SOFTWARE: PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2015; **45**:1659–1676
Published online 18 December 2014 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/spe.2307

Selecting linear algebra kernel composition using response time prediction

Aurélie Hurault[1,*,†], Kyungim Baek[2] and Henri Casanova[2]

[1]*IRIT, Université de Toulouse, France*
[2]*Information and Computer Sciences Department, University of Hawai'i at Mānoa, USA*

SUMMARY

Numerical linear algebra libraries provide many kernels that can be composed to perform complex computations. For a given computation, there is typically a large number of functionally equivalent kernel compositions. Some of these compositions achieve better response times than others for particular data and when executed on a particular computer architecture. Previous research provides methods to enumerate (a subset of) these kernel compositions. In this work, we study the problem of determining the composition that yields the lowest response time. Our approach is based on a response time prediction for each candidate combination. While this prediction could in principle be obtained using analytical and/or empirical performance models, developing accurate such models is known to be challenging. Instead, we define a feature space that captures salient properties of kernel combinations and predict response time using supervised machine learning. We experiment with a standard set of machine learning algorithms and identify an effective algorithm for our kernel composition selection problem. Using this algorithm, our approach widely outperforms the strategy that would consist in always using the simplest kernel composition and is often close to the fastest kernel compositions among those evaluated. We quantify the potential benefit of our approach if it were to be implemented as part of an interactive computational tool. We find that although the potential benefit is substantial, a limiting factor is the kernel composition enumeration overhead. Copyright ©2014 John Wiley & Sons, Ltd.

Received 11 April 2014; Revised 20 November 2014; Accepted 20 November 2014

KEY WORDS: numerical linear algebra; kernel compositions; performance prediction; machine learning

1. INTRODUCTION

Numerical linear algebra is fundamental in virtually all fields of science and engineering (e.g., for solving linear systems of equations, eigenvalue problems, and singular value problems). Consequently, much effort has been put into developing efficient numerical linear algebra libraries. Two such libraries have emerged as de facto standards: Basic Linear Algebra Subprograms (BLAS) [1] and LAPACK [2]. Implementations have been developed over the years for sequential execution but also for parallel execution on distributed memory platforms [3] as well as multicore and/or graphics processing unit (GPU) processors [4]. Regardless of the execution model, these libraries implement fundamental compute kernels that serve as building blocks for performing complex computations. Because of the properties of linear algebra operations, including commutativity and associativity, kernels can be composed in various ways to obtain a desired numerical result. A well-known example is the use of associativity for computing matrix products: $(A \times B) \times C = A \times (B \times C)$. Several kernels are available that combine multiple operations (e.g., a matrix multiplication and a matrix addition), making it possible to use different numbers of kernels for the same computation. In general, for a given computation, there can be a large or even unbounded number of functionally

*Correspondence to: Aurélie Hurault, IRIT - ENSEEIHT, 2 rue Camichel, B.P. 7122, F-31071 Toulouse Cedex 7, France.
†E-mail: Aurelie.Hurault@enseeiht.fr

equivalent kernel compositions. The motivation for this work is that these compositions are not all equivalent in terms of computational performance. Differences stem from the number of kernels used, the order in which the kernels are composed, the quality of the kernel implementations in the libraries, the dimensions of the matrices, and the compute hardware. Some cases are well studied, and techniques have been developed to determine the best kernel composition. For instance, the matrix multiplication associativity problem given as an example earlier can be solved optimally using a famous dynamic programming approach [5, Chapter 15]. But in general, the problem of determining for a given computation which composition of which kernels, among all valid composition options, is best for a given computer is difficult.

In this work, we seek to answer the following question: given a computation that can be carried out via calls to standard numerical linear algebra kernels, is it possible to determine automatically a kernel composition that achieves high performance? We define performance as the *response time*, that is, the time to perform the computation. We study this question assuming single-core sequential execution on a single given computer and find that even this seemingly simple setting gives rise to several challenges. Our approach consists in two phases: (i) *enumeration* of (a bounded set of) functionally equivalent kernel compositions that implement the desired computation and (ii) *selection* of the fastest of these compositions based on a response time prediction. The goal is to perform both the enumeration and the selection at runtime, for instance, as part of an interactive computing environment. Solving the enumeration problem requires a formal description of the kernels and of their properties. This description is used to reason about valid kernel compositions in a view to a full enumeration. We solve the enumeration problem by directly reusing the approach developed in [6, 7] and instead focus our efforts on the selection problem. One possible approach for selection is to develop analytical performance models for each kernel and for their compositions. However, given the complexity of modern software and hardware, accurate analytical performance models are elusive at best, even for single-core execution. Instead, we predict response time using supervised machine learning techniques based on a set of nonexhaustive benchmark executions of kernels and kernel compositions. Our goal in this work is not to make advances in the area of machine learning but rather to investigate whether well-established machine learning algorithms can be used for solving the kernel composition selection problem. More specifically, our contributions are as follows:

- We give a formulation of the kernel composition selection problem that is amenable to solution via machine learning techniques;
- We experiment with standard machine learning algorithms and identify algorithms that can be used effectively for solving the selection problem.
- We propose feature space definitions that capture important characteristics of kernel compositions and produce good results in response time prediction.
- We present experimental results that show the benefit of our approach over an approach that always uses the simplest kernel composition. We also quantify the performance loss when compared with the fastest kernel compositions among all evaluated compositions.
- We quantify the potential benefit of our approach when implemented as part of an interactive computing environment, identifying limiting factors and discussing possible improvements.

The rest of this paper is organized as follows. Section 2 provides background information on the enumeration and selection problems. Section 3 reviews related work. Section 4 presents our feature space construction. Section 5 describes the methodology we use to evaluate our proposed approach. Experimental results are provided in Section 6. Section 7 discusses practical implications of our results in the context of an interactive computing environment. Section 8 concludes with a summary of our results and perspectives on future work.

2. BACKGROUND AND PROBLEM STATEMENT

In this work, we target numerical linear algebra computations to be performed with the BLAS and LAPACK libraries. We consider that a user submits a *request* for a particular computation,

DOI: 10.1002/spe

say, in a Matlab-like syntax. For instance, the request '`eig(A * B)`' may correspond to computing the eigenvalues of the product of two double precision matrices. Such a request is to be answered by invoking *kernels* provided by BLAS and LAPACK. For instance, in BLAS, the general double precision matrix multiplication kernel is called `dgemm`, and in LAPACK, the general double precision eigensolver kernel is called `dgeev`. Therefore, one *solution* to the aforementioned request is to invoke `dgemm` followed by `dgeev`. In general, the number of solutions for a given request is unbounded. In practice, only a finite number of solutions are sensible as some solutions could involve idempotent operations such as multiplication with the identity matrix or inverting the inverse of a matrix. Nevertheless, for complex computations, the number of sensible solutions can be large.

The *enumeration* problem introduced in Section 1 consists in identifying all (sensible) solutions for a given request. We solve this problem using the approach in [6] by which requests and kernels are described as terms over an order-sorted algebra [8]. A domain expert provides the algebra as a formal description of fundamental operators and of equational axioms and also provides a formal description of the kernels. A set of solutions for a request is then determined using equational unification [9]. Equational unification consists in finding, for two terms, a set of symbol substitutions that when applied makes both terms provably equal modulo equational axioms. The request is then unified with all available kernels. Let us illustrate this process on a linear algebra example. Considering only the addition (+) and multiplication ($\times$) operators on scalars and matrices, a small subset of the axioms of linear algebra is $E = \{1 \times x = x; 0 \times x = 0; x + 0 = x\}$, where 1 and 0 are overloaded for both scalars and matrices. Consider the following formalization of a matrix multiplication operator: $mult(\alpha, a, b, \beta, c) = \alpha \times a \times b + \beta \times c$. For the sake of simplicity, in this example, we omit typing information. Consider now the request $x * y$, for which we must find a solution. One of the solutions that can be obtained using equational unification is derived from the following substitution: $\sigma = \{\alpha \to 1, a \to x, b \to y, \beta \to 0, c \to 0\}$. We obtain the solution itself: $mult(1, x, y, 0, 0)$. This example is simplistic, and in general, a solution involves the composition of multiple operators. These compositions are automatically determined by nested invocations of the unification procedure on terms that appear in the right-hand side of substitutions.

The *trader* component implemented in [6] computes solutions for a given request as described earlier given an arbitrary order-sorted algebra and kernel definitions. The authors have defined the algebra and the kernels for the BLAS library and a subset of the LAPACK library. While we refer the reader to [6] for all details on the equational unification algorithm and its implementation, we provide here an example to illustrate how the trader operates. Consider the following request assuming all matrices are square of dimension $n \times n$:

$$((A \times B)^t)^{-1} \times C + D \times C \ .$$

When providing this request as input to the trader, it returns multiple solutions including

- $((A \times B)^t)^{-1} \times C + D \times C$
- $((A \times B)^{-1})^t \times C + D \times C$
- $D \times C + ((A \times B)^{-1})^t \times C$
- $(((A \times B)^{-1})^t + D) \times C$
- $(((A \times B)^t)^{-1} + D) \times C$
- ...

It turns out that the solutions that most closely resemble the request itself are not the fastest ones in this example because they involve more operators. Furthermore, because the BLAS matrix multiplication kernel, `dgemm`, can do transposition 'for free', the order of the transposition and inversion operators matters. This example uses symbolic notations for showing solutions returned by the trader. But in fact, the trader returns actual sequences/compositions of library kernel invocations with specified input parameters. For instance, a partial solution $D \times C$ would actually be returned as

DOI: 10.1002/spe

```
dgemm('N','N',n,n,n,1.0,D,n,C,n,1.0,Temp,n)
```

where `Temp` holds the result of the matrix multiplication (we refer readers unfamiliar with the BLAS kernel signatures to the BLAS documentation [10]). For the solutions for the aforementioned example, the trader returns sequences of such library kernel invocations along with array declarations for holding intermediate and final results of the computation.

In this work, we reuse the trader as is to solve the enumeration problem. The equational unification problem is known to have high computational complexity and, furthermore, can lead to an unbounded number of solutions. When enumerating for solutions, we provide the trader with a bound on the enumeration time as well as a bound on the number of solutions needed. For a given solution s, let $numEq(s)$ and $numComp(s)$ denote the number of algebraic equations and the number of nested kernel compositions, respectively, used by the equational unification procedure to obtain s. Informally, the smaller $numEq(s)$ and/or $numComp(s)$, the simpler the solution. We configure the trader so that solutions are returned sorted by increasing $numEq(s) + numComp(s)$ value, meaning that simpler solutions are returned first.

Our goal is to solve the following selection problem: among a set of solutions returned by the trader for a given request, determine which solution leads to the lowest response time when executed on a target computer.

3. RELATED WORK

The trader component in [6] uses equational unification to derive multiple solutions for a particular linear algebra computation. Other approaches could be used instead. One option is to use ontologies for deriving appropriate kernel compositions [11]. Another option is to use rewriting techniques, by which one derives a normal form of a particular expression as a deterministic composition of kernels (see, for instance, the work in [12]). One challenge with this approach is that rewriting rules must possess particular properties, making the definition of appropriate rules potentially challenging. Furthermore, the approach would need to be adapted to yield multiple composition options for a single expression. Yet another option that could be used in the context of this work instead of equational unification is a compiler-based approach, making a direct analogy between produced 'programs' and our produced 'solutions'. In [13], an abstract program with precondition and postcondition is transformed into multiple instantiated programs, each corresponding to the result of a search over compositions of kernels specified in a glossary. The compiler then generates a set of instantiated programs with identical functionality but various performance characteristics.

We study the question of how to pick the combination of numerical kernels that produces the shortest response time on a target computer among a set of functionally equivalent compositions. This question can be answered easily if accurate performance models (i.e., analytical expressions of kernel response times on the target computer) are available. Unfortunately, deriving accurate performance models is known to be challenging because of software complexity (e.g., control flow and compiler optimization) and hardware complexity (e.g., out of order instruction execution, instruction-level parallelism, memory hierarchy, multicore, and hardware accelerators). Given the difficulties involved, developing an accurate analytical model for a kernel is painstaking, and the obtained model is often limited. For instance, in [14], the authors develop a performance model for one BLAS kernel. This model is derived based on an in-depth analysis of the kernel's implementation and leads to reasonable results with relative errors of a few percent. The model, while impressive, is only applicable in a limited range of conditions (no L2-cache misses, no Translation Lookaside Buffer misses, no page faults, and no multicore execution). Given these difficulties, many authors have explored instead empirical performance modeling approaches [15–19]. These approaches consist in automatically discovering salient application characteristics using one or a combination of several techniques (e.g., instrumented application execution to obtain trace data [18], analysis of application trace data and of known benchmark trace data to infer application performance [15, 17, 19], analysis of application simulation driven by trace data [17], and simulation driven by static code analysis [16]). These automatically derived empirical models explicitly capture

DOI: 10.1002/spe

important characteristics of the implementation of the target application (e.g., memory accesses, branch behavior, and floating point unit usage). Such level of details makes it possible to develop models for complex and diverse applications and to obtain performance predictions for a target computer different than the computer used to derive the model, with various degrees of accuracy.

Many projects have proposed using offline automatic tuning of linear algebra libraries: ATLAS [20] for a subset of LAPACK, OSKI [21] for sparse linear algebra, PHiPAC [22] for matrices multiplication, libflame [23] for dense linear algebra, MAGMA [24] for linear algebra on GPU, and so on. In these works, upon installation of the library on target computer, an extensive set of benchmarks are performed to search the space of the possible implementations of each kernel for an implementation that achieves low response time. Upon completion of this search, which typically requires at least several hours, an implementation is finalized, compiled, and usable for the target computer. This 'autotuner' approach has been successful because the complexity of computer architectures and of the kernels themselves makes it difficult for a human or a compiler to reason about the performance implications of (a large combination of) implementation options. OpenTuner [25] is a project that automatically generates autotuners for arbitrary domains given a specification of the search space.

Our work is orthogonal with these projects because instead of seeking fast implementations of the kernels in an offline manner, we seek fast compositions of existing kernels in an online manner. In fact, our approach is completely agnostic to the actual kernel implementations because the trader can generate compositions from an arbitrary set of kernel implementations. Therefore, our approach can work with any automatically tuned linear algebra libraries or a set of them. For instance, in an earlier phase of this work, we used the automatically tuned ATLAS library for our experimental evaluations instead of OpenBLAS (and obtained similar results).

Projects such as Orio [26] or the work in [27] rely on the compiler to perform automatic tuning. In a first phase, multiple functionally equivalent versions of a program are generated and empirically evaluated to select the version with the lowest response time. Conceivably, these approaches could be used to select among different kernel composition options. However, a major difference with our work is that we use learning algorithms to pick the fastest composition of kernels based on a set of offline benchmarks for different compositions. This is because our goal is to be able to pick the fastest composition online. In other words, when a user submits a request, we must find a fast solution quickly without performing any benchmarking.

In this work, we focus on a single computer and on single-core linear algebra kernels. In this setting, it is possible to construct reasonable empirical performance models by considering a kernel as a black box. The work in [28, 29] develops black box models of BLAS and LAPACK kernels, each kernel producing a response time for a given set of flag arguments and a given set of matrix dimensions. Response times are modeled as piecewise polynomials of matrix sizes for each given set of flag parameter instantiations. The obtained models are shown to be sufficiently accurate to make it possible to select the best alternative among multiple functionally equivalent compositions of kernels. Given the successful results in [28, 29], we also view kernels as black boxes in this work. But instead of constructing explicit performance models, we use machine learning techniques that predict response time based on a set of benchmark kernel executions. Like [28, 29], our work assumes a single target computer. The advantage of using machine learning algorithms is that our approach, if successful, should be straightforward to extend to other kernels or other kernel implementations, for instance, multicore implementations, GPU implementations, and/or parallel implementations. Rather than attempting to construct empirical performance models for these implementations, implicit performance models can be learned.

Machine learning has been used by several authors in the context of performance prediction. For instance, the work in [30] uses artificial neural networks (ANNs) for predicting the performance of parallel scientific applications. Other works use ANNs [31] or tree-structured predictive models [32] to obtain performance predictions in a view to deriving an efficient application schedule. More directly related to our work is the study of Thomas *et al.* [33] in which the authors use a decision tree, an ANN, and a naïve Bayes classifier to select one algorithm among a finite set of known algorithms for a given problem (e.g., choosing one of several classical matrix multiplication algorithms) and to determine an instantiation of the algorithm's parameters (e.g., block size). In this

DOI: 10.1002/spe

work, we also have a finite set of algorithms or kernels. However, we select among multiple arbitrary *compositions* of these kernels that correspond to arbitrary requests. As seen in our results, capturing the effect of kernel composition on performance is both necessary and nontrivial. While we focus on scientific computing, machine learning techniques for performance prediction have been used successfully in other domains. For instance, in [34], the authors use a variation of principle component analysis to predict the response time and other runtime characteristics of database requests written in SQL. More recently, in [35], the authors use regression and support vector machine techniques to predict the performance of SPARQL queries. Our approach could be applied to these domains for enumerating multiple semantically equivalent database queries selecting the one that leads to the lowest predicted response time.

4. FEATURE SPACE CONSTRUCTION

In this work, we study the feasibility of using machine learning algorithms to solve the selection problem defined in Section 2, namely, among several solutions for a given request select the one that leads to the shortest response time. Given the difficulties of accurate performance modeling, machine learning is attractive because it makes performance modeling both automatic and implicit.

The first step to address this problem is to define an effective representation of a solution to be used as an input to the learning algorithm. In this section, we propose feature space definitions that capture those characteristics of a solution that are the main drivers of response time. More precisely, we map a solution (i.e., a set of kernels with various input flags and the set of matrix dimensions for the input our output matrices) to a vector. A set of such vectors along with corresponding response time measurements are used to train a machine learning algorithm, which can then produce a response time prediction for an arbitrary vector. A natural attempt for defining the feature space would be similar to that used, for instance, in [34] in the context of relational database query optimization. It consists in capturing both the kernel counts and the amount of work carried out by each kernel, which is commensurate to matrix dimensions. More specifically, in that approach, the feature vector has one component for each possible kernel and that component's value is the number of occurrences of that kernel in the solution. In addition, the feature vector has three components that give the minimum, maximum, and average matrix dimensions that are given as input to or produced by the kernels.

One problem with this approach is that it fails to capture well-known pathological cases for linear algebra operations. Consider, for instance, the multiplication of three matrices: $A \times B \times C$. Unless the matrices are square, one of the two possible solutions, $(A \times B) \times C$ or $A \times (B \times C)$, leads to shorter response time than the other. On the computer described in Section 5.1, for three matrices A, B, and C with respective dimensions $1 \times 10,000$, $10,000 \times 1$, and $1 \times 10,000$, the time to compute $A \times (B \times C)$ is more than 7000 times larger than the time to compute $(A \times B) \times C$. Yet, with the aforementioned feature space definition, both solutions map to the same feature vector (two occurrences of the matrix multiplication kernel, and the minimum, maximum, and average of the dimensions of matrices A, B, and C).

The well-known issue of kernel associativity seen earlier in the case of matrix multiplications arises for other kernel compositions. We propose the following pragmatic approach to resolve this issue when defining our feature spaces. We compute the average matrix dimension based on the matrix dimension arguments as they appear in the text of the solution when written using the numerical linear algebra library function signatures, considering all kernels together. For instance, consider the solution $(A \times B) \times C$ with A, B, and C of dimensions $d1 \times d2$, $d2 \times d3$, and $d3 \times d4$, respectively. The text of the solution in terms of calls to BLAS kernels, eliding the arguments that are not matrix dimensions, is

```
// T <- A * B
cblas_dgemm(,,,d1,d3,d2,,,d1,,d2,,,d1);
// T * C
cblas_dgemm(,,,d1,d4,d3,,,d1,,d3,,,d1);
```


DOI: 10.1002/spe

$$\underbrace{(\#k_0, \ldots, \#k_i, \ldots, \#k_{n-1}, avg, min, max)}_{n+3 \text{ dimensions}}$$

(a)

$$\underbrace{(\ldots, \overbrace{\#k_i, avg_i, min_i, max_i, time_i}^{5 \text{ components for kernel } i}, \ldots, \overbrace{\#k_{n-1}, avg_{n-1}, min_{n-1}, max_{n-1}, time_{n-1}}^{5 \text{ components for kernel } n-1})}_{5n \text{ dimensions}}$$

(b)

Figure 1. A feature vector defined in the simple feature space (a) and the complex feature space (b) when n distinct kernels are used. $\#k_i$ is the number of occurrences of kernel i in the solution, and avg, min, and max represent the average, minimum, and maximum matrix dimensions, respectively. In (b), avg_i, min_i, and max_i, are the average, minimum, and maximum matrix dimensions for the calls to kernel i, respectively, and $time_i$ is the computational complexity measure of kernel i.

We compute the average matrix dimension as $(6 \times d1 + 2 \times d2 + 3 \times d3 + d4)/12$. The text of the $A \times (B \times C)$ solution is

```
// T <- B*C
cblas_dgemm(,,,d2,d4,d3,,,d2,,d3,,,d2);
// A * T
cblas_dgemm(,,,d1,d4,d2,,,d1,,d2,,,d1);
```

The average in this case is $(3 \times d1 + 5 \times d2 + 2 \times d3 + 2 \times d4)/12$. Let us consider the example $d1 = d3 = d4 = 100$ and $d2 = 10$. Because $d2$ is small, it is more expensive to compute $(A \times B) \times C$ than $A \times (B \times C)$. And indeed, the computed average matrix size for the first solution is 85 but only 62.5 for the second solution. Our definition of average matrix dimension is motivated by the need to differentiate between two solutions for what is perhaps the most commonplace linear algebra operation, that is, matrix multiplications. Nevertheless, results in upcoming sections show that this definition leads to good results even when the kernels are not matrix multiplications.

Based on the aforementioned notion of average matrix dimension, we propose two feature space definitions. The first definition is inspired by the feature space definition in [34]. Given a solution that can potentially use n distinct kernels, we define a feature vector with $n + 3$ components. Component $i = 0, \ldots, n-1$ is the number of occurrences of kernel i in the solution. Components n, $n + 1$, and $n + 2$ are the average, minimum, and maximum matrix dimensions, respectively. We term this feature space definition 'simple' and depict it in Figure 1(a).

Our second feature space definition consists of feature vectors with $5n$ components. For each of the n possible kernels, there are five components in the vector: the count of the kernel's occurrences; the average, minimum, and maximum matrix dimensions *for calls to that kernel*; and a (coarse) measure of the computational complexity of the kernel. For each kernel, we determine a computational complexity measure as a single number computed from the asymptotic computational complexity of the algorithm implemented by the kernel and the dimensions of the matrices. For instance, if the algorithm is $O(n \times m \times m)$, $n = 200$ and $m = 10$, then we compute the complexity measure as $200 \times 10 \times 10 = 20,000$. We term this feature space definition 'complex' and depict it in Figure 1(b).

5. EVALUATION METHODOLOGY

5.1. Considered linear algebra kernels

We consider seven double precision floating point arithmetic kernels from the BLAS and LAPACK libraries, as shown in Table I. While many other kernels are available, these seven kernels are commonly used, represent various types of computation, and exhibit different performance scaling

DOI: 10.1002/spe

Table I. The seven BLAS/LAPACK kernels used in this work.

Name	Description	Complexity
`daxpy`	Add two $n \times m$ matrices	$O(n \times m)$
`dgemm`	Multiply a $n \times m$ matrix by a $m \times p$ matrix.	$O(n \times m \times p)$
`dsymm`	Multiply a symmetric $n \times n$ matrix by a $n \times m$ matrix.	$O(n^2 \times m)$
`dtrmm`	Multiply a triangular $n \times n$ matrix by a $n \times m$ matrix	$O(n^2 \times m)$
`dgetrf`	Compute the LU factorization of a $n \times m$ matrix	$O(n^2 \times m)$
`dgetri`	Compute the inverse of an LU-factorized $n \times n$ matrix	$O(n^2)$
`dtrsm`	Solve a triangular system with a $n \times n$ left-hand side and a $n \times m$ right-hand side	$O(n^2 \times m)$

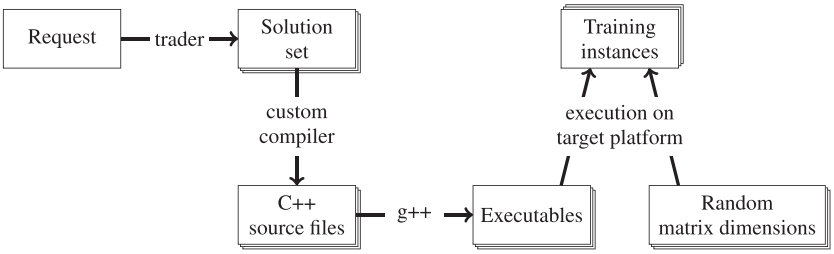

Figure 2. Generating training dataset instances that correspond to a given request.

behaviors. For all the experiments in this work, our target computer is a dual quad-core 2.6 GHz Intel Xeon with 24 GB of RAM running Linux with a 3.2.0 kernel. We use the OpenBlas v0.2.9 library and the LAPACK v3.5.0 library. Note that our approach is agnostic to the choice of the particular libraries used (e.g., we have obtained similar results using the ATLAS library [20] instead of OpenBlas).

5.2. Default training dataset generation

In this section, we describe how we generate a 'default' training dataset that we use to obtain initial results in Section 6. (We experiment with other training datasets in Section 6.2.) A training dataset is a set of feature vectors (in our context, kernel compositions) along with corresponding target values (in our context, response times). Each instance in our training dataset corresponds to a kernel composition with particular matrix dimensions, as described in Section 4, augmented with the response time measured on our target computer. For a given kernel composition, we measure response time for 50 random instances of the matrix dimensions, which are sampled uniformly between 1 and 500, thus generating 50 training instances.

Our default training dataset includes instances that correspond to the invocation of a single kernel, for a total of $7 \times 50 = 350$ instances (seven individual kernels, each executed for 50 random matrix dimension instances). It seems reasonable that each individual kernel should be part of any training dataset to allow correct prediction of the response times of those requests that are so simple that they can be answered by a single kernel invocation.

In addition to these 350 training instances, we also generate instances based on actual requests. Given an arbitrary request, which can be answered using the kernels in Table I, we generate a set of training instances using the methodology depicted in Figure 2. The trader generates several solutions. We bound the number of solutions to 20. Recall that the trader returns solutions in increasing order of complexity. In our experiments, solutions beyond the 20th, if any, are very complex (with large numbers of kernels and kernel invocations) and do not lead to response time improvements when compared with simpler solutions (and often lead to slowdowns).

Each solution is a specification of a particular composition of the kernels (an XML file in our implementation). We developed a simple custom compiler to transform this specification into a C++ program that invokes the kernels with the correct parameters and in the correct order, taking as command-line arguments the dimensions of all the matrices involved in the computation. This

DOI: 10.1002/spe

Table II. Requests used for generating the default training dataset.

	$+$	$*$	$^{-1}$	^{t}	lu
$+$	$(A+B)+C$	$(A+B)*C$	n/a*	$(A+B)^t$	n/a*
$*$	$(A*B)+C$	$(A*B)*C$	$(A*B)^{-1}$	$(A*B)^t$	$lu(A*B)$
$^{-1}$	$A^{-1}+B$	$A^{-1}*B$	n/a†	A^{-1^t}	n/a‡
^{t}	A^t+B	A^t*B	$A^{t^{-1}}$	n/a†	$lu(A^t)$
lu	$lu(A)+B$	$lu(A)*B$	n/a§	$lu(A)^t$	n/a§

†: idempotent operation; ‡: redundant because $lu(lu(A)) = lu(A)$; §: redundant because a matrix inversion is computed using an LU factorization; *: cannot be processed by the trader in [6].

program is then compiled using a C++ compiler and executed 50 times on a target computer, each time for a different random sampling of the matrix dimensions. The response time of each execution is measured, thus generating a new training instance that is added to our training dataset.

We generate a set of requests by combining two of the addition ($+$), multiplication ($\times$), inversion ($^{-1}$), transposition (^{t}), and LU factorization (lu) operations. These five operations are commonly used in linear algebra computations. Combining only two operations in the training results is sufficient to obtain a large training dataset and, as seen in later section, leads to good results. All our training requests are depicted in Table II. As explained in the caption of the table, some combinations are not considered because they correspond to idempotent or redundant operations. In addition, $lu(A+B)$ and $(A+B)^{-1}$ requests cannot be processed by the trader component we use in this work [6]. This is because the type system of its equational unification procedure requires that the $A+B$ matrix be known to be invertible, which cannot be determined based on whether A and/or B are invertible.

In total, 18 requests are shown in Table II. For each such request, we generate a version for 'general', 'symmetric', and 'triangular' matrices, for a total of $18 \times 3 = 54$ requests. Indeed, the trader accepts request specifications that indicate matrix properties, which then leads it to use different kernels among these listed in Table I. For these 54 requests, the trader produces a total of 448 solutions. Because each solution is then executed 50 times, we obtain $448 \times 50 = 22,400$ training instances. Together with the 350 training instances for single kernel invocations, our default training dataset contains a total of 22,750 instances.

5.3. Testing dataset

The requests we use to generate our default training dataset were chosen systematically (Table II). Instead, we wish to evaluate our approach with a testing dataset (i.e., a set of feature vectors whose target values must be predicted) that is derived from requests that may occur in representative applications. To this end, we have surveyed various uses of computational linear algebra in well-known domains of application and have selected seven typical requests. These requests are listed hereafter, with in parenthesis are the names by which we refer to them in all that follows:

- Image cross-dissolve operation: $\alpha A + \beta B$ (*dissolve*).
- Haar wavelets image compression/decompression:
  - Transformation matrix computation: $W = W_1 \times W_2 \times W_3$ (*transform*).
  - Compression : $T = W^t \times A \times W$ (*compress*).
  - Decompression : $A = (W^{-1})^t \times T \times W^{-1}$ (*uncompress*).
- Data encryption with LU factorization: $lu(D_1^{-1} \times A \times D_2)$, where D_1 et D_2 are diagonal matrices (*encrypt*).
- Schur complement computation:
  - General case: $A - B \times D^{-1} \times C$ (*schur*).
  - Symmetric initial matrix: $A - B \times D^{-1} \times B^t$ (*schursym*).

DOI: 10.1002/spe

We instantiate each aforementioned request assuming double precision arithmetic and with 50 random instantiations of the matrix dimensions, sampling matrix dimensions uniformly between 1 and 10,000. Our testing dataset thus contains $7 \times 50 = 350$ requests. Note that although our training datasets contain vectors with matrix dimensions in the 1–500 range, we use the larger 1–10,000 range for our testing dataset. This is to evaluate the generalizing power of the learning algorithms, that is, whether they are able to produce good predictions for requests with matrix dimensions that do not occur in the training dataset.

We must determine whether our testing dataset is appropriate for evaluating a solution (i.e., kernel composition) selection strategy. For instance, it could be that the shortest response time for each request is always obtained when using the first solution returned by the trader (recall that the trader returns the simplest solution first). Or it could be that all solutions returned by the trader for a given request lead to similar response times. In such cases, the trivial 'pick the first solution returned by the trader' strategy would be best. Instead, we wish to have some cases in which a solution selection strategy would have to rely on a response time prediction to select the appropriate solution. For each request in our testing dataset, using the methodology shown in Figure 2, we have measured the response time of the first 20 solutions produced by the trader component. For 193 of the requests (or 55.1%), the first solution is not the solution that leads to the shortest response time. For 116 of the requests (or 33.1%), there is at least one solution that leads to at least 10% response time reduction when compared with the first returned solution. For 50 of the requests (or 14.3%), this reduction is above 90%. We conclude that our testing dataset contains a sufficient number of cases in which solution selection is nontrivial and for which we hope a machine learning approach can succeed. In upcoming sections, we use these exhaustive response time measurements for our testing dataset as ground truth to precisely quantify the effectiveness of machine learning algorithms.

5.4. Machine learning algorithms

Our objective is not to make a novel contribution to the field of machine learning but rather to determine whether known machine learning techniques can be used for our purpose. Consequently, we use machine learning algorithms implemented in the Weka package [36, 37] with default configuration parameter values. We initially selected those algorithms in Weka that can be applied to regression problems, for a total of 22 algorithms. We then perform a 'sanity check' evaluation of these algorithms. To this end, we generate a training dataset using the methodology shown in Figure 2 but using the testing requests in Section 5.3 as starting points. In other words, we train the machine learning algorithms with the very requests that are used for testing. The only differences between the training and the testing datasets are the matrix dimensions, which are chosen randomly in the range 1–500 for the former and in the range 1–10,000 for the latter. We perform this evaluation using both feature space definitions described in Section 4. Our goal is to eliminate those algorithms that have little hope of producing useful results in the general case when the training dataset is not as similar to the testing dataset.

We find that 7 of the 22 selected algorithms lead to poor results regardless of the feature space definition used. These algorithms predict the same response time for all the solutions for a request and thus end up always returning the first solution. This behavior may be due to inherent weaknesses of these algorithms (e.g., the ZeroR algorithm in Weka is a baseline algorithm that relies only on the target value and ignores all input features, and the SimpleLinearRegression and DecisionStump algorithms only look at a single input feature for prediction). It is also possible that these algorithms lead to poor results because we use them with their default configuration parameter values or because their implementations in Weka are incorrect. Regardless, we are left with 15 algorithms. We list these algorithms in the succeeding text in parentheses, categorized in five different classes according to the Weka documentation:

- *Functions*: These algorithms classify the data by building a function, including neural network methods and regression methods (LeastMedSq, LinearRegression, MultilayerPerceptron, PaceRegression, RBFNetwork, and SMOreg).
- *Lazy*: These algorithms do not build a global model based on the training dataset but instead use the training dataset to build a local model for each received testing instance (IBk and LWL).

DOI: 10.1002/spe

- *Meta*: These algorithms are composite and use other algorithms as building blocks (AdditiveRegression, Bagging, and RegressionByDiscretization).
- *Rules*: These algorithms attempt to classify the data by building a set of if-then rules (ConjunctiveRule and M5Rules).
- *Trees*: These algorithms classify the data by building a tree-structured model in which each node represents a feature and each branch represents a range of node values (M5' and REPTree).

We refer the reader to the Weka documentation [38] for detailed descriptions and bibliographical references for these algorithms.

6. EXPERIMENTAL EVALUATION

6.1. Algorithm selection using the default training dataset

We have found that some of our 15 candidate machine learning algorithms are sensitive to the matrix dimensions in the training dataset. To observe this sensitivity, we have generated 20 default training datasets using the methodology described in Section 5.2, each generated using a different seed for the random number generator. Furthermore, we generate two sets of 20 training datasets, one for each of the two feature space definitions in Section 4.

Figure 3 shows the fraction, in percentage, of the testing requests for which each algorithm picks the solution with the shortest response time. A perfect algorithm would lead a value of 100%, and higher data points in the figure correspond to better performance. The strategy that would consists in always picking the first solution returned by the trader is shown as a horizontal line at 44.9%, and we see that several machine learning algorithms outperform this strategy. Some of these algorithms are sensitive to the random matrix dimensions in the training dataset, as seen in a wide spreads of the data points. For instance, MultilayerPerceptron shows results ranging from 20.0% to 50.6% when using the simple feature space definition and from 19.7% to 60.0% when using the complex feature space definition. Other algorithms, such as LinearRegression, PaceRegression, or SMOreg, show little sensitivity to the training dataset. When drawing pairwise comparisons between the two feature space definitions across all 20 training datasets, we find that 11 of the 15 algorithms achieve better results with the complex feature space for more than 18 of the 20 training datasets. The remaining four algorithms, IBk, LWL, AdditiveRegression, and RegressionByDisc, lead to better results using the simple feature space for 20, 18, 15, and 5 of the 20 datasets, respectively.

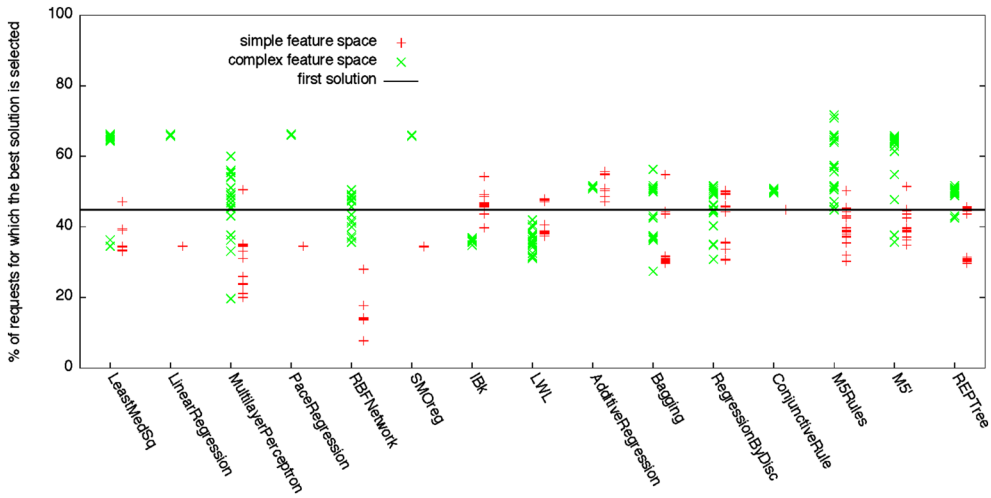

Figure 3. Percentage of testing requests for which an algorithm picks the solution with the shortest response time, for each of our 15 candidate algorithms and for both the simple and the complex feature space definition. For each algorithm and each feature space definition, 20 data points are shown, each for a different default training dataset instantiation. Many points overlap.

DOI: 10.1002/spe

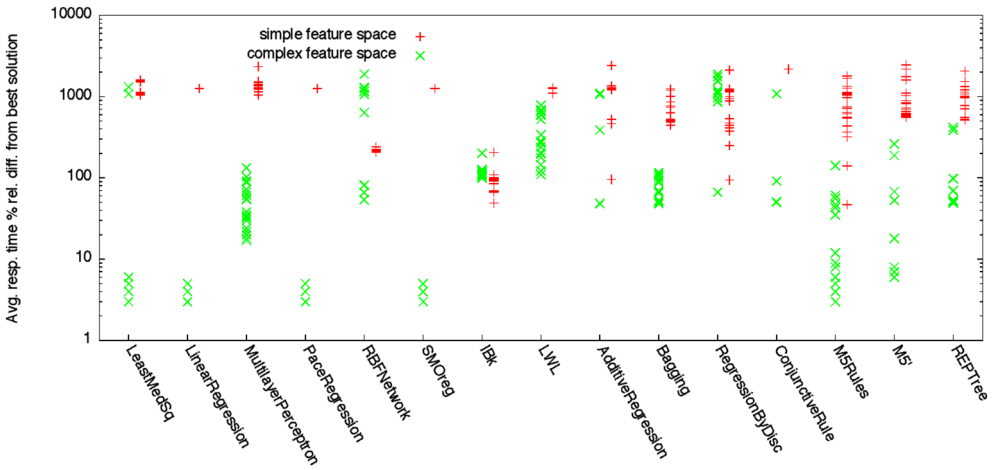

Figure 4. Average percent relative difference between the response time of the selected solution of that of the best solution, excluding cases in which the selected solution is the best solution, for each of our 15 candidate algorithms for both the simple and the complex feature space definition. For each algorithm and each feature space definition, 20 data points are shown, each for a different default training set instantiation. Many data points overlap. The vertical axis is on a logarithmic scale.

Based on the results in Figure 3, three algorithms lead to high and consistent performance (between 65.7% and 66.3% across all 20 training datasets) when the complex feature space definition is used: LinearRegression, PaceRegression, and SMOreg. All three algorithms are in the 'function' algorithm category in the Weka documentation.

Picking the best solution, that is, with the shortest response time, for many requests in the testing dataset is clearly desirable. But when the solution picked by an algorithm is not the best one, it is desirable for that solution to have low, even if not the lowest, response time. Figure 4 shows, for each algorithm, the average percentage relative difference between the response time of the solution selected by the algorithm and the lowest response time across all enumerated solutions. In this figure, lower data points correspond to better performance. These results exclude cases in which the algorithm selects the solution with the lowest response time so that the figure shows by how much each algorithm 'loses' on average when it does not select the best solution.

As in Figure 3, we see that the use of the complex rather than the simple feature space allows most algorithms to obtain better results. The key observation from these results is that the three algorithms that select the best solution most often (LinearRegression, PaceRegression, and SMOreg, all using the complex feature space) also lead to the best results in Figure 4 as well. On average, and across all 20 default training dataset instantiations, the solutions selected by these algorithms have response times at most 5.0% larger than the best response times. Furthermore, these three algorithms are the least sensitive to the instantiation of the training dataset.

We conclude that, for our purpose, the complex feature space definition should be used in conjunction with the LinearRegression, PaceRegression, or SMOreg algorithms as implemented in Weka. These algorithms lead to performance that is insensitive to the instantiation of the training dataset, often select the best solutions (i.e., for more than 65.7% of the requests in our testing dataset) and otherwise select solutions that are close to the best solutions (i.e., with response time at most 5.0% larger). In all that follows, we use the LinearRegression algorithm with the complex feature space definition.

6.2. *Impact of the complexity of the solutions in the training dataset*

All results in the previous section are obtained using the default training dataset defined in Section 5.2. We have seen that the LinearRegression algorithm has little sensitivity to the instantiation of the training dataset, that is, to the random matrix dimensions used to construct training instances. Our training dataset contains 22,750 instances. In this section, we consider subsets of

DOI: 10.1002/spe

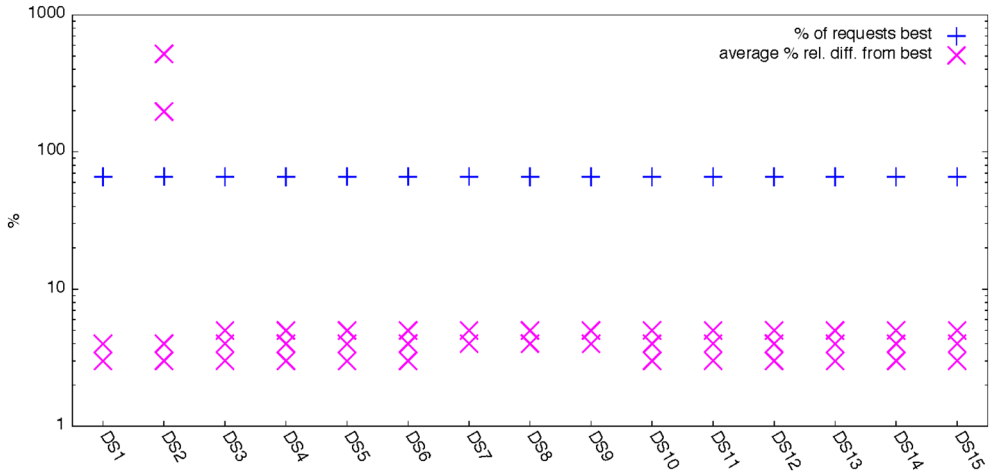

Figure 5. Percentage of testing requests for which LinearRegression selects the best solution and relative percentage difference in response time from the best solution when LinearRegression does not select the best solution, versus subset of the default training dataset. Twenty data points are shown for each metric and each training dataset subset, each for a different default training set instantiation. Many points overlap. The vertical axis is on a logarithmic scale.

this dataset to determine empirically how much training data are needed for our approach to remain effective.

Using DS to denote the default training dataset, we use DS_i to denote the subset of DS that contains training instances that correspond to solutions that use up to i kernel invocations ($DS_i \subset DS_{i+1}$). DS_1 contains 1800 instances, DS_2 contains 15,010 instances, DS_3 contains 21,100 instances or 92.75% of the training instances in DS, and $DS = DS_{15}$, meaning that the solutions to the requests in Table II use up to 15 kernel invocations. Note that solutions with this many kernels are likely not sensible and include redundant and idempotent operations as explained in Section 2. We train the LinearRegression algorithm with DS_i for $i = 1, \ldots, 15$ and evaluate its performance.

Figure 5 shows the two metrics shown in Figures 3 and 4: the percentage of the testing requests for which the algorithm selects the solution with the shortest response time, as well as the average relative percentage difference in response time from the best solution when it does not select the best solution. Results are shown for 20 instantiations of the DS_i training dataset, for $i = 1, \ldots, 15$. The first metric is not sensitive to the training dataset used, with all results within a narrow 65.4–66.3% range. The second metric is also mostly consistent across training datasets with a narrow 3.0–5.0% range, with the exception of the DS_2 training dataset for which some instantiations lead to significantly higher values (i.e., solutions with response time up to 519.0% larger than that of the best solution). We have been unable to explain these outliers.

We conclude that it is sufficient to include feature vectors that correspond to a single kernel invocations in our training dataset, even though the chosen solutions are multi-kernel. In other words, kernel composition behaviors can be captured well using regression from sets of individual kernel response time measurements. Overall, but for the exception of DS_2, the learning algorithm exhibits little sensitivity to the particular instantiation of the training dataset.

7. POTENTIAL IMPACT IN PRACTICE

Results in the previous section show that response time prediction through proper machine learning algorithms can lead to significant improvements over the 'pick the simplest' strategy for solving the solution selection problem and that it is possible to train a prediction model with simple datasets effectively. An attractive proposition is to implement our approach as part of interactive linear algebra computing environments, such as Matlab [39], Scilab [40], or GNU Octave [41]. Simple

DOI: 10.1002/spe

experiments with these tools reveal that, for instance, none of them can handle the pathological matrix multiplication case mentioned in Section 4. In what follows, we study the potential impact of our approach if it were implemented as part of Octave.

7.1. Methodology

All our experiments are performed on a single host, as described in Section 5.2, using GNU Octave v3.0.5. We run the LinearRegression algorithm with one random instantiation of the DS_3 training dataset, as defined in Section 6.2. For each request in our testing dataset, described in Section 5.3, we execute the request using Octave and using our approach. Both Octave and our approach use the same versions of the BLAS and LAPACK libraries.

When using Octave we simply input the text of the request at the Octave prompt (e.g., for the 'data encryption with LU factorization' request, we input `lu(inv(d1)*a*d2))`, after creating random input matrices with the particular dimensions specified in the request as it appears in our testing dataset.

For our approach, we use the trader to enumerate up to 10 solutions that can answer the request, run the LinearRegression algorithm to select the solution, and then execute that solution for random input matrices as well. In previous sections, we have used the term 'response' time to refer to the time to perform to computation. In this section, as we are considering a practical application of our approach, we consider the overall *time to solution*, which consists of three components: (i) solution enumeration time; (ii) solution selection time; and (iii) response time. It turns out that the solution selection time is negligible, under a milliseconds, while the time to solution is of the order of several seconds. This is expected because the LinearRegression algorithm builds a linear model of the components of the feature vectors, which can be evaluated in $O(x)$ time where x is the number of vector components, which is itself linear in the number of kernels ($x = 5n + 1$). By contrast, the first component can be large. For the seven requests in our testing dataset, the trader returns a set of solutions in 206.37s, 165.71s, 3.97s, 0.51s, 205.91s, 167.21s, and 209.82s, respectively. For this testing dataset, we observed that reducing the bound on the number of solutions to be returned has little impact on the enumeration time. Furthermore, even though we have limited the number of solutions to 10, we have determined experimentally that, out of our 350 testing requests, in only one instance would a solution beyond the 10th lead to a (hardly significant) reduction in response time. Because the enumeration time does not depend on matrix dimensions, it would become negligible provided that matrix dimensions, and thus response times, are sufficiently large. For small matrix dimensions, instead, the time to solution is dominated by the enumeration time. Matrix dimensions in our testing dataset are randomly sampled from a uniform distribution in the 1–10,000 range. As a result, some of the instantiated requests can have low response times (e.g., below 0.001s). For such requests with small matrix dimensions, using any approach to choose between multiple solutions cannot lead to measurable benefits because of enumeration overhead (and in particular our approach, which has overhead at least as high as 0.5s). Consequently, we generate a new testing dataset in which matrix dimensions are sampled in the 1000–10,000 range, thus ensuring that response times are at least a few seconds.

The reason for the high enumeration time is that the trader component is implemented in Java and that the code has not been the target of aggressive performance optimization (let alone a rewrite in another language). Re-implementing the trader is outside the scope of this work, but with the current implementation, our approach never outperforms Octave because the gain due to using better solutions for a request is offset by the solution enumeration time. So, in all that follows, we present results in which we divide the enumeration time by a factor 20, which could conceivably be achieved in practice if the trader were to be optimized/re-implemented. We also present results that completely ignore the enumeration time so as to quantify an upper bound on the performance that could be achieved with our approach.

7.2. Comparison

Figure 6 plots the percentage relative difference in time to solution between our approach and Octave, for each testing request. Data points above zero indicate cases in which our approach is

DOI: 10.1002/spe

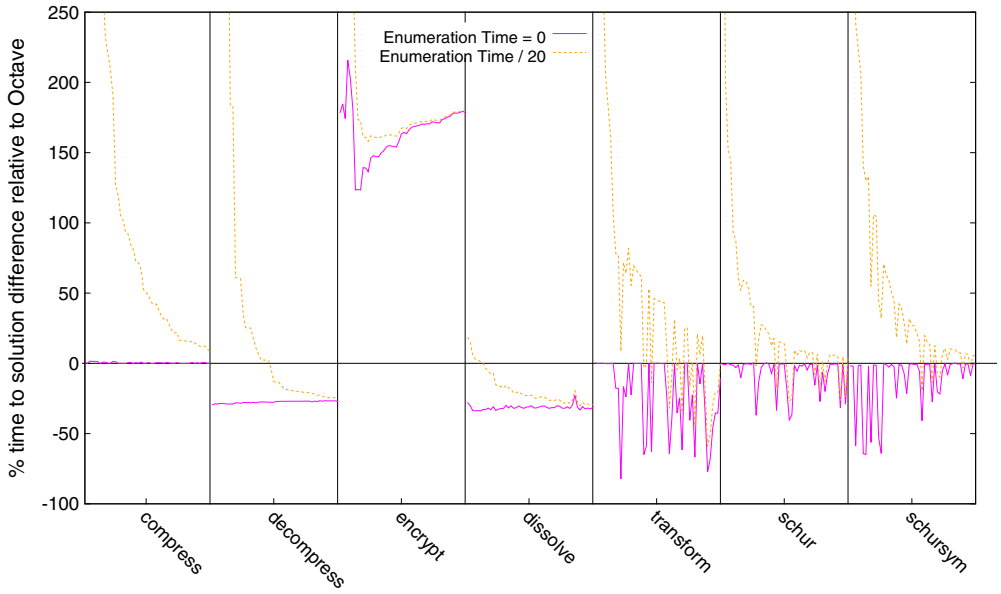

Figure 6. Percentage difference in time to solution relative to Octave for the seven testing requests. For each request, 50 data points are shown, sorted from left to right by increasing Octave time to solution. Data points above 250% are not shown to make the graph more readable.

outperformed by Octave. As explained in the previous section, two sets of results are shown, one assuming that the enumeration time is zero, and the other assuming that the enumeration time is reduced by a factor 20. For each testing request, 50 data points are shown, sorted from left to right by increasing Octave time to solution.

As one would expect, when the enumeration time is zero, our approach is equivalent to or better than Octave. The exception is the *encrypt* request, for which even with zero enumeration time our approach is more than twice as slow as Octave. It turns out that, for answering this request, Octave uses a custom method for performing fast matrix multiplication when one of the matrices is diagonal. Our method does not have a kernel to perform this fast operation at its disposal, hence its poor performance. For the 'compress' request, our approach assuming zero enumeration time is roughly equivalent to Octave with a relative percentage difference between −0.19% and 1.65%. For this request, Octave always uses the simplest solution, and the simplest solution is always the best. The small differences between our approach and Octave are due to timing imprecision and slight variations between executions of the same computation. For some of the requests, results show steady improvements across the board (i.e., between −29.53% and −26.58% for *decompress* and between −33.97% and −22.85% for *dissolve*). For the last three testing requests in the figure, the advantage of our approach versus Octave is less consistent but can achieve higher improvements, up to −82.31%, −40.29%, and −64.93% for *transform*, *schur*, and *schursym*, respectively.

When the enumeration time is nonzero (set to the actual enumeration time divided by a factor 20), many data points show our approach to be less effective than Octave. For readability reasons, Figure 6 cuts the vertical axis at 250%, but some data points are above 250%. Typically, for requests that are not compute intensive, the enumeration time is several factors longer than the response time. But as requests become more compute intensive, the enumeration time corresponds to a smaller share of the overall time to solution. This is why the general trend of the curves is decreasing. For the first testing request, *compress*, it is not surprising that our approach never outperforms Octave. As explained in the previous paragraph, the simplest solution is always the best for that request. Consequently, our approach is pure overhead. For the *decompress* testing request, our approach outperforms Octave when Octave's time to solution is larger than 35s (or 26 of the 50 data points). For the most compute intensive, such request Octave's time to solution is at 382s, while that of our approach is at 278s. A similar observation can be made for the *dissolve* request. For this request, our approach outperforms Octave for 43 of the 50 data points. This request leads to much less

DOI: 10.1002/spe

computation than the *decompress* request (with Octave's response time only as high as 1.1s), and yet our approach is able to bring some benefits. For the last three requests in the figure, there are still cases in which we outperform Octave: in 30%, 22%, and 12% of the cases for *transform*, *schur*, and *schursym*, respectively.

7.3. Discussion

Our results show that our approach of selecting a solution based on response time prediction can bring substantial benefits over Octave, at least when matrix dimensions are large enough that response time dominates enumeration time. Two directions can be pursued to improve the time to solution of our approach: (i) improve the quality of the selected solutions and (ii) reduce the solution enumeration time.

A reason for pursuing the first direction is that our results are obtained with standard machine learning algorithms with default configuration parameters, as implemented in Weka. It may be the case that even better results could be achieved with better tuned or different algorithms. Also, we do not preprocess our training datasets even though some preprocessing, that is, removing duplicate training instances, can potentially lead to improved results [42]. However, it is likely that only marginal improvements could be achieved. For each testing request instantiation used for the experiments in the previous section, we have exhaustively computed the response time of all the solutions enumerated by the trader. We find that across all $7 \times 50 = 350$ requests, our approach selects a solution that has a response time within 1% of that of the best solution in more than 96% of the cases and within 2% in more than 99.5% of the cases. In other words, at least for our testing dataset, our approach is close to being optimal within the set of candidate solutions evaluated.

Because the solution enumeration time is a large contributor to the time to solution, pursuing the second direction would have a large impact and is in fact necessary if our approach is to be used in production. This would require a complete rewrite of the trader component in [6], including translation to another language, revisiting the design of the algorithms and data structures, and likely parallelization so that the algorithm can utilize multiple cores. An orthogonal approach would be to modify the trader so that it caches enumerated solutions for past requests, for example, in a locally maintained database. The enumeration only depends on the algebraic expression of the request and not on matrix dimensions. Once this database is large and if the workload is relatively consistent, a newly entered request would have a high chance of having already been processed in the past.

8. CONCLUSION

In this paper, we have tested various machine learning algorithms and demonstrated the feasibility of using simple linear regression for selecting a fast composition of linear algebra kernels among multiple candidate compositions enumerated with the equational unification approach introduced in [6]. We have proposed two feature space definitions and have constructed training datasets to train predictors using 22 standard machine learning algorithms as implemented in Weka. We have found that our more complex feature space definition leads to better results than our simpler definition. We have seen that some algorithms exhibit high predictive power and little sensitivity to the particular instantiation of the training dataset. Among these, we have selected the LinearRegression algorithm implemented in Weka, trained with our complex feature space definition with a dataset that contains 22,750 feature vectors. We have evaluated the potential of our approach if it were implemented as part of an interactive computing environment such as Octave. Our approach consistently leads to substantial benefit over Octave when the response time is dominant (or when enumeration time is negligible) and always selects good solutions for the requests in our testing dataset. Unfortunately, it suffers from high solution enumeration overhead. To enumerate solutions, we use the trader component implemented in [6] as is. It turns out that, because of a Java implementation that has not been optimized aggressively, enumeration times can reach hundreds of seconds. As a result, our approach would only outperform Octave for very large request response times (i.e., with very large matrices). We have shown results assuming that the trader is 20 times faster than with its current implementation and have seen that significant improvements are then possible in many cases.

DOI: 10.1002/spe

A near-term future development direction is to revisit the work in [6] and to re-implement the trader component so as to reduce enumeration time by at least one order of magnitude, both via code optimization and enumeration caching. The next step is then to implement our approach as part of Octave or another interactive linear algebra computing tool. This implementation will have to support a large number of or all the kernels in the BLAS and LAPACK libraries. A broader direction is to extend this work to multi-threaded and parallel versions of these libraries [3, 4]. The advantage of our approach is that because it does not rely on explicit performance models, in principle, it can be applied to these more complex settings and still lead to good results.

REFERENCES

1. Blackford S, Demmel J, Dongarra J, Duff I, Hammarling S, Henry G, Heroux M, Kaufman L, Lumsdaine A, Petitet A. *et al.* An updated set of Basic Linear Algebra Subprograms (BLAS). *ACM Transactions on Mathematical Software* 2002; **28**(2):135–151.
2. Anderson E, Bai Z, Bischof C, Blackford S, Demmel J, Dongarra J, Du Croz J, Hammarling S, Greenbaum A, McKenney A. *et al. Lapack Users' Guide (third ed.)* Society for Industrial and Applied Mathematics: Philadelphia, PA, USA, 1999.
3. Choi J, Demmel J, Dongarra J, Dhillon I, Ostrouchov S, Petitet A, Stanley K, Walker D, Whaley RCW. ScaLAPACK: a portable linear algebra library for distributed Memory Computers - Design Issues and Performance. *Computer Physics Communications* 1996; **97**(1-2):1–15.
4. Agullo E, Demmel J, Dongarra J, Hadri B, Kurzak J, Langou J, Ltaief H, Luszczek P, Tomov S. Numerical linear algebra on emerging architectures: the PLASMA and MAGMA projects. *Journal of Physics: Conference Series* 2009; **180**(1).
5. Cormen T, Leiserson C, Rivest R, Stein C. *Introduction to Algorithms* (3rd edn). The MIT Press, 2009.
6. Hurault A, Daydé M, Pantel M. Advanced service trading for scientific computing over the grid. *Journal of Supercomputing* 2009; **49**(1):64–83.
7. Hurault A, Yarkhan A. Intelligent Service Trading and Brokering for Distributed Network Services in GridSolve. In *High Performance Computing for Computational Scienc-VECPAR 2010*, vol. 6449, Palma JML, Daydé M, Marques O, Lopes JC (eds)., Lecture Notes in Computer Science. Springer: Berlin Heidelberg, 2011; 340–351.
8. Goguen J, Meseguer J. Order-sorted algebra i: equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science* 1992; **105**(2):217–273.
9. Gallier J, Snyder W. Complete sets of transformations for general E-unification. *Theoretical Computer Science* 1989; **67**(2–3):203–260.
10. *Basic Linear Algebra Subprograms (BLAS) documentation*, 2014. Available from: http://www.netlib.org/blas/.
11. Asperti A, Padovani L, Coen CS, Schena I. Helm and the semantic math-web. *Proc. of the 14th international conference on theorem proving in higher order logics*, TPHOLs '01, Springer-Verlag, London, UK, 2001; 59–74.
12. Deductive Composition of Astronomical Software from Subroutine Libraries. In *Automated Deduction-CADE-12*, Bundy A (ed.). Springer: Berlin, Heidelberg, 1994; 341-355.
13. Johnson TA, Eigenmann R. Context-sensitive domain-independent algorithm composition and selection. *SIGPLAN Notices* 2006June; **41**:181–192.
14. Iakymchuk R, Bientinesi P. Modeling performance through memory-stalls. *ACM SIGMETRICS Performance Evaluation Review* 2012; **40**(2):86–91.
15. Snavely A, Carrington L, Wolter N, Labarta J, Badia R, Purkayastha A. A framework for performance modeling and prediction. In *SC*, Giles RC, Reed DA, Kelley K (eds). ACM, 2002; 1–17.
16. Grigori L, Li XS. Towards an accurate performance modeling of parallel sparse factorization. *Applicable Algebra in Engineering, Communication and Computing* 2007; **18**(3):241–261.
17. Tikir MM, Carrington L, Strohmaier E, Snavely A. A Genetic Algorithms Approach to Modeling the Performance of Memory-bound Computations. *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, SC '07, vol. 47, ACM, Reno, Nevada, New York, NY, USA, 2007; 47:1–47:12.
18. Marin G, Mellor-Crummey J. Cross-architecture performance predictions for scientific applications using parameterized models. *SIGMETRICS Perform. Eval. Rev.* 2004; **32**(1):2–13.
19. Hoste K, Phansalkar A, Eeckhout L, Georges A, John LK, De Bosschere K. Performance Prediction based on Inherent Program Similarity. *Proceedings of the Fifteenth International Conference on Parallel Architectures and Compilation Techniques (PACT)*, ACM, Seattle, WA, USA, 2006; 114–122.
20. Whaley RC, Petitet A, Dongarra JJ. Automated empirical optimization of software and the ATLAS project. *Parallel Computing* 2001; **27**(1–2):3–35.
21. Vuduc R, Demmel JW, Yelick KA. OSKI: a library of automatically tuned sparse matrix kernels. *Proceedings of scidac 2005*, Journal of Physics: Conference Series, Institute of Physics Publishing, San Francisco, CA, USA, June 2005.
22. Bilmes J, Asanovic K, Chin CW, Demmel J. Optimizing matrix multiply using phipac: a portable, high-performance, ansi c coding methodology. *Proceedings of the 11th international conference on supercomputing*, ICS '97, ACM, New York, NY, USA, 1997; 340–347.

DOI: 10.1002/spe

23. Zee Field GV, Chan E, van de Geijn RA, Quintana-Orti ES, Quintana-Orti G. The libflame library for dense matrix computations. *Computing in Science and Engineering* 2009; **11**(6):56–63.
24. Li Y, Dongarra J, Tomov S. A note on auto-tuning gemm for gpus. *Proceedings of the 9th international conference on computational science: Part i*, ICCS '09, Springer-Verlag, Berlin, Heidelberg, 2009; 884–892.
25. Ansel J, Kamil S, Veeramachaneni K, Ragan-Kelley J, Bosboom J, O'Reilly UM, Amarasinghe S. Opentuner: an extensible framework for program autotuning. *International conference on parallel architectures and compilation techniques*, Edmonton, Canada, 2014 August.
26. Hartono A, Norris B, Sadayappan P. Annotation-based empirical performance tuning using orio. *Proceedings of the 2009 ieee international symposium on parallel&distributed processing*, IPDPS '09, IEEE Computer Society, Washington, DC, USA, 2009; 1–11.
27. Shin J, Hall MW, Chame J, Chen C, Hovland PD. Autotuning and specialization: speeding up matrix multiply for small matrices with compiler technology. *In the fourth international workshop on automatic performance tuning*, New York, USA, 2009.
28. Peise E. Hierarchical Performance Modeling for Ranking Dense Linear Algebra Algorithms. *CoRR* 2012. http://arxiv.org/abs/1207.5217.
29. Peise E, Bientinesi P. Performance Modeling for Dense Linear Algebra. *Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis (PMBS12)*, SCC '12, IEEE Computer Society, Washington, DC, USA, 2012; 406–416.
30. Ipek E, Supinski BRD, Schulz M, Mckee SA. An approach to performance prediction for parallel applications. *Proceedings of the 11th International Euro-Par Conference on Parallel Processing,* Euro-Par'05, Springer-Verlag, Berlin, Heidelberg, 2005; 196–205.
31. Li J, Ma X, Singh K, Schulz M, de Supinski BR, McKee SA. Machine learning based online performance prediction for runtime parallelization and task scheduling. *Ispass*, IEEE, 2009; 89–100.
32. Gupta C, Mehta A, Dayal U. PQR: predicting query execution times for autonomous workload management. *Proceedings of the 2008 International Conference on Autonomic Computing,* ICAC '08, IEEE Computer Society, Washington, DC, USA, 2008; 13–22.
33. Thomas N, Tanase G, Tkachyshyn O, Perdue J, Amato NM, Rauchwerger L. A framework for adaptive algorithm selection in STAPL. *Proc. of the tenth acm sigplan symposium on principles and practice of parallel programming*, PPoPP '05, ACM, New York, NY, USA, 2005; 277–288.
34. Ganapathi A, Kuno H, Dayal U, Wiener JL, Fox A, Jordan M, Patterson D. Predicting multiple metrics for queries: better decisions enabled by machine learning. *Proc. of the 2009 ieee international conference on data engineering*, IEEE Computer Society, Washington, DC, USA, 2009; 592–603.
35. Hasan R, Gandon F. A machine learning approach to SPARQL query performance prediction. *The 2014 IEEE/WIC/ACM International Conference on Web Intelligence*, Warsaw, Poland, 2014.
36. Witten IH, Frank E. *Data Mining: Practical Machine Learning Tools and Techniques* (Second edn.), Morgan Kaufmann Series in Data Management Sys. Morgan Kaufmann, 2005.
37. Hall M, Frank E, Holmes G, Pfahringer B, Reutemann P, Witten IH. The weka data mining software: an update. *SIGKDD Exploration Newsletter* 2009November; **11**:10–18.
38. Weka 3: data mining software in Java.
39. MATLAB. *version 7.10.0 (r2010a)*. The MathWorks Inc.: Natick, Massachusetts, 2010.
40. Scilab Enterprises. *Scilab: Le logiciel open source gratuit de calcul numérique*: Scilab Enterprises: Orsay, France, 2012.
41. Hansen JS. *GNU Octave: Beginner's Guide. Jesper Schmidt Hansen*, Learn by doing : less theory, more results. Packt, 2011.
42. Kolcz A, Chowdhury A, Alspector J. Data duplication: an imbalance problem?. *Proceedings of workshop on learning from imbalanced data sets (ii), icml*, 2003.

DOI: 10.1002/spe