High-Dimensional Performance Modeling via Tensor Completion

Edward Hutter
Department of Computer Science
University of Illinois at Urbana-Champaign
hutter2@illinois.edu

Edgar Solomonik
Department of Computer Science
University of Illinois at Urbana-Champaign
solomon2@illinois.edu

Abstract—Performance tuning, software/hardware co-design, and job scheduling are among the many tasks that rely on models to predict application performance. We propose and evaluate low rank tensor decomposition for modeling application performance. We use tensors to represent regular grids that discretize the input and configuration domain of an application. Application execution times mapped within grid-cells are averaged and represented by tensor elements. We show that low-rank canonical-polyadic (CP) tensor decomposition is effective in approximating these tensors. We then employ tensor completion to optimize a CP decomposition given a sparse set of observed runtimes. We consider alternative piecewise/grid-based (P/G) and supervised learning models for six applications and demonstrate that P/G models are significantly more accurate relative to model size. Among P/G models, CP decomposition of regular grids (CPR) offers higher accuracy and memory-efficiency, faster optimization, and superior extensibility via user-selected loss functions and domain partitioning. CPR models achieve a 2.18x geometric mean decrease in mean prediction error relative to the most accurate alternative models of size ≤10 kilobytes.

Index Terms—performance modeling, tensor completion

I. INTRODUCTION

Application performance depends on both configuration parameters (e.g., block sizes, processor grid topology) and architectural parameters (e.g., processes-per-node, hyper-thread count) for a given set of inputs. Complex interactions among these parameters [1], [2] motivate observation of an application’s full parameter space, the size of which has increased to account for growth in algorithmic complexity and architectural diversity. Tasks such as program optimization [3]–[5], optimal tuning parameter selection [2], [6], architectural design [7], [8], dynamic load balancing, and machine allocation estimation (e.g., for weather forecasting and simulating virus spread dynamics) increasingly rely upon analytic and regression models for accurate and efficient performance prediction.

The demand for increasingly accurate performance prediction is reflected in the abundance of available modeling frameworks [9]–[12]. Application performance models, which we survey in Section II, are typically either (1) global (non-piecewise) models, configured semi-analytically or automatically (such as via multivariate adaptive regression splines (MARS) [13]), (2) piecewise/grid-based models, which discretize the modeling domain, or (3) alternative models configured by supervised learning methods (such as neural networks). We instead consider tensor decompositions, which generalize low-rank matrix factorizations to higher dimensions [14] and are effective models for tensor completion [15], the problem of approximating a tensor given a subset of observed entries. Specifically, we employ tensor completion based on low-rank CP decomposition [16] for application performance modeling, and compare this model, which belongs to type (2), to representative models of each type.

CP decompositions model an application configuration \( \mathbf{x} = (x_1, \ldots, x_d) \)’s performance \( f(\mathbf{x}) \) as a linear combination of separable piecewise functions \( m(\mathbf{x}) = \sum_{r=1}^{R} \prod_{j=1}^{d} g_{r,j}(x_j) \). CP decompositions thus have the advantage of generalizing simple multilinear cost models. These models also provide systematic improvability (via increased rank) and achieve linear model size with tensor order (number of application parameters) for a fixed rank. We provide specific details on how we discretize an application’s parameter space to build a low-rank CP decomposition model in Section IV.

We assess model predictions using a scale-independent error metric \( \log(m(\mathbf{x})/f(\mathbf{x})) \), for which it is useful to consider alternative loss functions and data transformations. Algorithms for computing CP decompositions, which we summarize in Section III, typically minimize the Frobenius norm of absolute errors, yet others can minimize alternative metrics [17] (e.g., mean relative error). We pursue scale-independent minimization of prediction error, and find that CP decompositions optimized using least-squares loss functions with logarithmic transformation of execution times achieve superior accuracy and optimization efficiency. We characterize error metrics for assessing performance model accuracy in Section II-A.

The most similar technique to CP decomposition studied in prior work for application performance modeling is sparse grid regression (SGR) [18], [19]. SGR uses a hierarchical reduced-size model of a regular grid containing performance measurements. Both SGR and CP decomposition can compress performance across discrete modeling domains with high accuracy. Our experimental results in Section VI show that CP decomposition of a regular grid (CPR) offers improved accuracy and scalability in high-dimensional domains.

We use experimental analysis to demonstrate the efficacy of CP decomposition for application performance modeling. We generate datasets for computational kernels, communication routines, and scientific applications (which invoke these kernels/routes) by executing randomly-selected configurations,
which we use to train a collection of piecewise/grid-based and alternative supervised learning models. We assess the generated models by the accuracy and efficiency with which they predict unobserved performance and compress application performance, respectively. Low-rank CP decompositions are consistently more accurate relative to model size, and improve mean prediction accuracy by a geometric mean of 2.18x across six kernels/applications for models of size ≤10 kilobytes.

Our novel contributions towards performance modeling are as follows:

- a methodology for learning piecewise multilinear models via application of CP tensor decomposition (CPR),
- a software library for CPR that leverages open-source high-performance software for tensor completion,¹
- an evaluation of piecewise/grid-based models and alternative supervised learning methods using six applications,
- empirical evidence that CP decompositions achieve highest prediction accuracy among existing models relative to model size.

II. MULTI-PARAMETER PERFORMANCE MODELING

Application performance models predict execution times given a configuration of application benchmark parameters. The structure of these models reflects application characteristics (e.g., execution requirements, scaling behavior) identified analytically and/or learned using measurements of executed configurations. We review methods to construct these models and summarize quantitative techniques to assess them.

A. Model Assessment

Performance models are assessed by size, trainability (e.g., model optimization costs), and the accuracy with which they predict application performance. Given a particular configuration of application parameters \( (x_1, \ldots, x_d) \equiv x \), measured execution time \( y \equiv f(x) \), and a model output \( m \equiv m(x) \), relative prediction error is given by \( |m - y|/y \). Relative error is more appropriate than absolute error for assessing performance predictions spanning multiple orders of magnitude (e.g., kernel evaluation over a range of inputs), yet does not assign similar errors to mispredictions of a similar scale (i.e., \( \log(m/y) \)). For example, this metric will identify \( y/n: 0 < n \leq 0.5 \) as a more accurate prediction of \( y = 1 \) than model output \( m = 2 \).

Model selection via minimization of relative error thus exhibits bias towards under-prediction which can misinform tasks that leverage model output e.g., construction of parallel schedules.

To assess performance prediction accuracy, we adopt error metrics that assign equal error to under/over-predictions of the same scale (i.e., scale independence). In particular, we penalize model outputs \( m_k = ay_k \) and \( m_k = y_k/a \) equally for positive factor \( a \). Consider a collection of measured configurations \( \{(x_k, y_k)\}_{k=1}^M \) and model outputs \( \{(x_k, m_k)\}_{k=1}^M \). Among the aggregate error metrics listed in Table I, only the arithmetic means of the absolute log and log-squared accuracy ratios (MLogQ and MLogQ2, respectively) achieve scale independence [20], [21]. This table also demonstrates the equivalence of the symmetric mean and log-transformed geometric-mean relative errors (SMAPE and LGMAPE, respectively), and MLogQ to first-order Taylor approximation for small relative errors \( \epsilon_k \) defined as \( m_k = y_k (1 + \epsilon_k) \). We configure models to minimize MLogQ in Section V, and describe a simple alternative loss function that achieves this in Section IV-B.

B. Model Formulation

Analytic models decompose an application’s execution time into that of the computational and communication kernels executed along its execution call paths [4], [10], [22]–[25]. Regression methods fit models to kernel benchmarking data or kernel execution costs (i.e., requirements models) [4] to instantiate analytic models of full applications. This composition of models facilitates incremental performance tuning and enables extrapolation from current to next-generation hardware [4]. Many open-source cross-platform profiling frameworks (e.g., HPC-ToolKit, Scalasca, TAU) facilitate model generation by extracting statistical profiles of invoked kernels. However, domain expertise and static source code analysis alone have been shown to achieve optimal configuration selection for both non-distributed kernels [26]–[28] and distributed kernels [29], [30]. We evaluate regression methods for modeling application performance that do not rely on program analysis.

Clustering and correlation analysis guide the selection of application benchmark parameters. These statistical techniques identify both redundant parameters and those which correlate strongest with performance [7], [8]. The modeling domain (i.e., parameter space) is then the tensor product of the enumerated ranges of chosen parameters. The enumerated range of each numerical parameter (e.g., block size) is typically transformed (e.g., via logarithm) or normalized to enforce similar scales across parameters. Nominal and ordinal categorical parameter values (e.g., solver type) are mapped onto a range of real numbers with a specified spacing (e.g., uniform) to supply a distance metric necessary for model optimization.

Domain expertise and/or additional offline statistical analysis guides the selection of an appropriate class of models. Global (non-piecewise) functions model performance behaviors present across the full domain, while piecewise/grid-based models and supervised learning methods are well-suited for modeling more complex behavior [31]. Kernel methods, neural

| Metric | Mathematical Expression | Error-centric Expression |
|--------|-------------------------|-------------------------|
| MAPE   | \( \sum_{k=1}^M \frac{|y_k - y_k|}{y_k} \) | \( \sum_{k=1}^M |\epsilon_k| \) |
| MAE    | \( \sum_{k=1}^M \frac{|m_k - y_k|}{y_k} \) | \( \sum_{k=1}^M |y_k - y_k| \) |
| MSE    | \( \sum_{k=1}^M (m_k - y_k)^2 \) | \( \sum_{k=1}^M (y_k - y_k)^2 \) |
| SMAPE  | \( 2 \sum_{k=1}^M \frac{|m_k - y_k|}{y_k} \) | \( 2 \sum_{k=1}^M |\epsilon_k| \) |
| LGMAPE | \( \sum_{k=1}^M \log(|y_k|/m_k) \) | \( \sum_{k=1}^M \log(|\epsilon_k|) \) |
| MLogQ  | \( \sum_{k=1}^M \log^2(|m_k/y_k|) \) | \( \sum_{k=1}^M \frac{1}{2} |\epsilon_k|^2 + O(\epsilon_k^2) \) |
| MLogQ2 | \( \sum_{k=1}^M \log^2((m_k/y_k)^2) \) | \( \sum_{k=1}^M (\frac{1}{2} |\epsilon_k|^2 + O(\epsilon_k^2)) \) |

¹https://github.com/huttered40/cpr-perf-model
networks, and recursive partitioning methods configure models by exploiting dependencies among benchmark parameters, while instance-based methods configure models on-the-fly to exploit distance between configurations. We summarize and evaluate representative models of each class.

Following the selection of class-specific hyper-parameters, a candidate model is configured by minimizing a particular aggregate error metric on measured execution times. The mean squared error (MSE), which is equivalent to the Frobenius norm of absolute errors and thus exhibits bias towards over-prediction, is typically used as a loss function to select among candidate models. This least-squares regression assumes noisy execution data is sampled from a normal distribution and identifies the maximum-likelihood candidate, while alternative loss functions are considered for alternative underlying noise models. As stated in Section II-A, the desired assessment metric should also be considered when selecting a loss function.

C. Model Class

1) Global: Relationships between application benchmark parameters and execution data are expressed as linear combinations of explanatory variables (i.e., non-piecewise functions parameterized on a subset of benchmark parameters), each of which has global support across the parameter space. These variables model interaction effects and nonlinear scaling behavior among subsets of parameters. Ordinary least-squares regression (OLS) is a convex quadratic optimization problem that estimates unknown parameters expressed linearly with respect to explanatory variables using MSE loss (i.e., linear regression). Given a suitable collection of variables, the Gauss-Markov theorem guarantees that OLS will attain the Gauss-Markov theorem guarantees that OLS will attain the minimum-variance linear unbiased estimate (MVE) of the true regression parameters. However, OLS assumes that the error terms are normally distributed and that the relationship between the response variable and the predictors is linear. In practice, these assumptions are often violated, and alternative regression techniques may provide more accurate predictions.

2) Non-piecewise/Grid: Non-piecewise explanatory variables are insufficient for modeling complex performance behavior with high accuracy. Complex dependence of performance on input and configuration parameters arises even in simple programs due to e.g., memory misalignment, register spilling, and load imbalance, which motivates both empirical search and model-driven search as general strategies for optimal parameter selection [1]. This behavior can be more accurately modeled as a linear combination of spline functions, i.e., tensor products of univariate functions defined piecewise by polynomials. These models introduce additional hyper-parameters, notably the selection of grid-points that discretize the modeling domain. Therefore, it is common to formulate global models, and only segment those parameters whose values correlate strongest with performance [7], [8].

Evaluation of performance on \( d \)-dimensional regular grids forgoes regression by modeling performance as a weighted average of \( 2^d \) neighboring grid-points, each of which stores the measured performance of a distinct configuration. For example, bilinear interpolation yields a prediction of a configuration \((x_1, x_2)\)'s execution time as

\[
t_{a,b}\left(\frac{1}{|x_1 - a|} - \frac{1}{x_{i+1} - x_i}\right) \left(\frac{1}{|x_2 - b|} - \frac{1}{x_{i+1} - x_i}\right),
\]

where \( x_{ij} \leq x_j < x_{ij+1}, \forall j \in \{1,2\} \) and \( t_{a,b} \) denotes the execution time of grid-point \((a,b)\) on a two-dimensional regular grid. Multilinear interpolation is suitable if performance behavior within grid-cells is sufficiently smooth. However, as observed in Section VI, this is rarely achieved without a significantly large number of observations.

Multiple frameworks, including multivariate adaptive regression splines (MARS) [13] and sparse grid regression [18], [32], compress regular grids by selecting a subset of grid-points and corresponding spline functions automatically. MARS recursively constructs products of univariate hinge functions \( b(x_i) = \{c, \max(0, x_i - c)\} \), each characterized by a parameter \( x_i \), and a scalar position \( c \) within its range. The selection of these parameters involves repeatedly searching across the parameter space and invoking OLS as candidate models are constructed and pruned. MARS therefore relies on various heuristics to accelerate model construction in high-dimensional settings. As we show in Section VI, MARS performance models [33] can achieve only a coarse discretization of the modeling domain and often produce global models.

Sparse grid regression (SGR) instead discretizes high-dimensional functions on anisotropic sparse grids. Sparse grids [34]–[36] are characterized by a hierarchical representation of piecewise-linear functions, the tensor-product of which yields piecewise \( d \)-linear basis functions. Only functions with sufficiently-large support across the modeling domain are retained. Sparse grids initially utilize \( O(2^n d \epsilon)^{d-1} \) grid-points, where \( n \) is a user-specified discretization level. SGR models performance as a linear combination of \( O(2^n d \epsilon)^{d-1} \) multi-scale basis functions [19] and offers automated spatially-adaptive grid refinement within the support of a specified number of basis functions [19]. Our methodology circumvents grid refinement and compresses regular grids, which feature \( O(2^{nd}) \) total grid-points, using rank-\( R \) tensor decompositions to \( O(2^n d R) \) size. We evaluate both methods in Section VI.

3) Kernel-based: Kernel methods project the modeling domain onto a higher-dimensional space characterized by user-specified kernel functions (e.g., sigmoid). Support vector machines (SVM) are global models formulated by partitioning the modeling domain so as to minimize a weighted distance between model predictions and observed performance. Loss functions for SVM regression typically use MSE and are characterized by a threshold parameter \( \epsilon \) such that prediction errors
smaller than $\epsilon$ are not penalized. Gaussian process regression (GPR) assumes that observations are samples of a multivariate normal distribution. The distributions are characterized by a mean vector and a positive-definite covariance matrix, the hyper-parameters of which are tuned by maximizing the log-marginal-likelihood using observations. We evaluate SVM kernel functions and GPR covariance kernels in this work.

4) Neural Networks: Multi-layer perceptrons model execution time as a composition of nonlinear activation functions [8], [37]. These models are constructed as stacks of layers, each of which consists of an array of units characterized by weights and an activation function. The output of each unit is used as an input to all units in the subsequent layer (i.e., fully-connected). Each activation function takes as input an inner product of a weight vector and those outputs. Optimization methods (e.g., back-propagation) determine weights by minimizing a loss function using observed execution times. Neural networks have been shown to be effective at extrapolating small-scale performance [31], [38]. These models feature a large design space, including layer size and number of layers, each of which we tune in our evaluation.

5) Recursive Partitioning: Decision trees partition the modeling domain into hyper-rectangles, each of which is assigned a constant value that models the performance of configurations mapped within its sub-domain. These hyper-rectangles are constructed by recursively splitting a parameter’s range to minimize a loss function (e.g., MSE) on a subset of observations. Random forest regression (RTR) models performance as an average of these constants, each of which corresponds to a hyper-rectangle defined by a distinct decision tree. Each tree is constructed using a random sample of observations via bootstrap aggregation, while each tree’s splits are constructed by minimizing a loss on a random sample of a subset of observations. Extremely-randomized tree regression (ETR) computes splits randomly as opposed to minimizing a specified error metric. Gradient-boosting regression (GBR) constructs trees sequentially by fitting residuals attained by the previously-constructed sequence of trees, which are proportional to the negative gradients of the MSE loss. In our experimental evaluation, we tune over a subset of hyper-parameters shared by each method, including forest size and tree-depth.

6) Instance-based: Instance-based methods construct models on-the-fly given unobserved configurations. We evaluate the $k$-nearest neighbors method, which predicts execution time by interpolating the execution times of the $k$ nearest observed configurations relative to a specified distance metric.

III. TENSOR COMPLETION

Tensor completion is the task of building a model to approximate a tensor based on a subset of observed entries [15]. These models are typically low-rank tensor decompositions. We review canonical-polyadic decomposition models and optimization methods for tensor completion in this section.

A. Low-rank tensor factorization models

A tensor $\mathcal{T} \in \mathbb{R}^{I_1 \times \cdots \times I_d}$ has order $d$ (i.e., $d$ modes/indices), dimensions $I_1$-by-$\cdots$-by-$I_d$, and elements $t_{i_1,\ldots,i_d} = t_i$ where $i \in \Omega_i = \{1, \ldots, I_i\}$. Tensors $\mathcal{T}$ exhibiting low-rank structure may be modeled as a summation of tensor products (i.e., canonical-polyadic decomposition [16]) or using other tensor decompositions such as Tucker [14]. We leverage canonical-polyadic decomposition, as it is relatively inexpensive to train, and leave exploration of other decompositions to future work.

The canonical-polyadic (CP) decomposition of an order three tensor $\mathcal{T} \in \mathbb{R}^{I_1 \times I_2 \times I_3}$ has the form $t_{i_1,i_2,i_3} = \sum_{r=1}^{R} u_{i_1,r} v_{i_2,r} w_{i_3,r} \equiv m_{i_1,i_2,i_3}$, where $R$ denotes the rank of the decomposition, and $U \in \mathbb{R}^{I_1 \times R}$, $V \in \mathbb{R}^{I_2 \times R}$, $W \in \mathbb{R}^{I_3 \times R}$ are factor matrices. The set of observed entries of an order-3 tensor $\mathcal{T}$ is represented by an index set $\Omega \subseteq \{1, \ldots, I_1\} \times \{1, \ldots, I_2\} \times \{1, \ldots, I_3\}$ so that every configuration $i \equiv (i_1, i_2, i_3) \in \Omega$ has an associated measurement $t_i \equiv t_{i_1,i_2,i_3}$. The size of these models grows linearly with tensor order for fixed dimensions and fixed rank, and linearly with CP rank for a fixed order and fixed dimensions. CP decompositions generalize the class of global performance models in Equation 1 as multilinear cost functions $m(x) = \sum_{r=1}^{R} u_r(x_1) v_r(x_2) w_r(x_3)$ composed of piecewise-linear functions $u_r, v_r, w_r, \forall r \in 1, \ldots, R$.

B. Optimization methods

CP decompositions must accurately represent observed entries, yet generalize effectively to unobserved entries. Optimizing these models involves minimizing an objective function that consists of a convex loss function and regularization terms in each factor matrix. Regularization decreases model parameter variance and thereby avoids over-fitting to observed entries within $\Omega$. Choice of element-wise loss function $\phi$ reflects both the distribution of elements within $\Omega$ and the selected error metric. Given a training set $\Omega$, CP rank $R$, and regularization parameter $\lambda$, tensor optimization minimizes CP decompositions of order-3 tensors by minimizing the objective function $f(U, V, W) = \lambda (\|U\|_F^2 + \|V\|_F^2 + \|W\|_F^2) + \sum_{(i_1, i_2, i_3) \in \Omega} \phi(t_{i_1,i_2,i_3}, m_{i_1,i_2,i_3})$.

(2)

We next summarize numerical methods that solve these non-linear optimization problems for various loss functions.

1) Least-squares loss: Algorithms for tensor completion typically fit CP decompositions to partially-observed tensors by minimizing the least-squares loss function $\phi(t_{i_1,i_2,i_3}, m_{i_1,i_2,i_3}) = (t_{i_1,i_2,i_3} - m_{i_1,i_2,i_3})^2$. Both quadratic optimization and gradient descent have been applied to minimize Equation 2 with this loss function. The alternating least-squares method (ALS) sweeps over the rows of each factor matrix, fixing the elements of all but one and minimizing, over the $i$’th row of $U$, the objective $f(u_{i_1}) = \frac{1}{|\Omega_i|} \sum_{(i_2,i_3) \in \Omega_i} (t_{i_1,i_2,i_3} - m_{i_1,i_2,i_3})^2 + \lambda \|u_{i_1}\|_2^2$, where $(i_2, i_3) \in \Omega_{i_2}$ if and only if $(i_1, i_2, i_3) \in \Omega$. Minimization of each row-wise convex objective involves solving a
linear least-squares problem. The total arithmetic complexity of ALS is $O((\sum_{j=1}^r f_j) R^3 + [\Omega] d^2 R^2)$ [39]. This overhead may be circumvented by optimizing factor matrix elements individually. In particular, the coordinate descent method (CD) minimizes the scalar objective $f(u_i, \cdot)$ and thereby reduces the arithmetic complexity of ALS by a factor of $R$. Both ALS and CD achieve monotonic convergence, yet CD often exhibits slower convergence due to its decoupling of row-wise factor matrix updates. Stochastic gradient descent instead iteratively updates all factor matrix elements at once and calculates the partial derivative of the corresponding objective using a random subset of observations within $\Omega$. The performance of these methods has been extensively studied for both matrix [40]–[46] and tensor completion [39], [47], [48].

2) Generalized loss functions: Tensor completion may fit CP decompositions to partially-observed tensors by minimizing alternative loss functions [17]. These loss functions often indirectly apply constraints to elements within factor matrices (e.g., non-negativity), which are enforced by incorporating additional terms (e.g., penalty and barrier functions) into the objective. Alternating minimization and coordinate minimization may apply Newton’s method to optimize row-wise subproblems, respectively. In particular, alternating minimization via Newton’s method (AMN) updates initial row-vector iterates $u_{i}^{(l)}$ as follows until convergence:

$$u_{i}^{(l+1)} \leftarrow u_{i}^{(l)} - H_f^{-1}(u_{i}^{(l)}) \nabla f(u_{i}^{(l)}),$$

where $\nabla f(u_{i}^{(l)}) = \sum_{d \in \Omega_i} \nabla \phi(u_{i}^{(l)}) + 2\lambda u_{i}^{(l)}, H_f(u_{i}^{(l)}) = \sum_{d \in \Omega_i} H_{\phi}(u_{i}^{(l)}) + 2\lambda I,$ and $H_f, H_{\phi}$ are Hessian matrices. Gradient-based methods such as stochastic gradient descent and L-BFGS have also been proposed [17], as have Gauss-Newton and quasi-Newton methods that also involve optimizing all factor matrices with each iteration [39]. We apply generalized CP decomposition via AMN to minimize alternative error metrics (see Section II-A). Specifically, we leverage existing work [39] and the corresponding parallelization strategies therein, with minor modifications to account for strictly positive observations.

IV. HIGH-DIMENSION PERFORMANCE MODELING VIA TENSOR COMPLETION

We apply tensor completion to optimize low-rank canonical-polyadic (CP) tensor decompositions of application performance. Our modeling framework enables user-specified partitioning of the modeling domain and provides a collection of loss functions to minimize user-specified error metrics. In this section, we describe model training and inference for accurate and scalable high-dimensional performance prediction.

A. Tensor model formulation

We map execution time measurements for each configuration of benchmark parameters to a tensor and use this tensor to predict the execution time of a given configuration. Each parameter corresponds to a distinct dimension (mode) of the tensor. For parameters $j$ with numerical values (e.g., problem size, block size), we discretize its range into $I$ sub-intervals $X^{(j)}_0, X^{(j)}_1, \ldots, X^{(j)}_I$ using uniform or logarithmic spacing. In the case of categorical parameters, we simply index the choices along the corresponding tensor dimension. We then define a grid/tensor whose grid-points/entries are associated with the tensor product of midpoints of each parameter sub-interval.

For $d$-parameter configurations, tensor element $t_{i_1, \ldots, i_d}$ is associated with the midpoints of a tensor product of sub-intervals $(X^{(1)}_{i_1}, X^{(1)}_{i_1+1}), \ldots, (X^{(d)}_{i_d}, X^{(d)}_{i_d+1})$. These bounding values define a grid-cell $C_{i_1, \ldots, i_d}$ with midpoint $(M^{(1)}_{i_1}, \ldots, M^{(d)}_{i_d})$, which we associate with element $t_{i_1, \ldots, i_d}$. If the $j$’th parameter is discretized with logarithmic spacing, the midpoint is associated with parameter value $M^{(j)}_{i_j} = \lfloor e^{(\log(X^{(j)}_{i_j}) + \log(X^{(j)}_{i_j+1}))/2}\rfloor$. Given a set of executed configurations, $t_{i_1, \ldots, i_d}$ stores the mean execution time among executed configurations mapped within cell $C_{i_1, \ldots, i_d}$. We leave evaluation of alternative quadrature schemes for future work.

Estimates are available for all mid-point configurations following completion of the tensor. We therefore use interpolation to predict execution times for arbitrary configurations. Given a particular configuration $x \equiv (x_1, \ldots, x_d)$, let $t_i = t_{i_1, \ldots, i_d}$ be the tensor element associated with the grid-point $M_i \equiv (M^{(1)}_{i_1}, \ldots, M^{(d)}_{i_d})$ just below $x$, so each $x_j \in [M^{(j)}_{i_j}, M^{(j)}_{i_j+1})$. Then, using $h_j(x) = x$ for uniformly and $h_j(x) = \log(x)$ for logarithmically discretized parameter $j$, linear interpolation of
neighboring grid-points gives the execution time prediction,
\[
\sum_{a \in \{0,1\}^d} t_{i+a} \cdot \prod_{j=1}^d \left( 1 - \frac{|h_j(x_j) - h_j(M_{i+a}^j)|}{h_j(M_{i+a}^j)} - h_j(M_{i+a}^j) - h_j(M_{i+e}^j) \right).
\]
Additional tensor elements are placed along the boundary of the modeling domain and store the estimated execution time of the closest mid-point. The selection of grid-spacing and sub-interval size along each dimension should facilitate accurate compression of tensor \( T \) with small CP rank. Our modeling framework therefore enables user-directed domain partitioning via specification of uniform or logarithmic to balance interpolation error and low-rank approximation error.

B. CP decomposition model formulation

We optimize low-rank CP decompositions of tensors \( T \) to achieve scale-independent minimization of prediction error. We first transform observed entries to enable the use of the MSE loss function, which among those summarized in Section II-A is most efficient and least susceptible to roundoff error. Specifically, we minimize Equation 2 with \( \phi(t_i, M_t) = \log(t_i - M_t^2) \). This simple technique minimizes deviation in the scale of observed runtimes and thereby enhances the robustness of MSE to both noise and biased under/over-prediction. We show this in Figure 1 using singular value decompositions, which minimize \( \sqrt{\text{MSE}} \). An increase in error with larger SVD rank signifies negative reconstructed matrix entries reset to \( 10^{-16} \) prior to evaluation of MLogQ. Use of logarithmic transformations attains non-negative CP decomposition model output implicitly without the requisite constraints to enforce non-negativity in factor matrix elements.

The resulting model approximates the execution time \( f(x) \) of configuration \( x \) as a function of \( M(x) = \sum_{a \in \{0,1\}^d} e^{M_{i+a}} \cdot \prod_{j=1}^d \left( 1 - \frac{\log(x_i) - \log(M_{i+a}^j)}{\log(M_{i+a}^j)} - \log(M_{i+e}^j) - \log(M_{i+e}^j) \right), \]
where \( M_i \leq x_i < M_{i+e}, \forall j \in \{1, \ldots, d\} \). Our modeling framework supports user-specified loss functions given gradient and Hessian information, as well as data transformations, the inverse of which is applied upon model inference. We summarize training and inference across a two-dimensional domain in Figure 2.

V. EXPERIMENTAL METHODOLOGY

1) Machine: We use the Stampede2 supercomputer at Texas Advanced Computing Center (TACC) [53]. Stampede2 consists of 4200 Intel Knights Landing (KNL) compute nodes (each capable of a performance rate over 3 Teraflops/s) connected by an Intel Omni-Path (OPA) network with a fat-tree topology (achieving an injection bandwidth of 12.5 GB/sec). Each KNL compute node provides 68 cores with 4 hardware threads per core. We use Intel environment 18.0.2, Intel MPI environment 18.0.2, and corresponding icc, mpicc, and mpicxx compilers. We generate all models using a single KNL core. We benchmark applications using \( \leq 64 \) KNL nodes and vary the number of processes-per-node and threads-per-process.

2) Application Libraries: We execute Intel MKL’s GEMM (MM) routine \( C_{m \times n} \leftarrow \alpha A_{m \times k} B_{k \times n} + \beta C_{m \times n} \) in a single-threaded environment with matrix dimensions \( 32 \leq m, n, k \leq 4096 \). We execute SLATE’s QR factorization (QR) routine [50] \( A_{m \times n} \leftarrow Q_{m \times R_{m \times n}} \) on \( p = \{1, 4, 16, 64\} \) nodes, first with 64 single-threaded processes-per-node (ppn), matrix dimensions \( 2^{10} \sqrt{p} \leq m \leq 2^{16} \sqrt{p}, 2^4 \sqrt{p} \leq n \leq 2^{10} \sqrt{p} \), and tuning parameters for processor-grid \( p_r \times p_x \) and outer/register block sizes \( oib \) chosen such that \( p_r = \min(\sqrt{\text{ib}}) \), \( n/256 \), \( ob = \min(128, |n/p_c|) \), and \( ib = \min(16, |n/p_c|) \). We then extend the dimension of the parameter space by varying each tuning parameter, yet still mapping a thread to each available core. We execute Intel MPI’s Broadcast (BC) routine on \( p = \{1, 2, 4, 8, 16, 32\} \) nodes with ppn \( n = \{1, 2, 4, 8, 16, 32\} \), and message size \( 2^k \leq m \leq 2^{24} \).

3) Dataset Generation: To optimize and evaluate performance models, we generate training sets and test sets for each kernel/application, respectively. The test sets for Broadcast and Kripke [52] are proxy applications for modern discrete ordinates neutral particle transport applications. We execute configurations across multiple hardware configurations and model each library’s Solve Time. Table II details each parameter space \( PS \) and defines \( |PS| \) as the parameter space dimension (number of evaluated parameters).

4) Model Assessment: We assess model prediction error using MLogQ as detailed in Section II-A. We assess model size by writing configured models to a file via Python’s joblib package and measuring the file size. We assess model time by measuring the time to configure a model on a single core. We forgo consideration of models optimized in \( \geq 1200 \) seconds in Figures 6.8, or models \( \geq 1 \) megabyte in Figures 7.8.

5) Model Tuning: We evaluate the following prediction methods: grid-based multilinear interpolation (MI), sparse grid regression (SGR), multi-layer perceptrons (NN), random forests (RFR), gradient-boosting (GBR), extremely-randomized trees (ETR), Gaussian process regression (GPR), support vector machines (SVM), adaptive spline regression (MARS), k-nearest neighbors (KNN), and our methodology described in Section IV (CPR). We utilize the Cyclops Tensor
Application | Input | Architectural | Configuration
--- | --- | --- | ---
ExaFMM [49] | Fast Multiple Method | Input: number of particles n, expansion order ord, #threads, particles per leaf ppl, tree level used for partitioning tl | $2^0 \leq n \leq 2^{13}$, $0 \leq$ ord $\leq 8$, $2^0 \leq \#$threads $\leq 2^6$, $4 \leq ppl \leq 128$, $0 \leq tl \leq 4$
SLATE [50] | QR factorization | $[\text{matrix } m \times n], \#$nodes $p$, processes-per-node ppl, outer block size ob, inner block size ib, processor grid $p_x \times p_y$ | $2^{10} \leq m \leq 2^{16}, 2^{10} \leq p \leq 2^6$, $2^0 \leq ob \leq 2^7, 2^{10} \leq ib \leq 2^6, 2^1 \leq pc \leq 2^5$
SNAP [51] | Discrete ordinates transport | $[\text{see } [51] \text{ for descriptions of } \{\text{nnode, nang, ng, ichunk}\}, \#$nodes $N$, processes-per-node ppl, #threads/process tp | $2^0 \leq \text{nmom} \leq 2^2, \text{nang} \leq 2^{12}, \text{ng} \leq 2^6$, $2^0 \leq \text{p} \leq 2^6$, $2^0 \leq \text{tp} \leq 2^{12}$
KRPKE [52] | Discrete ordinates transport | $[\text{see } [52] \text{ for descriptions of } \{\text{groups, legendre, quad, zones, dset, gset, zset, layout, solver}\}, \#$nodes $N$, processes-per-node ppl, #threads/process tp | $2^3 \leq \text{groups} \leq 2^{10}, 0 \leq \text{legendre} \leq 5, 2^3 \leq \text{quad} \leq 2^7, 2^3 \leq \text{zones} \leq 2^8$

| Library | Application | [PS] | Input |
| --- | --- | --- | --- |
| ExaFMM [49] | Fast Multiple Method | Input: number of particles n, expansion order ord, #threads, particles per leaf ppl, tree level used for partitioning tl | $2^0 \leq n \leq 2^{13}$, $0 \leq$ ord $\leq 8$, $2^0 \leq \#$threads $\leq 2^6$, $4 \leq ppl \leq 128$, $0 \leq tl \leq 4$
| SLATE [50] | QR factorization | Input: matrix $m \times n$, #nodes $p$, processes-per-node ppl, outer block size ob, inner block size ib, processor grid $p_x \times p_y$, | $2^{10} \leq m \leq 2^{16}, 2^{10} \leq p \leq 2^6$, $2^0 \leq ob \leq 2^7, 2^{10} \leq ib \leq 2^6, 2^1 \leq pc \leq 2^5$
| SNAP [51] | Discrete ordinates transport | Input: #nodes $N$, processes-per-node ppl, #threads/process tp | $2^0 \leq \text{nmom} \leq 2^2, \text{nang} \leq 2^{12}, \text{ng} \leq 2^6$, $2^0 \leq \text{p} \leq 2^6$, $2^0 \leq \text{tp} \leq 2^{12}$
| KRPKE [52] | Discrete ordinates transport | Input: #nodes $N$, processes-per-node ppl, #threads/process tp | $2^3 \leq \text{groups} \leq 2^{10}, 0 \leq \text{legendre} \leq 5, 2^3 \leq \text{quad} \leq 2^7, 2^3 \leq \text{zones} \leq 2^8$

### Table II: Parameter space ([PS]) description for parallel QR factorization and parallel scientific applications involving particle interaction and particle transport. Mapping of parameters to tensor modes reflects ordering in parameter description. See citations for further clarification.

Fig. 3: Prediction accuracy for grid-based models. Discretization granularity signifies the number of cells for MI/CPR and $2^{\text{number of levels for SGR}}$. CPR: R$k$ denotes a rank-$k$ CP decomposition. [PS] denotes the number of benchmark parameters. N denotes the training-set size. Annotations show number of grid-points placed across the modeling domain (CPR/MI are same).

Framework [54] for efficient CP decomposition optimization. We utilize the PyEarth library to evaluate MARS, the SG++ [18] library to evaluate SGR, and the Scikit-Learn [55] library to evaluate the remaining methods. We evaluate all relevant model configurations using the same training set and forgo training via cross-validation. In particular, we evaluate CP rank $1 \rightarrow 48$ and grid-cell count $2 \rightarrow 256$ for CPR; $2 \rightarrow 8$ discretization level, $1 \rightarrow 16$ refinements, and $4 \rightarrow 32$ adaptive grid-points for SGR; max spline degree $1 \rightarrow 6$ for MARS; max tree depth $2 \rightarrow 16$ and max tree count $2 \rightarrow 256$ for RFR, GBR, and ETR; $1 \rightarrow 6$ neighbor count for KNN, out-of-box covariance kernels for GPR; {poly, rbf} kernels and polynomial degree $1 \rightarrow 3$ for SVM; $1 \rightarrow 8$ hidden layers each of size $2 \rightarrow 8192$ for NN. CPR and MI models place grid-points equally-spaced on a log-scale/linear-scale along the ranges of input, architectural/configuration parameters, and interpolate as necessary. We select regularization parameter $\lambda = 10^{-k}, \forall k \in \{-6, \ldots, -1\}$ for CPR and SGR. We set a maximum number of 100 sweeps and 1000 iterations of alternating least-squares and conjugate gradient for CPR and SGR, respectively, and set a tolerance of $10^{-4}$ for SGR.

### VI. EXPERIMENTAL EVALUATION

We evaluate models generated by methods described in Sections II-C and specified in Section V-5. We first ablate piecewise/grid-based (P/G) models by varying the structure and refinement of underlying grids. We then incorporate alternative supervised learning methods to compare model accuracy as a function of model size and number of training observations. These studies reveal the tensor size and CP rank with which CP decompositions outperform alternative models.

#### A. Ablation Studies for Piecewise/Grid-based Models

Existing P/G performance models exhibit fundamental trade-offs between accuracy and scalability. Each metric is significantly influenced by discretization geometry (i.e., location of grid-points) and granularity (i.e., number of grid-points dictated by discretization geometry) across the modeling domain. Figure 3 demonstrates that given sufficiently many training observations, both multilinear interpolation using regular grids (MI) and sparse grid regression (SGR) systematically improve model accuracy by lowering discretization granularity. Although the discretization geometry adopted by SGR models offers enhanced scalability (in overall number of grid-points),...
in high dimensions, this results in higher error. Grid-point geometry selected via search alone (MARS) configures global models that are significantly less accurate than MI methods.

Our approach, CP decomposition of regular grids (CPR), decouples discretization granularity of the modeling domain from model accuracy if performance across its underlying grids is sufficiently low-rank. Figure 3 reveals that CPR models are consistently more accurate than MI models despite being exponentially smaller in size. As both models utilize regular grids, we attribute this improvement to CPR models interpolating averaged intra-cell performance rather than that of distinct configurations. Both strategies systematically reduce error as grid-cell count increases, yet CPR models achieve this error reduction only with sufficient CP rank. Model accuracy and CP rank exhibit positive correlation as additional piecewise multilinear cost functions are incorporated into the model only for sufficiently large grids. We observe in Figures 3, 4 that a CP rank between 6 and 12 mitigates the impact of low-rank approximation error for all evaluated tensors.

Compression of high-dimensional regular grids via CP...
decomposition is more accurate and efficient than regression across high-dimensional sparse grids. We observe in Figure 3 that CPR models achieve higher prediction accuracy than SGR models at the same discretization granularities by leveraging regular grid discretization geometries. In particular, CPR achieves superior accuracy using grids of size $k \times 6 \times 7$, $k \times k \times k$, and $k \times k \times 4$ with $k \geq 8$ for the MPI Broadcast (BC), matrix multiplication (MM), and QR factorization (QR) kernels, respectively. Accuracy continues to improve even as underlying tensors become increasingly sparse. The mean error of a CPR model for the MM kernel with CP rank $R = 12$ is 2.47x smaller from $k = 8$ to $k = 64$, yet the density of its underlying tensor decreases from 100% to 22%, respectively.

The optimization time to configure accurate CPR models is significantly smaller than that for accurate sparse grid models. Following specification of an initial sparse grid (see Figure 3), its refinement is performed sequentially. In particular, SGR’s refinement mechanism sweeps over training data to identify grid-points and corresponding basis functions contributing most to training error. SGR then lowers the discretization granularity around those grid-points before restarting the procedure on the newly refined grid. As observed in Figure 4, sparse grid models cannot achieve competitive accuracy without sufficient refinement. A CPR model’s underlying grid is instead fixed, and refinement consists of increasing the CP rank to reduce prediction error for all tensor elements.

The enhanced scalability offered by CPR is evident in Figure 5 by varying training-set size $N$ and parameter space dimension $|PS|$. A rank $R = 8$ CP model for MM with $N = 65536$ achieves better accuracy and is $> 256x$ faster to optimize. We observe a smaller speedup when reducing to training set to $N = 4096$ for QR with 3D modeling domain, yet a similar speedup is attained solely by increasing in dimension of modeling domain to 7D. We conclude that CPR is the only scalable method able to systematically reduce prediction error given user-directed discretization.

Accurate CP decompositions may be configured largely independent of tensor sparsity and constraints among bench-
mark parameters. The most accurate CPR models for MM, QR, and Kripke in Figure 6 utilize CP ranks 16, 8, and 10 to model tensors of size 16x16x16, 32x32x4x4x16x8x6, and 5x5x3x3x3x2x6x6x2x4x4x4 that are 83\%, 0.03\%, and 0.03\% dense, respectively. Thus, more than 99.9\% of grid-cells are unobserved when modeling QR and Kripke, whereas the underlying tensor for MM is nearly dense. CPR models with fewer underlying grid-cells can utilize larger CP ranks when optimized using N observations. We find this trade-off suitable when modeling the performance of MM. In particular, CP decompositions of k\times k tensors can be effectively optimized using N = 8192 if k \leq 16, yet an error reduction of 1.38x is achieved in Figure 3 for tensors of sizes between 16 \leq k \leq 64 using N = 65536. In contrast, we place significantly more grid-points along the higher-dimensional application domains (e.g., those of Kripke, ExaFMM, SNAP) and decompose the corresponding tensors with smaller CP rank to account for a smaller number of training observations N \leq 4096.

B. Comparison Studies for Supervised-Learning Methods

We observe in Figure 6 that CPR models achieve highest prediction accuracy relative to model size for MM, QR, and Kripke. The most accurate alternative models require a 20.2x geometric mean factor increase in model size to achieve equivalent error. We further observe in Figure 7 that accurate CPR models can be optimized with small training sets, and that CPR model accuracy improves by a geometric mean factor increase of 2.59x with 4x more training observations. However, alternative models are more competitive in this regime, indicating that CPR is most effective relative to the state-of-the-art for training sets of size N \geq 1024. In particular, CPR models achieve a geometric mean decrease in prediction error of 0.75x for training sets of size N \leq 256, while that factor increases to 1.35x for N \geq 512. Finally, we highlight in Figure 8 that CPR models achieve a geometric mean decrease in prediction error of (2.18x, 1.40x, 1.35x) relative to the most accurate sparse grid and neural network models of maximum size (10,100,1000) kilobytes, respectively, for training sets of size N \geq 1024. We conclude that other supervised learning methods cannot achieve similar accuracy when considering all kernels/applications, despite using significantly larger models.

SGR models are most competitive with CPR relative to model size, yet the automated and irregular placement of grid-points following iterative grid refinement is not appropriate for all applications (e.g., ExaFMM in Figures 7, 8) and requires expensive mapping logic which partially negates the storage benefits of sparse grids in high dimension. Further, as shown in Figure 5, SGR cannot efficiently leverage larger training sets to reduce error via grid refinement. Compositions of nonlinear functions (e.g., neural network (NN)) model complex performance behavior less directly than piecewise multilinear cost functions, and compensate by utilizing significantly larger models. However, as shown in Figure 8, neural networks are most competitive with CPR for large training sets, which both can efficiently utilize to reduce model error.

Non-piecewise polynomial cost models (e.g., MARS) cannot fit to complex performance data using any specified spline degree. Recursive partitioning methods (e.g., extremely-randomized tree regression (ETR)) can partition the modeling domain as a regular-grid, yet feature no compression mechanism. The alternative partitions it learns to retain scalability are consequently less accurate. We observe offline that ETR and support vector machines (SVM) are consistently more accurate and no less memory-efficient than other partitioning methods (e.g., gradient-boosting, random forest regression) and Gaussian-process regression, respectively, and thus forgo further discussion of these methods. Instance-based methods (e.g., k-nearest neighbors (KNN)) achieve increasingly poor accuracy in higher dimensions due to increasingly sparse observation of the modeling domain.

VII. Conclusion

Our experimental analysis demonstrates that CP decompositions accurately and efficiently model high-dimensional application performance. These piecewise multilinear cost models are significantly more accurate than alternative models relative to model size, and systematically reduce prediction error given increasingly many observations of a fixed modeling domain. The impetus for tensor completion driven by artificial intelligence (e.g., recommendation systems, image recognition) ensures that performance modeling frameworks that adopt this technique will benefit from its continued optimization and dissemination among software libraries.

This work presents promising avenues for autotuning research involving the use of tensor factorizations as surrogate models of application performance. An immediate extension would include incorporating methods for efficiently updating CP decompositions to effectively model streaming data in online settings. The prevalence of highly-constrained parameter spaces promotes investigation into the use of tensor completion applied to execution datasets with structured sparsity. The efficacy of tensor completion applied to extrapolation and transfer learning tasks also warrants further investigation.

ACKNOWLEDGMENTS

The first author would like to acknowledge the Department of Energy (DOE) and Krell Institute for support via the DOE Computational Science Graduate Fellowship (grant No. DE-SC0019323). This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number ACI-1548562. Via XSEDE, the authors made use of the TACC Stampede2 supercomputer (allocation TG-CCR180006). This research has also been supported by funding from the National Science Foundation (grant No. 2028861).

REFERENCES

[1] R. Vuduc, J. W. Demmel, and J. A. Bilmes, “Statistical models for empirical search-based performance tuning,” The International Journal of High Performance Computing Applications, vol. 18, no. 1, pp. 65–94, 2004.
B. Recht, C. Re, S. Wright, and F. Niu, “Hogwild!: A lock-free approach to parallelizing stochastic gradient descent,” Advances in neural information processing systems, vol. 24, 2011.

P. Jain, P. Netrapalli, and S. Sanghavi, “Low-rank matrix completion using alternating minimization,” in Proceedings of the forty-fifth annual ACM symposium on Theory of computing, 2013, pp. 665–674.

R. Keshavan, A. Montanari, and S. Oh, “Matrix completion from noisy entries,” Advances in neural information processing systems, vol. 22, 2009.

H.-F. Yu, C.-J. Hsieh, S. Si, and I. Dhillon, “Scalable coordinate descent approaches to parallel matrix factorization for recommender systems,” in 2012 IEEE 12th international conference on data mining. IEEE, 2012, pp. 765–774.

T. Hastie, R. Mazumder, J. D. Lee, and R. Zadeh, “Matrix completion and low-rank SVD via fast alternating least squares,” The Journal of Machine Learning Research, vol. 16, no. 1, pp. 3367–3402, 2015.

C. Feltjouidi, F. Makari, and R. Gemulla, “Distributed matrix completion,” in 2012 IEEE 12th international conference on data mining, IEEE, 2012, pp. 655–664.

L. Karlsson, D. Kressner, and A. Uschmajew, “Parallel algorithms for tensor completion in the CP format,” Parallel Computing, vol. 57, pp. 222–234, 2016.

S. Smith, J. Park, and G. Karypis, “An exploration of optimization algorithms for high performance tensor completion,” in SC ’16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE, 2016, pp. 359–371.

R. Yokota, “An fmm based on dual tree traversal for many-core architectures,” Journal of Algorithms & Computational Technology, vol. 7, no. 3, pp. 301–324, 2013.

M. Gates, J. Kurzak, A. Charara, A. Yarkhan, and J. Dongarra, “Slate: design of a modern distributed and accelerated linear algebra library,” in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2019, pp. 1–18.

T. Deakin, S. McIntosh-Smith, and W. Gaudin, “Many-core acceleration of a discrete ordinates transport mini-app at extreme scale,” in International Conference on High Performance Computing. Springer, 2016, pp. 429–448.

F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg et al., “Scikit-learn: Machine learning in Python,” the Journal of machine Learning research, vol. 12, pp. 2825–2830, 2011.