LLNL-CONF-723337

Apollo: Reusable Models for Fast, Dynamic Tuning of Input-Dependent Code

D. A. Beckingsale, O. Pearce, I. Laguna, T. Gamblin

February 13, 2017

31st IEEE International Parallel & Distributed Processing Symposium
Orlando, FL, United States
May 29, 2017 through June 2, 2017

Apollo: Reusable Models for Fast, Dynamic Tuning of Input-Dependent Code

David Beckingsale, Olga Pearce, Ignacio Laguna, Todd Gamblin
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
Livermore, CA 94550
Email: {david, olga, ilaguna, tgamblin}@llnl.gov

***Abstract*—Increasing architectural diversity makes performance portability extremely important for parallel simulation codes. Emerging on-node parallelization frameworks such as Kokkos and RAJA decouple the work done in kernels from the parallelization mechanism, allowing for a single source kernel to be tuned for different architectures at compile time. However, computational demands in production applications change at runtime, and performance depends both on the architecture and the input problem, and tuning a kernel for one set of inputs may not improve its performance on another. The statically optimized versions need to be chosen dynamically to obtain the best performance. Existing auto-tuning approaches can handle slowly evolving applications effectively, but are too slow to tune highly input-dependent kernels. We developed Apollo, an auto-tuning extension for RAJA that uses pre-trained, reusable models to tune input-dependent code at runtime. Apollo is designed for highly dynamic applications; it generates sufficiently low-overhead code to tune parameters *each time a kernel runs*, making fast decisions. We apply Apollo to two hydrodynamics benchmarks and to a production multi-physics code, and show that it can achieve speedups from 1.2x to 4.8x.**

I. Introduction

Performance portability is extremely important for modern simulation codes. Algorithms must run efficiently on an increasing number of target architectures, but choosing the best threading model, block size, scheduling policy, and other parameters for each architecture is a daunting task. Large codes contain thousands of independent kernels, and developing and maintaining multiple versions of each one is infeasible. Portability frameworks such as RAJA [1] and Kokkos [2] have emerged to fill this gap, allowing developers to separate the concerns of tuning and correctness. Developers write one, single-source, version of each kernel, and set architectural tuning parameters at compile time using C++ template parameters.

The performance of modern scientific codes depends not only on the target architecture, but also on input-dependent aspects of the code. For example, in an adaptive mesh refinement (AMR) simulation, a kernel may handle a wide range of patch sizes. A small patch may not contain enough computational work to amortize the cost of spawning a parallel OpenMP region, so we might choose to execute it sequentially. Moreover, such behaviors evolve over the course of a run. These dynamic applications require *on-line* tuning to achieve the best performance, as tuning parameters may need to be set based on the state of the code each time the kernel executes.

Auto-tuning is a broad field, and there has been much existing work on tuning the performance of codes. Extensive work has been done in the area of compile-time auto-tuning [3], [4], [5] and run-time adaptation [6], [7], [8], [9], [10]. Existing approaches typically perform a large, guided search of the performance parameter space, running many instances of the kernel to determine the best assignment of tuning parameters. Despite a host of techniques that prune the search space and reduce the required number of trials, the fastest searches today still take minutes to run, meaning existing tuning approaches work only if the code's behavior changes slowly.

To cope with this complex tuning problem, we use machine-learning in an off-line training step to generate classifiers that can rapidly predict the fastest parameter values for a kernel at runtime. Using data collected from training runs, we train a decision tree classifier, and generate light-weight code for the classifier that can be used on-line to dynamically select tuning parameters for each kernel. To the best of our knowledge, this is the first technique to approach tuning at this granularity with this level of generality *and* responsiveness.

Our work is implemented in *Apollo*, an extension of the RAJA performance-portability library. This paper makes the following contributions:

1) A novel tuning technique using simple decision tree classifier models instead of costly search strategies;
2) An interface for collecting arbitrary training features from kernel executions in multi-physics codes;
3) A method for dynamically selecting statically optimized code paths at runtime; and
4) *Apollo*, a production-ready framework that implements these techniques and has been tested on a real multi-physics code.

We have used our framework in two proxy applications, and with several different input problems in a production multi-physics code, where Apollo achieves speedups from 1.2x to 4.8x.

II. Background

To tune applications, we must have a way of both expressing and selecting the possible values for a given tuning parameter. In this paper, we focus on tuning applications that use the RAJA performance portability framework. In the following sections we describe RAJA, as well as the parameters we tune and the applications to which we apply this tuning.

A. RAJA Programming Model

Supercomputing architectures are increasingly diverse, and applications must be adapted to each architecture to achieve the best performance. Developing and maintaining multiple implementations of complex scientific applications with thousands of lines of code is infeasible, so having a single-source code at the application kernel level is essential. RAJA allows the applications to maintain a single source code, while allowing performance portability to different architectures. RAJA decouples the kernels in the application from the execution parameters, enabling static tuning of the policy used to perform the loop iterations. RAJA uses lightweight syntax and standard C++ features for portability and ease of integration into existing production applications.

RAJA enforces decoupling of the kernel body from its execution model (how the iterations are scheduled to hardware) with C++11 lambda functions. Using a lambda function, the kernel body and its surrounding scope is captured and passed to RAJA's `forall` execution method. The execution method allows a single-source kernel body to be dispatched to multiple programming model backends, determined by the `exec_policy`. The following listing contains an example RAJA kernel:

```
RAJA::forall<exec_policy>(iset, [=](int i) {
  sigxx[i] = sigyy[i] = sigzz[i] = - p(i) - q(i);
});
```

The `exec_policy` is the execution policy, a template parameter that determines how the kernel execution will be scheduled onto the hardware. This code is translated to a specific, specialized version of the generic `forall` method based on the template parameter.The iteration pattern and range of indices can be defined by an `IndexSet` object, the `iset` parameter.

Since a template parameter is used to specialize the `forall` method for specific execution backends, the code can be both inlined and optimized by the compiler. Note that the `forall` method is also templated on the type of the lambda function. Since each C++ lambda function has a unique type, a unique instance of this `forall` function will be created for every application kernel, meaning that the compiler can optimize each code path.

B. Input-Dependent Parameters

In this paper we focus on input-dependent parameters, those whose best value depends on information known only at application runtime. The most obvious tuning parameter exposed by RAJA is the execution policy, which determines whether a kernel is executed sequentially or in parallel using OpenMP threads. By default, RAJA requires the execution policy to be selected at compile time, allowing for static tuning. However, the best execution policy may depend on dynamic parameters, like the number of iterations required.

We also expose an additional OpenMP parameter that can be tuned dynamically. OpenMP's `static` schedule can take an integer parameter to control how blocks of loop iterations are shared between threads. Specifically, the parameter controls the number of consecutive iterations that get assigned to each thread. For example, a value of 1 would interleave the iterations across threads. The default value is N/t where N is the number of iterations, and t is the number of OpenMP threads.

C. Potential Benefits of Dynamic Tuning

Statically choosing an execution policy for a kernel prevents us from making the best choice based on run-time adaption of the code, and the static choice will not be fastest in all cases. To determine the potential improvement of a fast, dynamic tuning approach, we examined kernels from several applications of interest to the U.S. Department of Energy.

a) LULESH: a proxy application that models shock hydrodynamics developed to aid the Department of Energy's co-design effort[11]. LULESH has two main categories of kernels using RAJA's `forall` construct. The first category iterates over domain elements, with problem size-dependent iteration counts. The second category iterates over material regions; these kernels have lower iteration counts dependent solely on the number of material regions in the problem (the default case has 11 iterations).

b) CleverLeaf: a shock hydrodynamics proxy application with Adaptive Mesh Refinement developed at the University of Warwick and the UK Atomic Weapons Establishment. CleverLeaf uses the SAMRAI library from Lawrence Livermore National Laboratory to add an AMR capability to a parent shock hydrodynamics mini-application [12], working on dynamically sized *patches* of data that represent areas of interest in the physical domain with different levels of resolution. There are 88 `forall` kernels in CleverLeaf, the majority of which are over all the elements of the current AMR patch. The other kernels in CleverLeaf iterate over the boundary regions of these patches in strips that are 2 elements wide in order to apply the physical boundary conditions to the hydrodynamics problem. The main input-dependence in CleverLeaf thus comes from the shape and size of the patches.

c) ARES: an arbitrary Lagrangian-Eulerian (ALE) radiation hydrodynamics code capable of running small serial applications to large, massively parallel applications[13], [14] on millions of processors. ARES comprises several million of lines of code, and it is used primarily in munitions modeling and inertial confinement fusion applications. ARES also has an AMR capability, with patches created dynamically as the simulation evolves. One of the physics packages in ARES has been ported to use 536 RAJA kernels that iterate over a number of different aspects of the problem domain. ARES contains a mixed material capability; however, the number of material regions is not fixed and can change dynamically during the course of a simulation as materials mix together.

The kernels in the above applications display significant performance variability and the fastest execution policy can be 1-3 orders of magnitude faster than the slowest (Figure 1). There is a great potential for performance improvement by dynamically selecting the best policy each time a kernel executes.

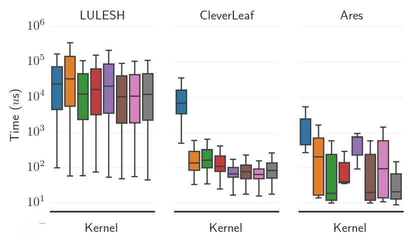


Fig. 1: Runtime variation for different execution policy choices in LULESH, CleverLeaf, and ARES.

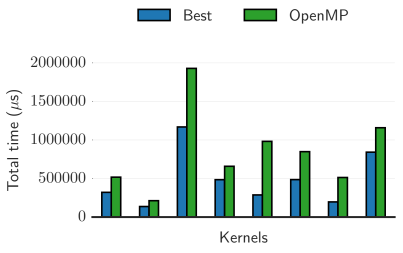


Fig. 2: Total time spent in the eight most variable kernels in CleverLeaf using the dynamic fastest policy selection, compared to statically choosing OpenMP everywhere.

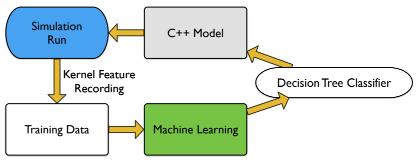


Fig. 3: Workflow for dynamically tuning applications with Apollo.

D. *Static Optimization and Dynamic Policy Selection*

When tuning an application to the hardware, an application developer might select a single parameter value for the entire run. However, due to the input-dependence, choosing a single value will often result in significantly slower runtime when compared to a dynamic selection of the parameters. Figure 2 shows the potential runtime if we were able to pick the best execution policy for each unique kernel launch in CleverLeaf, rather than statically selecting OpenMP.

While the obvious way to dynamically select execution policies at runtime would be to use more generic execution methods, the language and compiler design for C++ prevents this from being a good choice because it prevents many static optimizations that compilers are able to perform. We developed a version of LULESH where all kernels shared a common OpenMP execution function meaning that the RAJA `forall` method is no longer specialized for every single kernel, and instead only once. The result of this more general implementation was a 30% slowdown, which is not acceptable for our target scientific applications. We need a framework that allows both static code optimization and dynamic policy selection.

III. Approach & Implementation

Apollo is a framework for tuning templated kernel execution parameters at runtime. Figure 3 shows the Apollo workflow. First, we run the application to generate training data samples. The samples are the input to Apollo's model-generation framework which trains a per-kernel decision model that can predict the fastest parameter values for each kernel execution. These models can be linked into the application dynamically, without recompilation. We designed Apollo's decision models to execute with very low overhead, and we trade the costly on-line search used by existing auto-tuners for off-line training.

Apollo leverages RAJA's separation of concerns. Programmers continue to write single-source kernels, but they no longer need to select an optimal execution policy. Apollo selects the fastest known template variant of the kernel at runtime, and generates the code to perform the dynamic selection. This provides the performance of statically optimized kernels *and* selects the best kernel version for the input data. The remainder of this section describes our implementation in detail.

A. *Recording Training Data*

During a training run, we collect data about each kernel execution. This information takes the form of a tuple, or, in machine learning parlance, a *feature vector*. The vector serves as the input to our decision model. Each element of the tuple is the value of a particular feature of the kernel. For each feature vector, we also record the kernel's runtime. The features we collect are designed to characterize each kernel, as well as to capture application-specific information that might affect kernel performance. We collect the following distinct categories of information about the kernel invocation:

1) Kernel features, collected from parameters passed to the RAJA `forall` method;
2) Instruction features, gathered from the lambda function that corresponds to the kernel body;
3) Measurements of the kernel runtime;
4) Application features, optionally specified by the application developer.

The kernel features describe the type of the kernel (e.g., forall), the constraints on the number of iterations, and the index type.

Instruction features capture the frequency with which specific instruction mnemonics occur within each lambda function, and provide an insight into the character of the kernel. We measure function size by recording the number of instructions in the lambda function that represents the kernel body. We must have instruction counts prior to making a prediction, so we collect them from the application binary using the Dyninst library [15].

We use a lightweight annotation system, Caliper [16], to measure the runtime of the kernels, and to store arbitrary additional attribute-value pairs representing additional features. Caliper provides a simple interface that allows application developers to add semantic annotations of interest to each kernel. For example, a developer could provide the current time step or the dimensions of a patch in an AMR mesh.

Table I lists the features we collect. While this is a small sample of all the runtime features we could collect in a multi-physics code, it allows us to explore the efficacy of our technique for improving application performance. New features, such as additional problem-specific information from an application input file, can be easily added to characterize kernels and their input data more completely. However, relying

	Feature	Description
🟦	func func_size index_type loop_id num_indices num_segments stride	Name of function Total number of instructions in kernel body Type of RAJA IndexSet Address identifying kernel Number of indices in each segment Number of segments Stride of indices in each segment
🟩	add, and, call, cmp, comisd, divsd, inc, jb, lea, loop, maxsd, minsd, mov, mulpd, nop, pop, push, pxor, ret, sar, shl/sal, sqrtsd, sub, test, ucomisd, unpckhpd, unpcklpd, xor, xorps	Instruction mnemonics are grouped to save space (for example, the add mnemonic corresponds to add, addpd and addsd).
🟥	timestep problem_size problem_name patch_id	Current cycle Global problem size Name of the input deck Numeric ID assigned to the AMR subdomain being processed (CleverLeaf only).

TABLE I: Kernel (🟦), instruction(🟩), and application (🟥) features collected for each RAJA kernel.

on too many parameters can perturb execution in real runs. In practice, we use feature importance analysis to find small sets of important features and to reduce the size of our models; we discuss feature importance analysis in more detail in Section IV-B. To generate a complete set of training data, a given input problem must be run multiple times, once for each value of the parameter we are modeling.

Execution policy is a static parameter. To allow us to vary the execution policy dynamically, we developed a RAJA extension which reads the execution policy from an environment variable. Using the new generic lambda feature added to C++14, we use the `auto` keyword to determine the type of the execution policy passed into the RAJA forall method

```
template <typename LOOP_BODY>
void forall(policy_from_env,
            Index_type begin, Index_type end,
            LOOP_BODY loop_body) {
  apollo::policySwitcher(POLICY, [=](auto exec_policy) {
     forall( exec_policy,
             begin, end,
             loop_body );
  });
}
```

The `apollo::policySwitcher` method implements a switch statement that uses a policy enumerator to determine which policy type to pass to the lambda function. This allows us to maintain the static optimization advantages of templates, whilst still exposing all possible template choices as dynamically tunable parameters at runtime.

```
template <typename BODY>
void policySwitcher(POLICY_TYPE policy, BODY body) {
  switch (policy) {
    case seq_segit_seq_exec:
      body(RAJA::seq_exec());
      break;
    case seq_segit_omp_parallel_for_exec:
      body(RAJA::omp_parallel_for_exec());
      break;
    // other policies...
  }
}
```

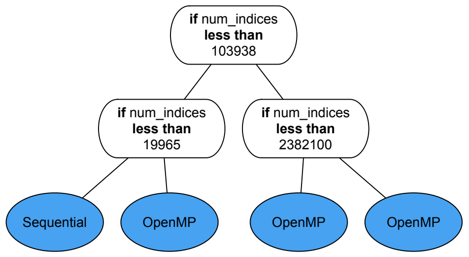

if num_indices less than 103938
if num_indices less than 19965
if num_indices less than 2382100
Sequential
OpenMP
OpenMP
OpenMP

Fig. 4: Example decision tree model to predict execution policies based on the `num_indices` feature.

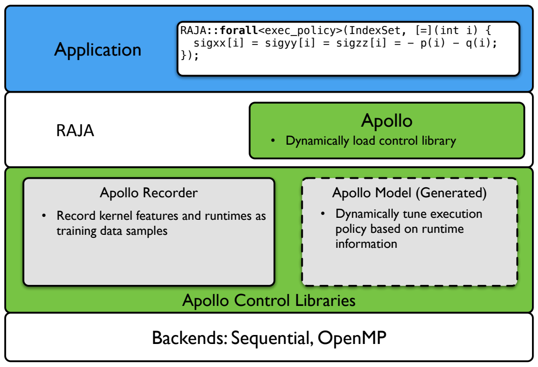
B. Building Lightweight Decision Models

We use the data set collected by our measurement framework to build runtime auto-tuners, which select the fastest execution model for each kernel when it is executed. We model the problem of selecting a code variant (in this case a template instantiation) as a classification problem. In machine learning, the *classification* problem asks us to identify the category to which a sample belongs. Our samples are the feature vectors we collected for each kernel invocation, and we categorize samples into groups by their fastest execution model. Using this classifier at runtime allows us to select the best kernel implementation for its input data.

To train a classifier, we start with the feature vectors mentioned in Section III-A. For each training run, we collect a set of feature vectors mapped to their runtimes. Because there are multiple training runs with different execution models, the same feature vector may map to many possible runtimes. We *label* each unique feature vector with the execution model that resulted in the fastest runtime. We feed these labeled feature vectors to a learning algorithm, which trains a *classifier* to select a label given a feature vector.

There are many types of classifiers in the literature. For Apollo, we chose to use a *decision tree* classifier, which uses a binary tree-like model of decisions to label samples [17], [18]. To make a prediction for an unseen sample, the tree is evaluated from the root to a leaf, and at each node a comparison of one feature value determines which child node to visit. We use decision trees for two reasons. First, they are comparatively simple, and it is very easy to convert a decision tree model into a set of conditional statements that can be executed at runtime. Second, it is easy to prune decision trees to create smaller, less complex models: we simply cut the tree off at a low level and evaluate only the first few levels of conditions. This allows us to consider fewer input features if we need to reduce model evaluation cost. As we add a larger number of tuning parameters to Apollo, we may need to consider more complex classifiers.

We implemented our data processing and model generation as a Python package, that is a complement to the runtime C++ components in the Apollo framework. We read training data samples into Pandas dataframes [19], process them, then convert them to NumPy [20] arrays for use with the scikit-learn package [21].


Application
RAJA
Apollo
• Dynamically load control library
Apollo Recorder
• Record kernel features and runtimes as training data samples
Apollo Model (Generated)
• Dynamically tune execution policy based on runtime information
Apollo Control Libraries
Backends: Sequential, OpenMP

```
RAJA::forall<exec_policy>(IndexSet, [=](int i) {
  sigxx[i] = sigyy[i] = sigzz[i] = - p(i) - q(i);
});
```

Fig. 5: Apollo interface for training data collection and dynamic kernel tuning.

C. *Dynamically Tuning Application Parameters*

Apollo uses a decision tree to generate code for run-time auto-tuning. This approach differs from existing auto-tuning frameworks in several respects. First, most existing frameworks select the fastest version of each kernel by executing it many times with different tuning parameters. This precludes any input-dependent optimizations, as it selects a kernel version before run-time, when the kernel's inputs are known. Second, frameworks that can adapt at runtime search the tuning parameter space at run-time, which incurs high overhead and does not allow kernels to adapt quickly to changing inputs. The tuning parameter space can be very large, and even sophisticated search methods can take minutes to converge.

Apollo's tuners trade costly on-line search for off-line training. Training can be expensive, but it is performed off-line before the program runs. Decision trees encode the knowledge required to make a tuning decision quickly at runtime, and they can use measured features to make decisions at runtime. Instead of exploring the tuning parameter space, Apollo can select a kernel variant directly based on the run-time values of features. This allows kernels to quickly adapt to their input data with only a few conditional evaluations. For example, Apollo kernels can use differently optimized code variants for each differently-sized patch of data in an AMR simulation.

Apollo generates a tuning model from a decision tree using a simple code generation algorithm. We traverse the decision tree generated by our training algorithm and generate nested conditional statements that test the values of features corresponding to each node. Internal nodes in the tree become `if` statements, and the leaves become assignments that select a kernel variant or parameter value to be used at runtime.

We developed a simple interface that enables Apollo to tune RAJA-exposed kernel execution parameters without further modifications to the application (Figure 5). In RAJA, we have added `apollo::begin()` and `apollo::end()` hooks before and after each RAJA loop template. Through this interface, RAJA interfaces with one of two Apollo components:

1) Recorder: Collects kernel features and measures the execution item to use as training data;
2) Tuner: Selects kernel execution parameters at runtime.

Because Apollo components are decoupled from the application, we can choose which Apollo component (recorder or tuner) to load at runtime. We can run the same executable in either recording or tuning mode. The control code looks like this:

```
template <typename EXEC_POLICY_T, typename INDEXSET_T, typename LOOP_BODY>
void forall(const INDEXSET_T& iset, LOOP_BODY loop_body) {
 // Delegate begin hook to control library
 if (apollo::controlLibraryLoaded()) {
  apollo::begin_forall_iset(static_cast<IndexSet>(iset),
                (void*) &LOOP_BODY::operator());
 }
 // Execute RAJA forall kernel
 forall(EXEC_POLICY_T(), iset, loop_body);

 // Delegate end hook to control library
 if (apollo::controlLibraryLoaded()) {
  apollo::end_forall_iset(static_cast<IndexSet>(iset));
 }
}
```

The Apollo calls pass the `IndexSet` object and the kernel body (as a lambda function) to the loaded Apollo component. The recorder simply stores observed feature values in a file to be used as training inputs. The tuner contains a function with the generated conditional code from our decision tree. It uses runtime feature information to tune the execution parameters. With this design, the decision model is not tightly coupled with the code, and we can re-train our decision model on new recording data *without* recompiling the application. This allows us to generate new, finely tuned decision models over time as the inputs to our application codes change.

When a kernel is executed, the `begin` call of the Apollo interface uses the predicted parameter values to tune the RAJA execution policies. A simple model generated to tune execution policies looks like this:

```
void apollo_begin_forall_iset(const RAJA::IndexSet& iset, void*) {
 RAJA::apollo::model_params p;
 const int num_indices = iset.getLength();

 if ( num_indices <= 103938.0 ) {
  if ( num_indices <= 19965.5 ) {
    p.policy = seq_exec;
   } else {
    p.policy = omp_exec;
   }
   // rest of tree

 // Write predicted model parameters to the blackboard.
 RAJA::apollo::set_model_params(p);
}
```

Again, Apollo uses the models learned offline to dynamically tune application parameters. Learning offline allows us to amortize the cost of modeling over a number of runs, and remove the cost of constructing the model from the runtime overhead of the framework. In Section IV, we show how these models can speed up all three of our target applications.

IV. Results

To evaluate Apollo's tuning capabilities, we apply it to the three applications described in Section II-C: LULESH, CleverLeaf, and ARES. All three of these applications exhibit input-dependent performance characteristics, and both CleverLeaf and ARES use adaptive mesh refinement (AMR).

To examine input-dependence of our results, we tested our applications on a range of input problems. We ran the

Sedov blastwave problem [22], a common hydrodynamics test problem, in all three applications. In CleverLeaf we also simulate Sod's shock tube problem [23], and a version of the triple point shock interaction problem presented in [24]. In this problem, a shock generates a large amount of vorticity and creates a complex area of interest which will be covered with a large number of subdomain patches by the AMR algorithm.

In ARES, the Sedov blastwave problem is simulated with a full mixed material capability. This adds an additional layer of input-dependent behavior to a code since additional logic is required to correctly simulate this mixed cell. The *Jet* problem is a simple shaped charge deck, and the *Hotspot* problem simulates the ignition of an inertial confinement fusion capsule. Both these problems use multiple materials and additional physics simulation capability enabled. This additional physics capability requires a different layout of simulation data in memory and presents more complex patterns of mixed cells throughout the simulation domain.

a) Training Data: In all three applications, we run each problem configuration at a range of global problem sizes, capturing the range of computational performance attributed to hardware specifications such as cache size. For each problem and size combination we perform a single run for each possible parameter value, giving a total of 46 executions per problem and size combination. The parameter values are: {OpenMP, Sequential} for policies, and {1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024} for chunk sizes. Each training run was performed on a single, dedicated node. The training runs we performed generated 1.7 GB of data, containing over 4.7 million samples. The total duration of all collected training samples (kernel executions) was 3 hours and 12 minutes, collected in under 5 hours. All experiments were performed on a commodity cluster system at Lawrence Livermore National Laboratory. Each node contained two Intel E5-2670 "Sandy Bridge" CPUs running at 2.6 GHz, and 32 GB of RAM with a peak memory bandwidth of 51.2 GB/s.

A. Model Accuracy Analysis

For our first experiments, we evaluate the accuracy of our decision tree models for tuning two parameters: execution policy and OpenMP chunk size, measuring how often the models pick the *fastest* parameter value. We build a single model that takes as input all available features for each application and parameter combination. The model is trained to make decisions *regardless* of the input problem – that is, we did not use any features specific to a particular input deck. We use 10-fold cross validation, where the input cases are divided into 10 equal sets, with 10 models created using rotating groups of 9 sets as training data, and testing our classifier on the tenth set. Reported scores are the mean accuracy of the ten models. Table II shows the results for each application. Apollo selects the best parallel execution model (RAJA execution policy) 92% of the time, on average, across all three applications and problem configurations. The chunk size models are significantly less accurate; as low as 21% in the case of CleverLeaf.

Application	Execution Policy	Chunk Size
LULESH	98%	38%
CleverLeaf	92%	21%
ARES	96%	36%

TABLE II: Model accuracy: the percentage of time that Apollo selected the correct execution policy or chunk size.

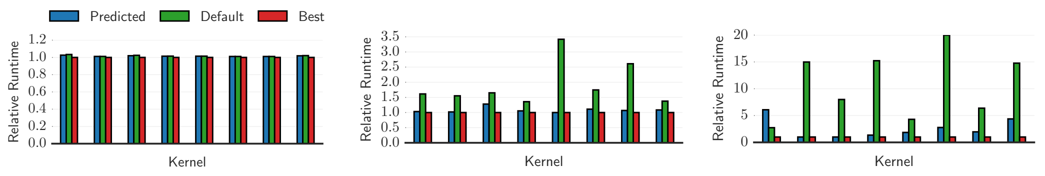
We can also look at the runtime of the predicted parameter values compared to the best possible choice, helping us quantify the case where the model might make an incorrect choice, but pick the second best value. In this case, the accuracy of the model would be penalized, although the overall runtime of the set of parameter values selected might remain close to the best. Figure 6 contains the relative speedups for each kernel when dynamically tuning the execution policy values using the model, compared to the best possible choices, and the default static selection of OpenMP everywhere. Figure 7 shows the relative speedups for each kernel when tuning the OpenMP chunk size, compared to a default static selection of 128.

The runtimes of the predicted policies are close to best, matching the statistical accuracy of the model. For all three applications, using the predicted policies results in a mean speedup when considering all kernels. For both LULESH and CleverLeaf, the predicted policies always beat the default configuration for the 8 most variable kernels in both applications. In ARES, one kernel shows a slowdown when using the predicted policy, but the remaining 7 most variable kernels all show a significant speedup. The runtimes of the predicted chunk sizes are close to the best for LULESH and CleverLeaf, despite the model only picking the fastest size 36% and 22% of the time respectively. This result shows that even in the case of an incorrect decision, the models often pick parameter values that perform well. In the case of ARES, the predicted chunk sizes are far worse that the default or best in most cases. We present this result to show how our approach can be applied to multiple parameters, but will focus the remainder of the paper exclusively on the more accurate models for tuning execution policy.

B. Tuning Model Reduction

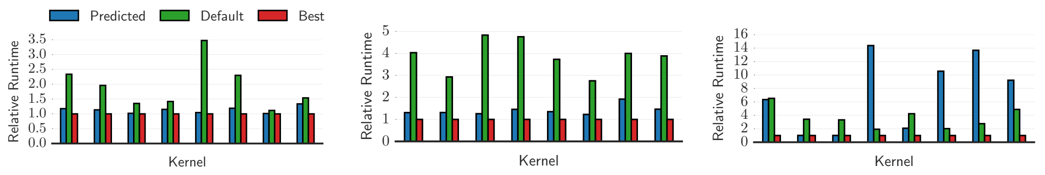
Apollo can use a large number of features to make run-time tuning decisions, but this requires us to measure each input feature, and doing so can be costly. Decision tree models can be reduced to focus on the most important features by simply removing deeper levels of the tree. This reduces the accuracy of the model, but it also decreases the measurement cost of each decision. In this section we analyze the impact of decision trees constructed using a reduced subset of features, and a reduced decision tree depth.

Figure 8 show the normalized importance of the top 5 features for each application. The number of indices and timestep are important in all applications. The problem name is also an effective feature in CleverLeaf and ARES, highlighting the input-dependent nature of the decisions. We also see instruction count features appearing. The `movsd` feature is a scalar load, showing the impact of a kernels memory demands on the execution policy choice.



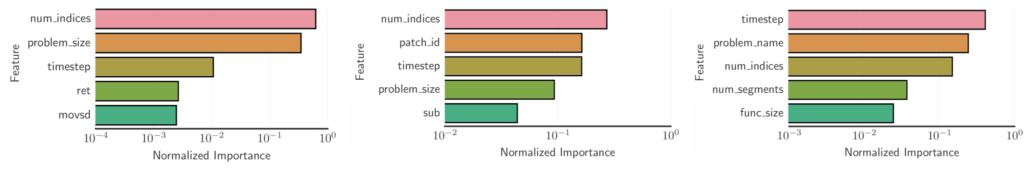
(a) LULESH. (b) CleverLeaf. (c) ARES.

Fig. 6: Relative runtimes of predicted execution policies compared to the best possible combination and a static choice of OpenMP for the eight kernels that consume the most time in each application.



(a) LULESH. (b) CleverLeaf. (c) ARES.

Fig. 7: Relative runtimes of predicted chunk sizes compared to the best possible combination and a default static choice of 128 for the eight kernels that consume the most time in each application.



(a) LULESH. (b) CleverLeaf. (c) ARES.

Fig. 8: Normalized importance of the top 5 features in the models generated for LULESH, Cleverleaf and ARES.

To examine the impact of the features used in the models on their accuracy, we build and evaluate models using subsets of up to ten of the most important features identified in Figure 8. We can see in Figure 9 that accuracy is stabilized when using 4 features, with accuracy scores nearing those achieved using all features.

The second aspect we consider when generating lightweight, responsive models for use at runtime is the depth of the decision tree used, which directly corresponds to model complexity. Using the five most important features for every model, Figure 10 shows the accuracy of each model at a range of decision tree depths (1 through 25). We can see that using the top five features and a tree depth of 15 produces models whose accuracy for LULESH and ARES is within 0.1% of the models using all available features, and for CleverLeaf, is within 8% of the model using all available features.

C. Online Auto-Tuning with Generated Models

To showcase how our technique can speed up the execution of real kernels, we built tuning models for all three applications, and tested them on all input problem configurations. In this section we present the speedups we achieved by dynamically selecting statically optimized kernel variants. The decision models are evaluated every time a RAJA kernel is invoked, and the selected parameter will be set to the value predicted by the model. In the previous section we determined a lightweight model configuration with fewer features (5) and reduced tree depth (15), and we use that configuration to generate the models we use here. Note that for each application, the same model is re-used across input decks, and across MPI ranks for the parallel runs of CleverLeaf and ARES.

Since evaluating the model at runtime adds overhead, we perform a comparison between a static parameter choice and the model choice for a range of problem sizes. In LULESH and CleverLeaf, the default execution policy is OpenMP everywhere, as chosen by the developers of the RAJA versions of these applications. ARES is a more complex code, and the developers have manually assigned kernels in the code as being more appropriate for serial or OpenMP execution (regardless of input data), and the default version of ARES uses these policy selections.

Figure 11 shows the speedup with Apollo in all three applications running on a single node. For CleverLeaf, our dynamic tuning is the most successful, speeding up execution by up to 4.8x. Apollo achieves a speedup of 3.36x for LULESH and 1.15x for ARES. The speedup comes mainly from avoiding the overhead of spawning OpenMP regions for very small patches, and our models are able to determine from the patch size and other features when it is faster to run a loop sequentially than to pay for the overhead of using threads.

While we only train our models on single-node runs, we can use our tuning models in larger, multi-process runs. This

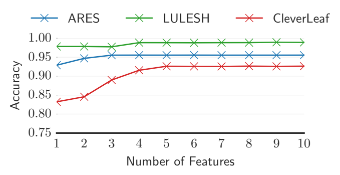


Fig. 9: Model accuracy using subsets of the most important features identified in Figure 8.

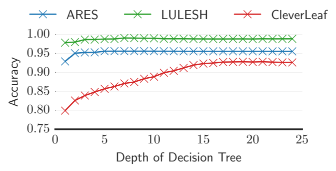


Fig. 10: Model accuracy at various decision tree depths. Each model is built using the 5 most important features, identified in Figure 8.

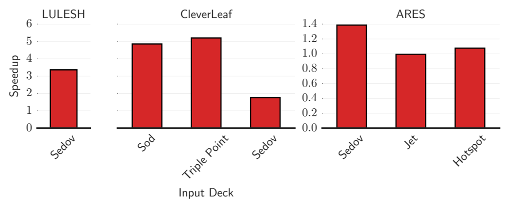
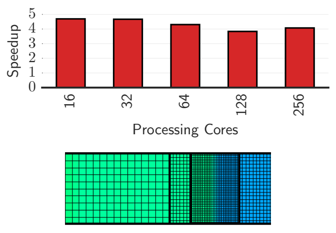
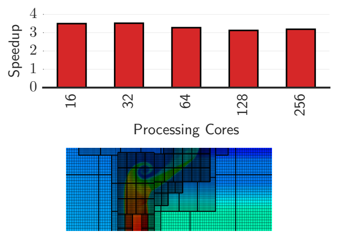


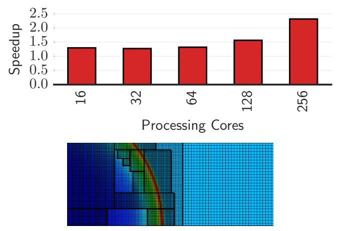
Fig. 11: Speedups when dynamically tuning execution policies in all three applications.



(a) Sod's shock tube.

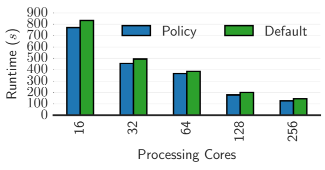


(b) Triple Point interacting blastwaves.

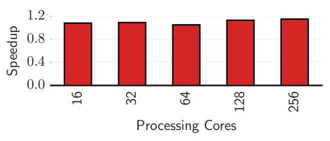


(c) Sedov's blastwave.

Fig. 12: Runtimes and speedups when dynamically tuning parallel runs of CleverLeaf for the three different input problems.



(a) Runtimes.



(b) Speedups.

Fig. 13: Dynamically tuning the ARES *Hotspot* problem in parallel.

		L	C			A		
		Sedov	Sod	Sedov	Triple Pt	Sedov	Jet	Hotspot
L	Sedov	0.99	0.93	0.91	0.72	0.90	0.98	0.69
C	Sod	0.14	0.99	0.91	0.76	0.89	0.97	0.69
	Sedov	0.86	0.86	0.99	0.75	0.90	0.98	0.69
	Triple Pt	0.83	0.86	0.77	0.92	0.89	0.97	0.69
A	Sedov	0.14	0.40	0.29	0.40	0.99	0.81	0.49
	Jet	0.14	0.93	0.91	0.72	0.90	0.99	0.70
	Hotspot	0.14	0.93	0.91	0.72	0.89	0.83	0.78

TABLE III: Accuracy breakdown of dynamically tuning using different training (rows) and test (columns) set combinations for LULESH (L), CleverLeaf (C), and ARES (A).

is important, since CleverLeaf and ARES use adaptive mesh refinement, and while the overall refinement of the mesh is deterministic for our runs, the *sizes* of refined patches created in a distributed run are not the same as those created in a sequential run. Each node in an MPI-parallel run may have a very different workload. This is the kind of input-dependent behavior our models are designed to optimize.

Figure 12 shows the runtimes and speedups of dynamically tuning CleverLeaf for all three input problems. We use a larger initial problem size in each case and we strong scale up to 256 cores. In all cases, tuning with Apollo is significantly faster than using the default RAJA execution policy. Figure 12 also shows a visualization of the mesh configuration and density field for a subset of the problem space at the final time of the simulation. Since the best parameter value is largely dependent on the subdomain size, these visualizations explains some of the speedups provided when dynamically tuning with Apollo's models. For example, the curved shock in the Sedov problem generates many small subdomains which Apollo correctly runs serially, beating the default execution policies by up to 2.3x.

Whilst Figures 12a and 12b show consistent speedups values between 4-5x and 3-4x, the Sedov problem in Figure 12c shows speedups from 1.29x on 16 cores to 2.3x on 256 cores. That is, Apollo gets *more* improvement out of this problem as we strong scale. In strong scaling, the adaptively refined mesh is subdivided into increasingly smaller and smaller subdomains, and Apollo is able to speed up execution for *more* patches at the strong scaling limit. The same trend is visible in Figure 13. Using Apollo, ARES runs from 8% faster on 16 cores up to 15% faster on 256 cores. Note that ARES is a production multi-physics code, and the Hotspot deck exercises many different physics modules. Our speedups with Apollo are measured in wall clock time for the *entire* ARES run, but only a single physics component has been modified to work with RAJA (and therefore Apollo).

D. Cross-Application Predictions

Our tuning models so far have been built and applied to single applications independently. An important aspect of our work is the ability to generate tuning models that can be used to dynamically tune other applications. In Table III we show a first step in this direction, where we evaluate each model on a test set taken from a single application and input problem combination.

From these results we can see that in many cases, Apollo's models are applicable across both applications and input decks. Of particular interest to us is the fact that the models learned from LULESH are effective when applied to CleverLeaf and ARES. LULESH is a mini-app containing less than 40 unique kernels, but the range of training data collected appears to cover the performance space containing the kernels from both CleverLeaf and ARES. Conversely, models produced with CleverLeaf and ARES do not perform well for LULESH. We attribute this drop in accuracy to the narrower range of iteration counts observed in these applications, since each problem configuration was run with fewer global problem sizes.

V. Related Work

The field of auto-tuning comprises many techniques. The overarching goal is to understand the effects of a set of *tuning parameters* on a piece of code, and to select the tuning parameters that minimize the code's running time. Loosely speaking, we can divide auto-tuning techniques into three categories according to how they select optimal parameters. *Empirical* techniques directly run many variants of a code block and select the fastest based on actual running time. *Analytical* techniques use a (usually human-generated) model to predict the code's run time, and select based on the results of the model. *Statistical* techniques are similar, but they derive models using machine-learning techniques. Table IV contains a sample of existing approaches and their categories, as well as when they tune and cost of each tuning decision.

a) Empirical Techniques: Some of the most successful techniques are tuners for numerical libraries, such as ATLAS [3] and FFTW [4]. These libraries run many small tests to find good tuning parameters for specialized numerical algorithms. Oski [31] similarly tunes sparse linear algebra kernels at *runtime*, and Spiral performs similar optimizations for signal processing [5].

Frameworks like OpenTuner [30] and Orio [29] are general approaches for arbitrary kernels and tuning parameters. They focus on intelligently searching a large, multi-dimensional tuning parameter space. The ActiveHarmony project [25], [6] uses heavily optimized, parallel search algorithms to empirically tune applications on-line. It can adapt dynamically to slowly changing applications, but re-tuning takes several minutes, even with parallel search.

b) Analytical Techniques: Analytical modeling has proven to be a useful in the past as a way to optimize serial code, and recently, analytical modeling techniques have been revived for parallel performance *prediction* [27], [32], [28], [26] Analytical models can be used to predict scaling performance for the application, but they do not directly relate to any tuning parameters and cannot be used for tuning.

c) Statistical Techniques: Empirical tuning is costly because it requires potentially many runs of kernels with different parameters. Rather than running directly, some tools train a model to estimate the runtime of a code block, and they use statistical regression or other machine learning techniques to build their estimators. This approach has been applied to compiler optimizations [33], [34], GPU power modeling [35], [36], concurrency throttling [37], thread mapping [38], and tuning filter banks in signal processing algorithms [26]. Feature importance analysis, has also been used to diagnose the causes of performance variation in communication-bound codes. [39]. Muralidharan and Ding [9], [10] both use machine learning models, but focus on code variants for input-dependent algorithm choices. Our approach is much finer-grained and makes thousands of decisions during each application timestep, requiring lightweight models compiled to machine code.

d) Apollo: Our approach, Apollo, is a *dynamic*, *low-overhead*, *data-driven* auto-tuning framework. Like ActiveHarmony, Apollo tunes codes on-line, but it uses off-line training to build statistical classifiers that *directly* select values for tuning parameters. This approach allows Apollo to respond quickly to changes in the input data that kernels process. We can recognize, for instance, that particular runtime policies work best for particular array sizes or data properties. Our approach also allows Apollo models to execute quickly. Prior papers present results from benchmarks [9], [10], but Apollo tunes over 500 kernels in a production multi-physics application with real inputs.

VI. Conclusions

In this paper, we introduced Apollo, a lightweight auto-tuning framework for adaptive, input-dependent codes. Rather than using costly on-line search, Apollo uses off-line training to generate pluggable tuning models that can be used at run-time. Apollo allows users to preserve the benefits of statically optimized C++ templates, while still allowing dynamic selection of execution models. While Apollo is implemented in RAJA, the techniques for separating the concerns of implementation and tuning are general, and we plan to apply these techniques to other performance portability frameworks. Apollo is designed so that decision models can be re-trained and re-loaded over time without rebuilding production applications. We showed that Apollo can generate auto-tuners that select the fastest execution policy 98% of the time and achieve speedups of 1.2x-4.8x for adaptive proxy applications *and* a multi-million line production multi-physics code.

Acknowledgements

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-XXXXXX).

Software

Our model generation and analysis framework, Apollo, is available on GitHub at `https://github.com/LLNL/Apollo`.

References

[1] R. D. Hornung and J. A. Keasler, "The RAJA Poratability Layer: Overview and Status," Lawrence Livermore National Laboratory, Tech. Rep. LLNL-TR-661403, Sep. 2014.

[2] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, pp. 3202–3216, Dec. 2014.

Package & Domain	Model	Tuning Style	peed	Technique
ActiveHarmony (application kernels) [6], [25]	Empirical	Dynamic (run-time)	Slow	Search
Apollo (application kernels)	Statistical	Dynamic (run-time)	Fast	Classifier
ATLAS (dense linear algebra) [3]	Empirical	Static (off-line)	Fast	Search
Bergstra, et al. (image filters) [26]	Statistical	Static (off-line)	Fast	Search
Calotoiu, et al. (MPI scaling) [27]	Analytical	Dynamic (run-time)	N/A	N/A
FFTW (FFT) [4]	Empirical	Static (off-line)	Slow	Search
Hoefler, et al. (application runtime) [28]	Analytical	Dynamic (run-time)	N/A	N/A
Orio (application kernels) [29]	Empirical	Static (off-line)	Slow	Search
OpenTuner (application kernels) [30]	Empirical	Static (off-line)	Slow	Search
Oski (sparse linear algebra)[31]	Empirical	Dynamic (run-time)	Slow	Search
PEMOGEN (application kernels)[32]	Analytical	Dynamic (run-time)	N/A	N/A
Nitro (code variants)[9]	Statistical	Dynamic (run-time)	Slow	Classifier
Ding, et al. (code variants)[10]	Statistical	Dynamic (run-time)	Slow	Classifier

TABLE IV: Sample of auto-tuning and modeling techniques using empirical, analytical and statistical (machine learning) approaches.

[3] R. Clint Whaley, A. Petitet, and J. J. Dongarra, "Automated empirical optimizations of software and the ATLAS project," *Parallel Computing*, vol. 27, no. 1-2, pp. 3–35, Jan. 2001.

[4] M. Frigo and S. G. Johnson, "The Design and Implementation of FFTW3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005.

[5] M. Püschel *et al.*, "Spiral: A Generator for Platform-Adapted Libraries of Signal Processing Alogorithms," *International Journal of High Performance Computing Applications*, vol. 18, no. 1, pp. 21–45, Feb. 2004.

[6] C. Tapus, I.-H. Chung, and J. K. Hollingsworth, "Active Harmony: Towards Automated Performance Tuning," in *Supercomputing 2002 (SC'02)*, Nov. 2002, p. 44.

[7] I.-H. Chung and J. K. Hollingsworth, "Using Information from Prior Runs to Improve Automated Tuning Systems," in *Supercomputing 2004 (SC'04)*, Nov. 2004, pp. 30–30.

[8] ——, "A Case Study Using Automatic Performance Tuning for Large-Scale Scientific Programs," *Proceedings of the 15th IEEE International Symposium on High Performance Distributed Computing*, pp. 45–56, Jun. 2006.

[9] S. Muralidharan, M. Shantharam, M. Hall, M. Garland, and B. Catanzaro, "Nitro: A Framework for Adaptive Code Variant Tuning," in *Proceedings of the IEEE International Symposium on Parallel & Distributed Processing*, May 2014, pp. 501–512.

[10] Y. Ding, J. Ansel, K. Veeramachaneni, X. Shen, U.-M. O'Reilly, and S. Amarasinghe, "Autotuning algorithmic choice for input sensitivity," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15)*, Jun. 2015, pp. 379–390.

[11] I. Karlin *et al.*, "Exploring Traditional and Emerging Parallel Programming Models Using a Proxy Application," in *Proceedings of the 27th IEEE International Symposium on Parallel & Distributed Processing*, May 2013, pp. 919–932.

[12] D. A. Beckingsale, W. Gaudin, A. Herdman, and S. Jarvis, "Resident Block-Structured Adaptive Mesh Refinement on Thousands of Graphics Processing Units," in *Proceedings of the 44th International Conference on Parallel Processing*, Aug. 2015, pp. 61–70.

[13] R. Darlington, T. McAbee, and G. Rodrigue, "A Study of ALE Simulations of Rayleigh-Taylor Instability," *Computer Physics Communications*, vol. 135, pp. 58–73, 2001.

[14] B. E. Morgan and J. A. Greenough, "Large-Eddy and Unsteady RANS Simulations of a Shock-Accelerated Heavy Gas Cylinder," *Shock Waves*, Apr. 2015.

[15] Dyninst. [Online]. Available: http://www.dyninst.org

[16] Caliper. [Online]. Available: https://github.com/LLNL/Caliper

[17] J. Quinlan, "Simplifying decision trees," *International Journal of Man-Machine Studies*, vol. 27, no. 3, pp. 221 – 234, 1987.

[18] P. E. Utgoff, "Incremental induction of decision trees," *Machine Learning*, vol. 4, no. 2, pp. 161–186, 1989.

[19] pandas: Python Data Analysis Library. [Online]. Available: http://pandas.pydata.org

[20] NumPy. [Online]. Available: http://www.numpy.org

[21] F. Pedregosa *et al.*, "Scikit-learn: Machine Learning in Python," *The Journal of Machine Learning Research*, vol. 12, Feb. 2011.

[22] L. I. Sedov, "Propagation of strong shock waves," *Journal of Applied Mathematics and Mechanics*, vol. 10, pp. 241–250, 1946.

[23] G. A. Sod, "A Survey of Several Finite Difference Methods for Systems of Nonlinear Hyperbolic Conservation Laws," *Journal of Computational Physics*, vol. 27, no. 1, pp. 1–31, Apr. 1978.

[24] S. Galera, P.-H. Maire, and J. Breil, "A two-dimensional unstructured cell-centered multi-material ALE scheme using VOF interface reconstruction," *Journal of Computational Physics*, vol. 229, no. 16, pp. 5755–5787, Aug. 2010.

[25] J. K. Hollingsworth and P. J. Keleher, "Prediction and adaptation in Active Harmony," in *Proceedings of the 7th International Symposium on High Performance Distributed Computing*, Jul. 1998, pp. 180–188.

[26] J. Bergstra, N. Pinto, and D. Cox, "Machine learning for predictive auto-tuning with boosted regression trees," in *Proceedings of Innovative Parallel Computing*, May 2012, pp. 1–9.

[27] A. Calotoiu, T. Hoefler, M. Poke, and F. Wolf, "Using automated performance modeling to find scalability bugs in complex codes," in *Supercomputing 2013 (SC'13)*, Nov. 2013, pp. 1–12.

[28] T. Hoefler, W. Gropp, W. Kramer, and M. Snir, "Performance modeling for systematic performance tuning," in *Supercomputing 2011 (SC'11)*, 2011, pp. 1–12.

[29] A. Hartono, B. Norris, and P. Sadayappan, "Annotation-based empirical performance tuning using Orio," in *IEEE International Symposium on Parallel & Distributed Processing*, May 2009, pp. 1–11.

[30] J. Ansel *et al.*, "OpenTuner," in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation Techniques*, New York, New York, USA, 2014, pp. 303–316.

[31] R. Vuduc, J. W. Demmel, and K. A. Yelick, "OSKI: A library of automatically tuned sparse matrix kernels," *Journal of Physics: Conference Series*, vol. 16, no. 1, pp. 521–530, Aug. 2005.

[32] A. Bhattacharyya and T. Hoefler, "PEMOGEN: Automatic Adaptive Performance Modeling during Program Runtime," in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation Techniques*, Aug. 2014, pp. 393–404.

[33] E. Park, J. Cavazos, L. N. Pouchet, and C. Bastoul, "Predictive modeling in a polyhedral optimization space," in *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, Apr. 2011, pp. 119–129.

[34] F. Agakov *et al.*, "Using Machine Learning to Focus Iterative Optimization," in *Proceedings of the International Symposium on Code Generation and Optimization*, Mar. 2006, pp. 295–305.

[35] G. Wu, J. L. Greathouse, A. Lyashevsky, N. Jayasena, and D. Chiou, "GPGPU performance and power estimation using machine learning," in *Proceedings of the 21st IEEE International Symposium on High Performance Computer Architecture*, Feb. 2015, pp. 564–576.

[36] S. Song, C. Su, and B. Rountree, "A simplified and accurate model of power-performance efficiency on emergent GPU architectures," in *Proceedings of the 27th IEEE International Symposium on Parallel & Distributed Processing*, May 2013, pp. 673–686.

[37] M. A. Curtis-Maury *et al.*, "Identifying energy-efficient concurrency levels using machine learning," in *Proceedings of the IEEE Conference on Cluster Computing*, Sep. 2007, pp. 488–495.

[38] C. Su, D. Li, D. S. Nikolopoulos, K. W. Cameron, B. R. de Supinski, and E. A. Leon, "Model-based, memory-centric performance and power optimization on NUMA multiprocessors," in *Proceedings of the IEEE International Symposium on Workload Characterization*, Nov. 2012, pp. 164–173.

[39] N. Jain, A. Bhatele, M. P. Robson, T. Gamblin, and L. V. Kale, "Predicting application performance using supervised learning on communication features," in *Supercomputing 2013 (SC'13)*, Nov. 2013, pp. 1–12.