, Intel Xeon CLX-8276L, AMD EPYC 7742, Intel(R) Xeon(R) Platinum 8479, Intel Xeon Platinum 8000 series (Skylake-SP), AMD EPYC 7R13, Intel Xeon 8375C, Intel Xeon E5-2695, IBM Power9, 121 nodes AMD Rome, AMD Trento, 96 custom Intel Xeon Scalable (Skylake) vCPUs, Intel Xeon Gold 6140, and Intel Cascade Lake 8275CL.

V-F Evaluation Metrics

We use Mean Absolute Percentage Error (MAPE) to quantify the average deviation of predictions of a downstream predictive model from actual values as a percentage. A lower MAPE indicates higher accuracy. For instance, a MAPE of 10% means the predictions deviate by 10% on average from the actual values. MAPE is mathematically calculated as:

$$\text{MAPE} = \frac{1}{n}\sum_{i=1}^{n}\left|\frac{y_i - \hat{y}_i}{y_i}\right| \times 100$$

Where y_i is the actual value, $\hat{y}_i$ is the predicted value, and n is the total number of predictions.

VI. Results

In this section, we evaluate the effectiveness of our proposed data transformation methodology. Specifically, we:

1) Evaluate the overall impact of our proposed data transformation method ($f_{\text{attention}} + \text{AMMs}$) on downstream models.
2) Evaluate the impact of the $f_{attention}$ method.
3) Evaluate the impact of AMMs.
4) Evaluate different options for constructing AMMs.

In Figures 3, 5, and 6, the X-axis represents different cross-platform prediction scenarios, while the Y-axis shows the average MAPE (lower is better) for four downstream models.

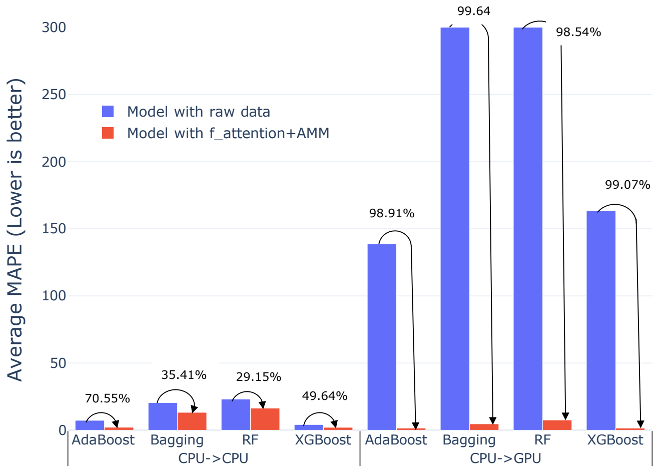


Fig. 3: Our data transformation method improves the MAPE for all models compared to their counterparts using raw data. XGBoost achieves an average MAPE reduction of 49.64% for CPU $\rightarrow$ CPU (predicting the performance on a CPU using that on another CPU) and 99.07% in CPU $\rightarrow$ GPU (predicting performance on a GPU using that on another CPU).

VI-A Exp 1: Evaluate the overall impact of our proposed data transformation method

The objective of this experiment is to evaluate the impact of our proposed data transformation method on downstream models' (Section V-D) prediction error, measured using MAPE. We run this experiment using a one-out cross-validation setup where in each of five iterations, one application from Table I is used as the test application, while the remaining applications are used to train the source model. Figure 3 presents the average MAPE across all five iterations for three cases: CPU$\rightarrow$CPU, CPU$\rightarrow$GPU, and GPU$\rightarrow$CPU. Here, $M_1 \rightarrow M_2$ represents the scenario of predicting the performance of an application on M_2 using its performance on M_1.

Observations In Figure 3, we observe that: (1) our proposed data transformation method, $f_{attention}$ + AMMs, improves the accuracy of `ADABOOST`, `BAGGING`, `RANDOM FORREST`, and `XGBOOST` by 84.7%, 67.5%, 63.85%, and 74.35%, respectively, compared to their counterparts using raw samples averaged across both CPU $\rightarrow$ CPU and CPU $\rightarrow$ GPU. (2) Attention to detail in sample embedding learning and the inclusion of AMMs reduce the prediction error of the state-of-the-art XGBoost model by 49.64% for CPU$\rightarrow$CPU predictions and by 99.07% for CPU$\rightarrow$GPU predictions. (3) All predictive models generalize better using our method.

VI-B Exp 2: Evaluate the impact of the $f_{attention}$ method.

This experiment evaluates the effectiveness of our proposed attention-based sample embedding learning method. An embedding learning method is effective if, using these embeddings, a clustering algorithm can discriminate between samples from different machines while maintaining similarity among samples from the same machine. For this experiment, we compare our attention-based embedding learning method with other state-of-the-art ones: NN, AE, VAE. The output of the

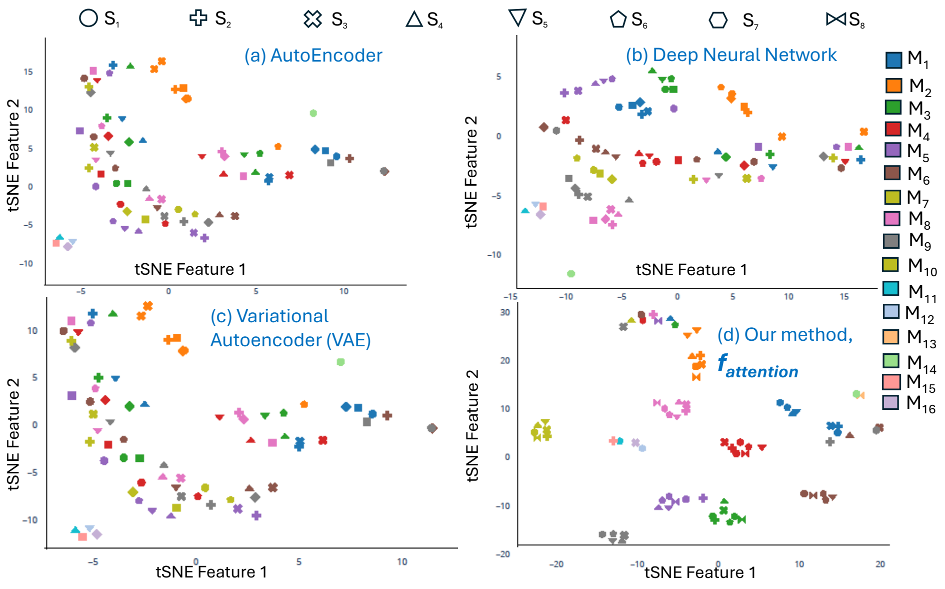
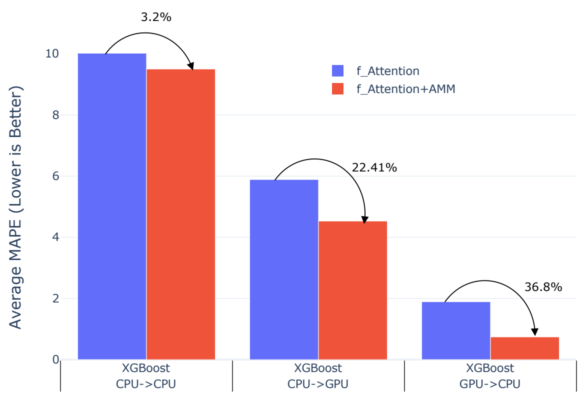


Fig. 4: t-SNE plots reveal that our attention-based sample embedding method creates distinct clusters for samples from different machines while tightly grouping samples from the same machine. Although not shown in this paper, further investigation shows that $f_{attention}$-based embeddings reduce XGBoost's MAPE by 20.3% compared to raw data, significantly outperforming VAE, which achieves only a 3.31% reduction.



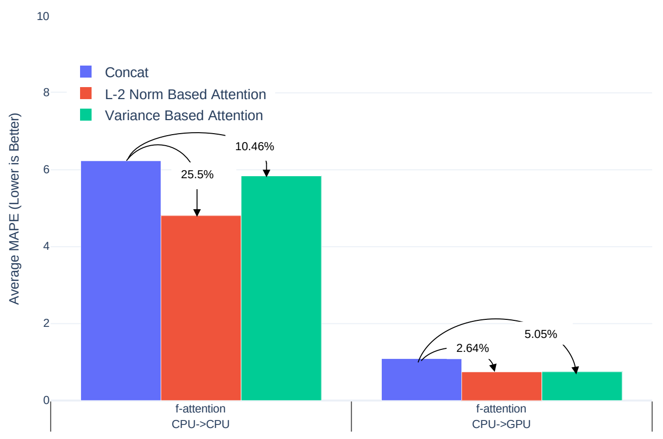


Fig. 5: Adding AMMs reduces XGBoost's average prediction error by 3.2%, 22.41%, and 36.8% for CPU $\rightarrow$ CPU (homogeneous), CPU $\rightarrow$ GPU (heterogeneous), and GPU $\rightarrow$ GPU (heterogeneous) scenarios, respectively.

Fig. 6: AMMs constructed using L2-norm and variance based methods (Option 2 and 3 in Algorithm 1) improve prediction accuracy over the Concat-based approach (Option 1). For CPU$\rightarrow$CPU, Option 2 reduces MAPE by 25.5% and Option 3 by 10.46%. For CPU$\rightarrow$GPU, Option 2 achieves a 2.64% reduction, while Option 3 achieves 5.05%.

`Sample Embedding Learning` module is fed directly to the clustering algorithm. Figure 4 visualizes sample embeddings using t-SNE [32] plots. t-SNE plots project the high-dimensional embeddings into a two-dimensional space.

Observations Figure 4d shows that our proposed attention-based sample embedding learning method ($f_{\text{attention}}$) generates well-separated clusters for samples from different machines while keeping samples from the same machine tightly grouped within distinct clusters. In contrast, embeddings generated by AE (Figure 4a), NN (Figure 4b), and VAE (Figure 4c) fail to capture this underlying structure, resulting in scattered and poorly defined clusters. To further study the impact of these embeddings, we trained XGBoost models to predict the relative runtimes of applications across CPU $\rightarrow$ CPU and CPU $\rightarrow$ GPU scenarios (not shown). Compared to training XGBoost models on raw data, using $f_{attention}$ embeddings reduced XGBoost's MAPE by 20.3%, while VAE embeddings achieved a reduction of 3.31%. This observation highlights the effectiveness of attention-based embeddings in improving the performance of downstream models.

VI-C Exp 3: Evaluate the impact of concatenating AMMs

The objective of this experiment is to evaluate the impact of AMMs on cross-platform prediction. We compare the effectiveness of XGBoost, since it is the best model in Figure 3, by providing two types of inputs: one using only the application sample embeddings generated by $f_{\text{attention}}$, and the other combining $f_{\text{attention}}$ embeddings with AMMs.

Observations In Figure 5, we observe that the addition of AMMs significantly improves the accuracy of the XGBoost model for all three scenarios. Adding AMMs reduces the average MAPE across one-out experiments in XGBoost by 3.2% for CPU→CPU predictions. The impact is significantly larger for heterogeneous platforms, with reductions of 22.4% for CPU→GPU and 36.8% for GPU→GPU compared to using only $f_{attention}$. This improvement arises because AMMs capture architectural feature relationships, such as memory hierarchies, providing XGBoost with structural correlations.

VI-D Exp 4: Evaluate options for constructing AMMs

This experiment evaluates three different methods, as presented in Algorithm 1, of constructing AMMs. To achieve this, we compare three aggregation methods outlined in Algorithm 1: averaging, and attention using L2-norm, and variance.

Observations From Figure 6, we observe that AMMs constructed using L2 norm and variance-based methods (Option 2 and Option 3 in Algorithm 1) improve prediction accuracy compared to the Concat-based approach (Option 1). For CPU→CPU predictions, L2 norm-based approach reduces MAPE by 25.5%, while variance-based one achieves a 10.46% reduction. Similarly, for CPU→GPU predictions, L2-norm reduces MAPE by 2.64%, and variance-based approach achieves a 5.05% reduction.

VI-E Discussions

Our results demonstrate that (1) the proposed attention-based embedding method improves downstream predictive accuracy, regardless of their architectures. Future work will evaluate the impact of our data transformation methodology on more advanced architectures such as Transformers. (2) embeddings generated by our proposed sequential-attention-based method cluster samples from the same machine closely while separating samples from different machines, as shown in t-SNE plots (Figure 4), providing key discriminative information for downstream models; and (3) the method generalizes across diverse applications (can be unseen during training) and architectures, provided the machines are included in the training set, highlighting its significance in improving the effectiveness of cross-platform performance prediction.

Our methodology is flexible and modular. While the combination of attention-based sample embeddings and AMMs yields the best results, even using the sample embedding learning method alone boosts predictive accuracy. For instance, XGBoost benefits significantly from using our attention-based embeddings. This adaptability allows our approach to complement many existing ML-based methods.

One limitation of our work is that to use AMMs during inference, the source model must be included during training. Although the dataset may appear small, consisting of 649 experiments from 5 applications across 3 platforms (Table I), collecting this data was time-intensive, reflecting the practical constraints of HPC environments. Despite this limitation, `xAMM` achieves a significant reduction in MAPE, demonstrating that even with a smaller dataset, our method enables downstream models to generalize across platforms, underscoring the success of our approach.

VII. Related Work

Several studies have explored cross-platform runtime prediction in HPC, focusing on diverse methodologies. Kumar et al. [10], Marathe et al. [5], and Wyatt et al. [11] develop transfer learning methods using various neural network-based architectures, predicting absolute execution time. Yang et al. [1] takes a different approach by partially executing applications and using window-based averaging to predict cross-platform performance, which contrasts with ML-driven methodologies. Gupta et al. [6] analyzed performance across platforms without leveraging embedding learning or machine abstractions like AMMs. Yokelson et al. [33] and Sun et al. [34] used linear regression and neural networks, focusing on runtime features, while Malakar et al. [7] benchmarked ML techniques without exploring embedding-based representations. Similarly, Chen et al. [35] applied ensemble learning for runtime prediction but did not incorporate embedding learning or AMMs.

In contrast, our work is complementary to all existing modeling approaches, as the proposed attention-based data transformation pipeline can be applied to improve input quality and predictive accuracy across any downstream model architecture. Moreover, even in the absence of AMMs, our sample embedding learning method alone improves predictive accuracy, as demonstrated in our experiments with XGBoost, underscoring the impact of our work.

While extensively used in other domains, attention mechanisms remain underexplored in HPC performance analytics. Previous methods such as AutoEncoders (AE), Variational AutoEncoders (VAE), and Neural Networks (NN) generate embeddings; however, these methods often fail to capture the nuanced relationships necessary for accurate cross-platform predictions. By incorporating attention into the embedding learning process, `xAMM` encodes critical performance characteristics that enable downstream models to better generalize.

For researchers without access to machine benchmark datasets to construct AMMs, our results show that the sample embedding learning method itself can still be used independently to improve downstream model performance. While our primary focus is on attention-based embeddings, we recognize the potential for more sophisticated downstream models, such as Transformer-based architectures. This provides a natural direction for future work and highlights the flexibility of our methodology in adapting to evolving ML paradigms.

VIII. Conclusions

This work addresses the problem of performance prediction across heterogeneous platforms. Traditional methods, which rely on static hardware descriptions and absolute runtimes, often fall short of capturing the dynamic behaviors of modern systems. To address this gap, we leverage relative performance, propose attention-based embeddings, and, to our knowledge, are the first to introduce a principled methodology for creating AMMs. Our evaluations show that incorporating the attention mechanism alongside machine-specific representations, significantly improves cross-platform prediction accuracy.

Acknowledgment

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-871860). This work was supported in part by LLNL LDRD project 24-SI-005, the National Science Foundation Grant No. 2443561 and 2047120, and Texas Advanced Computing Center (TACC) at The University of Texas at Austin.

References

[1] L. T. Yang, X. Ma, and F. Mueller, “Cross-platform performance prediction of parallel applications using partial execution,” in *SC’05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*. IEEE, 2005, pp. 40–40.

[2] L. Li, T. Flynn, and A. Hoisie, “Learning Generalizable Program and Architecture Representations for Performance Modeling,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, Nov. 2024.

[3] A. Dey, A. Dhakal, T. Z. Islam, J.-S. Yeom, T. Patki, D. Nichols, A. Movsesyan, and A. Bhatele, “Relative performance prediction using few-shot learning,” in *2024 IEEE 48th Annual Computers, Software, and Applications Conference (COMPSAC)*. IEEE, 2024, pp. 1764–1769.

[4] D. Nichols, A. Movsesyan, J. Yeom, A. Sarkar, D. Milroy, T. Patki, and A. Bhatele, “Predicting cross-architecture performance of parallel programs,” in *Proceedings of the IEEE International Parallel & Distributed Processing Symposium*, ser. IPDPS ’24. IEEE Computer Society, may 2024.

[5] A. Marathe, R. Anirudh, N. Jain, A. Bhatele, J. Thiagarajan, B. Kailkhura, J. Yeom, B. Rountree, and T. Gamblin, “Performance modeling under resource constraints using deep transfer learning,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2017, pp. 1–12.

[6] A. Gupta, P. Faraboschi, F. Gioachin, L. V. Kale, R. Kaufmann, B.-S. Lee, V. March, D. Milojicic, and C. H. Suen, “Evaluating and improving the performance and scheduling of hpc applications in cloud,” *IEEE Transactions on Cloud Computing*, vol. 4, no. 3, pp. 307–321, 2014.

[7] P. Malakar, P. Balaprakash, V. Vishwanath, V. Morozov, and K. Kumaran, “Benchmarking machine learning methods for performance modeling of scientific applications,” in *2018 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. IEEE, 2018, pp. 33–44.

[8] S. Kim, A. Sim, K. Wu, S. Byna, Y. Son, and H. Eom, “Towards hpc i/o performance prediction through large-scale log analysis,” in *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing*, 2020, pp. 77–88.

[9] T. Chen, T. He, M. Benesty, V. Khotilovich, Y. Tang, H. Cho, K. Chen *et al.*, “Xgboost: extreme gradient boosting,” *R package version 0.4-2*, vol. 1, no. 4, pp. 1–4, 2015.

[10] R. Kumar, A. Mankodi, A. Bhatt, B. Chaudhury, and A. Amrutiya, “Cross-platform performance prediction with transfer learning using machine learning,” in *2020 11th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*. IEEE, 2020, pp. 1–7.

[11] M. R. Wyatt, S. Herbein, T. Gamblin, A. Moody, D. H. Ahn, and M. Taufer, “Prionn: Predicting runtime and io using neural networks,” in *Proceedings of the 47th International Conference on Parallel Processing*, 2018, pp. 1–12.

[12] O. Pearce, J. Burmark, R. Hornung, B. Bogale, I. Lumsden, M. McKinsey, D. Yokelson, D. Boehme, S. Brink, M. Taufer *et al.*, “Raja performance suite: Performance portability analysis with caliper and thicket,” in *SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2024, pp. 1206–1218.

[13] “Raja performance suite.” [Online]. Available: https://github.com/LLNL/RAJAPerf

[14] W. Gu, A. Tandon, Y.-Y. Ahn, and F. Radicchi, “Principled approach to the selection of the embedding dimension of networks,” *Nature Communications*, vol. 12, no. 1, p. 3772, 2021.

[15] N. Pinnow, T. Ramadan, T. Z. Islam, C. Phelps, and J. J. Thiagarajan, “Comparative code structure analysis using deep learning for performance prediction,” 2021.

[16] D. A. Beckingsale, J. Burmark, R. Hornung, H. Jones, W. Killian, A. J. Kunen, O. Pearce, P. Robinson, B. S. Ryujin, and T. R. Scogland, “RAJA: portable performance for large-scale scientific applications,” in *2019 ieee/acm international workshop on performance, portability and productivity in hpc (p3hpc)*. IEEE, 2019, pp. 71–81.

[17] P. J. Mucci, S. Browne, C. Deane, and G. Ho, “Papi: A portable interface to hardware performance counters,” in *Proceedings of the department of defense HPCMP users group conference*, vol. 710, 1999.

[18] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent, “Hpctoolkit: Tools for performance analysis of optimized parallel programs http://hpctoolkit.org,” *Concurr. Comput. : Pract. Exper.*, vol. 22, no. 6, pp. 685–701, Apr. 2010. [Online]. Available: http://dx.doi.org/10.1002/cpe.v22:6

[19] T. Ramadan, A. Lahiry, and T. Z. Islam, “Novel representation learning technique using graphs for performance analytics,” in *2023 International Conference on Machine Learning and Applications (ICMLA)*. IEEE, 2023, pp. 1311–1318.

[20] D. Zhang, J. Yin, X. Zhu, and C. Zhang, “Network representation learning: A survey,” *IEEE transactions on Big Data*, vol. 6, no. 1, pp. 3–28, 2018.

[21] M. Tschannen, O. Bachem, and M. Lucic, “Recent advances in autoencoder-based representation learning,” *arXiv preprint arXiv:1812.05069*, 2018.

[22] M. Zhang, T. Z. Xiao, B. Paige, and D. Barber, “Improving vae-based representation learning,” *arXiv preprint arXiv:2205.14539*, 2022.

[23] M. Dippel, A. Kiezun, T. Mehta, R. Sundaram, S. Thirumalai, and A. Varma, “Attention improves concentration when learning node embeddings,” *arXiv preprint arXiv:2006.06834*, 2020.

[24] Y. Wang, Q. Yao, J. T. Kwok, and L. M. Ni, “Generalizing from a few examples: A survey on few-shot learning,” *ACM computing surveys (csur)*, vol. 53, no. 3, pp. 1–34, 2020.

[25] S. O. Arik and T. Pfister, “Tabnet: Attentive interpretable tabular learning. arxiv 2019,” *arXiv preprint arXiv:1908.07442*, 1908.

[26] L. Gao, P. Kou, F. Gao, and X. Guan, “Adaboost regression algorithm based on classification-type loss,” in *2010 8th World Congress on Intelligent Control and Automation*. IEEE, 2010, pp. 682–687.

[27] Q. Sun and B. Pfahringer, “Bagging ensemble selection for regression,” in *Australasian Joint Conference on Artificial Intelligence*. Springer, 2012, pp. 695–706.

[28] M. R. Segal, “Machine learning benchmarks and random forest regression,” 2004.

[29] Lawrence Livermore National Laboratory, “Quartz compute system,” 2025. [Online]. Available: https://hpc.llnl.gov/hardware/compute-platforms/quartz-decommissioned

[30] ——, “Ruby computing system,” 2025. [Online]. Available: https://hpc.llnl.gov/hardware/compute-platforms/ruby

[31] ——, “Corona computing system,” 2025. [Online]. Available: https://hpc.llnl.gov/hardware/compute-platforms/corona

[32] B. Kang, D. Garcia Garcia, J. Lijffijt, R. Santos-Rodríguez, and T. De Bie, “Conditional t-sne: more informative t-sne embeddings,” *Machine Learning*, vol. 110, pp. 2905–2940, 2021.

[33] D. Yokelson, M. R. J. Charest, and Y. W. Li, “Hpc application performance prediction with machine learning on new architectures,” in *Proceedings of the 2023 on Performance EngineeRing, Modelling, Analysis, and VisualizatiOn Strategy*, ser. PERMAVOST ’23. New York, NY, USA: Association for Computing Machinery, 2023, p. 1–8. [Online]. Available: https://doi.org/10.1145/3588993.3597262

[34] J. Sun, G. Sun, S. Zhan, J. Zhang, and Y. Chen, “Automated performance modeling of hpc applications using machine learning,” *IEEE Transactions on Computers*, vol. 69, no. 5, pp. 749–763, 2020.

[35] X. Chen, H. Zhang, H. Bai, C. Yang, X. Zhao, and B. Li, “Runtime prediction of high-performance computing jobs based on ensemble learning,” in *Proceedings of the 2020 4th International Conference on High Performance Compilation, Computing and Communications*, 2020, pp. 56–62.