Nitro: A Framework for Adaptive Code Variant Tuning

Saurav Muralidharan, Manu Shantharam, Mary Hall
University of Utah
Salt Lake City, UT
Email: {sauravm, manushan, mhall}@cs.utah.edu

Michael Garland, Bryan Catanzaro
NVIDIA Corporation
Santa Clara, CA
Email: {mgarland, bcatanzaro}@nvidia.com

***Abstract*—Autotuning systems intelligently navigate a search space of possible implementations of a computation to find the implementation(s) that best meets a specific optimization criteria, usually performance. This paper describes Nitro, a programmer-directed autotuning framework that facilitates tuning of code variants, or alternative implementations of the same computation. Nitro provides a library interface that permits programmers to express code variants along with meta-information that aids the system in selecting among the set of variants at run time. Machine learning is employed to build a model through training on this meta-information, so that when a new input is presented, Nitro can consult the model to select the appropriate variant. In experiments with five real-world irregular GPU benchmarks from sparse numerical methods, graph computations and sorting, Nitro-tuned variants achieve over 93% of the performance of variants selected through exhaustive search. Further, we describe optimizations and heuristics in Nitro that substantially reduce training time and other overheads.**

***Keywords*-Autotuning; Performance Optimization; GPUs;**

I. Introduction

Autotuning systems empirically evaluate a search space of possible implementations of a computation to identify the implementation that best meets a specific optimization criteria, usually performance. A subset of autotuning systems are programmer-directed, permitting programmers to express compactly the set of possible implementations and the system then performs the search in an automated fashion [1], [2], [3], [4].

A common mechanism in programmer-directed autotuning systems is the *code variant*, which represents a unique implementation of a computation, among many, that has the same interface and is functionally equivalent to the other variants but may employ fundamentally different algorithms or implementation strategies. Some programmer-directed autotuning systems permit expression of code variants, such as Sequoia [1] and PetaBricks [2]; earlier work by Brewer [5] on an auto-calibration toolkit built a model for variant selection related to input parameters. However, what is missing from these frameworks is more general meta-information that can be used to select variants, beyond input data set size. This presents a particular problem for *irregular applications*, such as sparse numerical methods and graph algorithms, and any other applications (e.g., sorting) where characteristics of the input data set may significantly impact selection of the best code variant, and is not known until run time.

One approach to selecting among code variants is to build a statistical *Model* that maps characteristics of the input data set to the appropriate variant. In previous work on the *algorithm selection problem* [6], statistical learning techniques are used to select among a set of different algorithms [7], [8], [9]. Frameworks that use statistical learning techniques include STAPL [10], which uses learning to dynamically select among algorithms, and OSKI [11], which uses learning to tune sparse linear algebra computations. To date, no general-purpose framework enables users to specify and tune arbitrary code variants, and also customize the tuning process.

This paper describes a new programmer-directed autotuning system called Nitro. This paper focuses on how code variants and meta-information for variant selection are expressed in Nitro, and the underlying system support that selects the most appropriate variant. Nitro targets two classes of users: expert programmers who specify the variants and their meta-information, and end users who invoke Nitro-enabled software *without using any Nitro-specific constructs*. Code variants are created and added to the system with library calls. In addition to expressing code variants, expert programmers specify how to calculate *features* or characteristics of the input data sets for each variant and representative training input data sets. The underlying Nitro system uses supervised learning in an off-line training phase to infer a model that maps from features of the input data set to variants. The model is then used by end users to select optimized code variants for new, unseen inputs. Nitro also includes a tuning interface to optionally customize the tuning process, which then invokes optimizations and heuristics to reduce training time of the model and amortize feature evaluation costs.

This system described in this paper makes the following contributions:

- **Portable library interface:** To make autotuning technology widely accessible for different platforms, Ni-

This research was funded in part by the U.S. Government. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

tro code variants are represented through library calls rather than language extensions.

- **Expressiveness:** Nitro permits programmer expression of features, their evaluation functions and training inputs, which allow programmers to construct general applications that use autotuning capability.
- **Demonstration:** We present the results of code variant selection in Nitro for five distinct computations targeting NVIDIA GPUs, an architecture where performance is particularly sensitive to details of parallel decomposition, data layout and control flow. The computations are from domains where input data properties significantly impact variant selection (sparse matrix, graph and sorting computations). Nitro achieves over 93% of the performance of variants selected through exhaustive search.

II. Nitro Library and Autotuner

Before describing the Nitro system, we first motivate our approach with an example, a sparse matrix-vector multiply (SpMV). In SpMV implementations, the driving principle is to avoid representing and computing zero-valued elements of the sparse matrix, thus saving both space and computation. A common sparse matrix representation is the Coordinate representation, where for each nonzero element in matrix A, the corresponding row and column are recorded and used in the computation in the following way:

```
for(i = 0; i < nnz; i++)
   y[row[i]] += A[i]*x[col[i]];
```

While general, the representation and associated computation can be improved if structural properties of the matrix, such as the distribution of row lengths, are known. In fact, most SpMV libraries incorporate a variety of matrix representations and associated code for this reason [11], [12], [13], [14], [15]. Unfortunately, the structure of the matrix is usually not known until run time, requiring the programmer to select the most appropriate variant directly, or some preprocessing of the input by the system to determine which version to use.

SpMV libraries usually incorporate multiple formats and sometimes multiple variants per format. For example, the CUSP library [15] for NVIDIA GPUs exposes the different variants and representations as part of the interface, and users select the appropriate variant to execute.

The way in which CUSP supports the end user in making these variant selections (and similar aspects of other libraries) inspired the approach taken in Nitro. Internally, CUSP examines properties of the input data set at run time to determine if a specific matrix representation selected by the user is likely to be efficient for that input. By encapsulating these properties along with a few others into features, a training phase can learn a model to guide the selection of the variant corresponding to the best matrix representation and among variants representing different parallelization strategies for a single representation. At run time, the variant selection can then be performed automatically.

This automatic support of variant selection in Nitro benefits the expert programmers designing software to be used by others in a variety of contexts. Such expert programmers often have an understanding of what variants are appropriate for a class of target architectures and some intuition about how the input data set properties affect variant selection. However, managing the details of collecting properties and determining cutoff values for variant selection requires extensive and costly trial-and-error experimentation. Therefore, it is realistic for expert programmers to provide a collection of variants and features, which are used as meta-information for variant selection. This support in Nitro not only increases the productivity of expert programmers by eliminating the manual encoding of variant selection, but also improves the useability of the software for its end users.

In the remainder of this section, we illustrate how the Nitro system can be used to automate variant selection for SpMV.

A. Nitro System Overview

Figure 1 provides a high-level overview of the Nitro system, which consists of two parts: the Nitro Library, implemented using C++ Templates (Figure 1a), and the Nitro Autotuner, written in Python (Figure 1b). The Nitro Library is invoked within an application/library to define a set of variants, $V = \{V_1, V_2, .., V_i\}$. The programmer also expresses meta-information for selecting variants: functions to compute features $F = \{F_1, F_2, .., F_j\}$ and optional constraints for each variant, shown as $\{C_1, C_2, .., C_k\}$ in the figure. Constraints are used to rule out certain variants that are either inappropriate or incorrect to use for a particular input.

The Nitro Autotuner is invoked with an external Python *tuning script* that allows programmers to specify the *Training Inputs*, how to perform feature evaluation, and other tuning properties for specific variants and the entire application/library. This decoupling between the library and the autotuner ensures that the main application code only contains algorithm-specific details such as variants and features, and allows programmers convenient experimentation with different tuning options and porting to different architectures. To communicate with the library, the Python-based autotuner generates a C++ header file and encapsulates the tuning properties within *tuning policies* for each variant.

The Nitro Autotuner builds a statistical *Model* (Figure 1b) that maps a set of features represented by a *feature vector* $[x_1, x_2, ..., x_n]$ to the label corresponding to the optimal variant for the corresponding input. By default, Nitro employs for this purpose *Support Vector Machines (SVMs)* [16], a widely-used machine learning algorithm to build the model from an off-line training phase on the *Training Input* so that

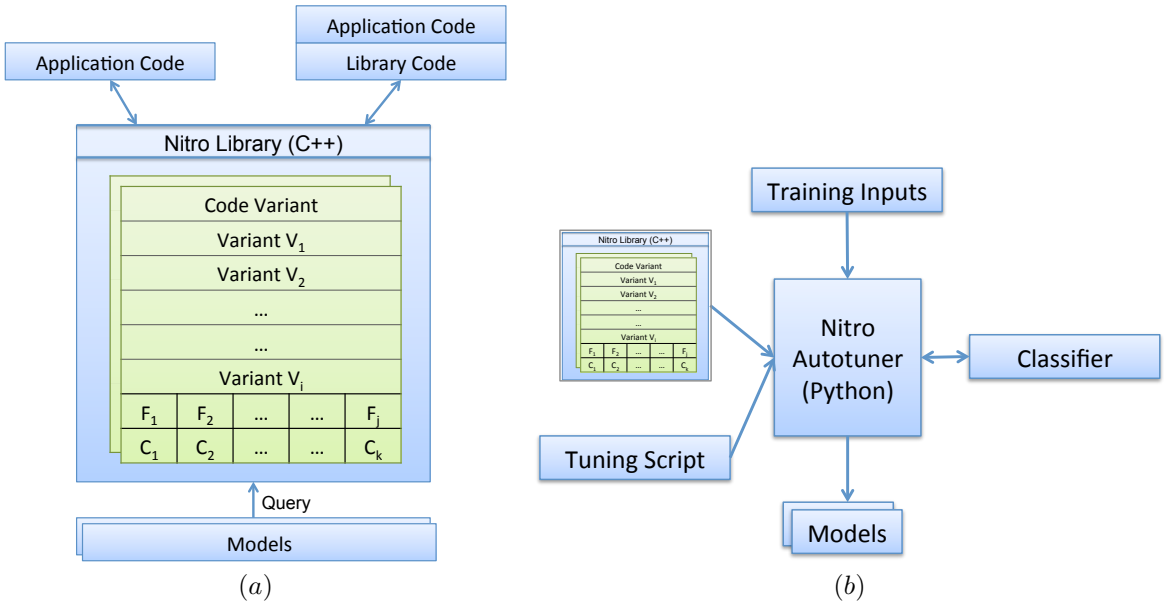

Figure 1. Overview of the Nitro system. (a) The production version of the library/application. The C++ library is used to define variants, features, and constraints. Calling the variant evaluates the input features at run-time and queries the accompanying model to select the right variant to execute for a given input. (b) The offline autotuning process. User provides a tuning script and training inputs. The autotuner runs the application/library for each training input and collects training data. The classifier is then consulted with the training data to construct the model(s).

Table I
LIST OF FUNCTIONS PROVIDED BY NITRO FOR VARIANT, FEATURE, AND CONSTRAINT MANAGEMENT

Function	Parameters	Description
`code_variant` Constructor	*Template parameters*: `tuning_policies` object, Tuple of argument types. *Arguments*: Pointer to `context` object	Creates `code_variant` object
`add_variant`	Pointer to Variant Function Object	Adds a variant to the code_variant object's internal variant table
`set_default`	Pointer to Variant Function Object	Used to set default variant to execute
`add_input_feature`	Pointer to Feature Function Object	Adds the specified function to the list of feature functions
`add_constraint`	Pointer to Variant Function Object, Pointer to Constraint Function Object	Adds a constraint function to execute before evaluating given variant.
`fix_inputs()`	Argument(s) to Variant	Fixes inputs to variant. Used for asynchronous feature evaluation.
`operator()(...)`	Argument(s) to Variant (empty with async_feature_eval)	Executes the correct underlying variant.

it can be consulted at run time given the feature vector of a new input. We use the publicly available `libSVM` [17] for this purpose.

B. Nitro Library Constructs

Table I provides a summary of the constructs available in Nitro for expressing variants and their associated features and constraints. Figure 2 provides Nitro code for SpMV, to illustrate these constructs, as described in the following paragraphs.

Defining and Adding Code Variants: Nitro represents a function that has code variants using the `code_variant` class. Each variant is expected to be functionally equivalent and must use the same interface. During instantiation, a tuple of the function's argument types, and its tuning policy must be specified as template arguments. The tuning policy for each such function is generated by the tuning script in a separate header file, as discussed in the next section. A function to be tuned by Nitro can thus be any general-purpose C++ function. Also during instantiation of the `code_variant` class, a pointer to a `context` object that maintains global state among all the variants in the program must be included as a constructor argument. In Figure 2, we define a function `SparseMatVec` within the `MySparse` library which provides a tuned SpMV implementation using Nitro. The details of the tuning process are thus abstracted away from the end user, who can use the `MySparse` library without ever needing to know about Nitro.

Each variant must be defined as a C++ function object deriving from the `variant_type` class. An example variant definition is provided in the bottom of Figure 2. Variants are added to the `code_variant` object using the `add_variant` function, which accepts a pointer to the

```
namespace MySparse {
void SparseMatVec(HostMatrix *matrix)
{
  using namespace nitro;
  typedef thrust::tuple<HostMatrix *> ArgTuple;

  // Create Nitro Tuning Context
  context cx;

  // Create code_variant object
  code_variant<tuning_policies::spmv,
               ArgTuple> spmv(cx);

  // Declare and Add Variants
  csr_vector_type<HostMatrix> __csr_vector;
  dia_type<HostMatrix> __dia;
  ...
  spmv.add_variant(&__csr_vector);
  spmv.add_variant(&__dia);
  ...

  // Set Default Variant
  spmv.set_default(&__csr_vector);

  // Declare and Add Features...
  nnz_type<HostMatrix> __nnz;
  num_rows_type<HostMatrix> __num_rows;
  ...
  spmv.add_input_feature(&__nnz);
  spmv.add_input_feature(&__num_rows);
  ...

  // ...and Constraints
  dia_cutoff_type __dia_cutoff;
  spmv.add_constraint(&__dia, &__dia_cutoff);
  ...

  // Variant Call
  spmv(matrix);
}

// Define CSR Vector Variant
template <typename HostMatrix>
struct csr_vector_type :
       nitro::variant_type<HostMatrix *> {
  double operator()(HostMatrix *matrix) {
    ...
  }
};
...
} // end namespace MySparse
```

Figure 2. Example Nitro Library interface for SpMV.

function object for that variant. All variants of a function must have the same argument type(s). Users may explicitly specify a default variant using the `set_default` function. Default variants are assumed to work correctly for all inputs and are used when one or more user-defined constraints fail. If no default is specified, the system selects the first variant as the default.

In Figure 2, we add two different variants for SpMV corresponding to different formats for the sparse matrix: `csr_vector_type` for *Compressed Sparse Row*, and `dia_type` for *Diagonal* [15].

The code for the variant must be specified in the `operator()` function, which is used by Nitro to invoke the desired variant. Nitro variants are required to return a double precision value, which by default denotes the time taken by the variant. However, by returning the appropriate value, Nitro can also be used to predict variants according to other optimization criteria, for example, energy usage, or to find the variant that provides the approximate result with the smallest margin of error.

Defining Input Features: Input features are described in Nitro through *feature functions*. These have the same argument types as the variant, but always return a `double`, which represents the value of the calculated feature for an input. In Nitro, feature functions must be wrapped in a function object derived from `input_feature_type`.

The `add_input_feature` function accepts a pointer to a feature function object and adds it to the internal feature function table. All values from the feature functions automatically get evaluated before the code for the variant starts executing. For example, in Figure 2, input features include `__nnz`, and `__num_rows`, the number of nonzeroes, and the number of rows, respectively. To hide the runtime overhead of feature evaluation, an optimization discussed in Section III-C is asynchronous feature evaluation; asynchronous feature evaluation is enabled by calling the `fix_inputs` function before calling `operator()`.

Defining Constraints: For certain inputs, it is possible that a variant produces wrong results, or takes unacceptably long to execute. Nitro provides support for handling such cases using user-defined *constraints*. Constraint functions can be added to code variants using the `add_constraint` function which accepts a constraint function and the specific variant for which it is valid. Constraints are automatically evaluated by the library and either force the variant to return an ∞ value during the offline training phase (thus ensuring that variant is not selected), or revert back to the default variant during the online deployment phase. In the example of Figure 2, the constraint `__dia_cutoff` ensures that the `__dia` variant does not get executed if the constraint evaluates to false.

C. Nitro Autotuner Interface

The Nitro Autotuner uses an external Python interface to allow users to precisely control various aspects of the autotuner and the tuning process for each variant. The interface exposes the `autotuner` and `code_variant` classes which can be used to configure tuning options globally, and for each code variant, respectively. Table II shows the various configuration options available. Most of these options have a default value, and the only essential

Table II
CONFIGURATION OPTIONS IN THE NITRO AUTOTUNER INTERFACE.

Option	Description
`classifier`	Classifier Object to Use (Default: `classifier_svm`)
`parallel_feature_evaluation`	Enable/Disable Parallel Feature Evaluation
`parallel_constraint_evaluation`	Enable/Disable Parallel Constraint Evaluation
`constraints`	Enable/Disable Constraints
`async_feature_eval`	Enable/Disable Asynchronous Feature Evaluation
`feature_selection`	Enable/Disable Feature Selection
Tuning Algorithm	**Description**
`tune`	Default, trains on entire training input
`itune`	Incremental tuning, optional *iter* or *acc* parameters

```
from nitro.autotuner import *
from nitro.code_variant import *

import glob

# Set tuning properties for spmv
spmv = code_variant("spmv", 6)
spmv.classifier = svm_classifier()
spmv.constraints = False
spmv.parallel_feature_evaluation = False
spmv.constraints = True
spmv.async_feature_eval = False

tuner = autotuner("spmv")

# Set global tuning properties
matrices = glob.glob("inputs/training/*.mtx")
tuner.set_training_args(matrices)
tuner.set_build_command("make")
tuner.set_clean_command("make clean")

# Tune
tuner.tune([spmv])
```

Figure 3. Example Nitro Autotuner interface for SpMV.

information that must be provided is the training input data set and the functions to be tuned. The remaining functionality allows the expert user to optionally control the tuning process as desired.

Figure 3 shows a tuning script for the SpMV example. A single `code_variant` object is created (named `'spmv'`) and both global and variant-specific tuning properties are set. The call to the `tune` method starts the autotuning process.

Tuning options specified using this interface are written out to a header file so that the autotuner can communicate with the C++ part of the system. Generating a static header file also enables us to use the C++ template mechanism to selectively generate relevant code.

III. NITRO AUTOTUNER IMPLEMENTATION

This section elaborates on the functionality of the Nitro Autotuner. We describe how it builds a model for variant selection and its optimizations and heuristics to reduce the overhead of training and feature evaluation.

A. Building a Model for Variant Selection

As mentioned in the previous section, the Nitro Autotuner automatically constructs a model for variant selection using SVMs, a form of supervised classification. Supervised classification utilizes a set of labeled training examples to infer a function that maps new, unseen input instances to their correct labels. A set of training examples of the form $\langle \mathbf{x}_i, y_i \rangle$ is provided, where each $\mathbf{x}_i$ refers to a feature vector and y_i refers to the corresponding label for $\mathbf{x}_i$. In our case, the label set is integers in the range $\{0, 1, ...|V|-1\}$, where V is the set of variants. During the training phase, for each training input i with corresponding feature vector $\mathbf{x}_i$, the Nitro Autotuner performs exhaustive search over the code variants and assigns to label y_i the integer designating the variant that leads to the best performance. The result of the training phase is a classification model that predicts the appropriate label for a new, unseen feature vector.

Nitro uses the Radial-Basis Function (RBF) [18] kernel to perform classification by default. The features are scaled to the range $[-1, 1]$, and subsequently a cross-validation based parameter search is performed to find the kernel parameters.

B. Incremental Tuning to Reduce Training Inputs

The execution time of code variants is difficult to predict in general, and can often be very high for certain inputs. Coupled with the fact that programmers may provide a large number of redundant training instances, the training phase can often become unacceptably time consuming. To reduce the number of training inputs required for the training phase, the Nitro Autotuner supports *incremental tuning*, which enables Nitro to perform exhaustive search of variants on only a subset of the training inputs.

A key observation is that the execution time required to derive feature vectors is typically far lower than the cost of actually executing variants. Therefore, we compute feature vectors for all the given inputs, and compute output labels using exhaustive search (which requires running all variants for that input) for only a small subset of the inputs and then

select additional inputs to add to the training set to improve the model.

For this purpose, we employ Active Learning [19], an iterative learning technique. We provide an initial training set consisting of i labeled input instances, with at least one input that has the label of each variant. An additional j unlabeled input instances ($j >> i$) provides the *active pool* for active learning. Using the feature vectors, Nitro then iteratively picks new training instances to label using the Best-vs-Second-Best active learning heuristic for SVMs proposed in [20]. At each iteration, Nitro updates the model.

When using incremental tuning, Nitro requires a stopping criteria to determine when the number of training inputs is sufficient to construct an accurate model. As shown in Table II, the incremental tuning algorithm is selected by invoking `itune`, with either a number of iterations *iter* or an accuracy threshold *acc*. Limiting the number of iterations is useful when the number of training inputs is too large for Nitro to evaluate. For problems whose decision boundaries are of moderate complexity, our experience shows that 20-25 iterations is usually sufficient to build a good model (see Section V-B). Alternatively the accuracy threshold with respect to the test input is useful if the all of the test inputs have known labels. The tuner then runs automatically, checking the prediction performance at each step on the test set, and then converges when the model reaches this accuracy. For the benchmarks in this paper, we were able to achieve considerable reductions in training times using this strategy for incremental tuning (see Section V-B).

C. Optimizing Feature and Constraint Evaluation

As additional optimizations, Nitro can also (1) parallelize feature and constraint evaluation; and, (2) start executing feature functions asynchronously. The latter mode returns control to the main thread immediately and thus allows the overlap of other computation with feature evaluation so that some of the feature evaluation time may be amortized. Calling the variant while in asynchronous mode introduces an implicit barrier, ensuring the correct evaluation of all features before variant execution. These two modes are currently implemented in Nitro using the Intel TBB [21] library.

IV. Benchmarks

Figure 4 lists the benchmarks we use to evaluate Nitro's effectiveness, including a description of the set of variants, the features used, and number of inputs for training and test data sets. All of these benchmarks are derived from high-performance CUDA libraries that already included code variants. Further, for each benchmark, the best-performing code variant varies according to properties of the input data. By using existing high-performance libraries, we are able to focus the experiment on the small amount of additional code required to integrate Nitro and deriving the features to be used in variant selection. The training and test inputs come from standard sources, as described, and the training inputs are not included in the test inputs. Further, we choose training inputs such that all variants are well-represented in the training set for each benchmark.

Sparse Matrix-Vector Multiplication (SpMV). As described in Section II, SpMV is a critical operation that is used in many iterative methods for solving large-scale linear systems. For this experiment, we use the CUSP library [22] to provide the code variants for SpMV. We use 3 features related to the matrix row lengths (average non-zeros per row, standard deviation of the row lengths, and deviation of the longest row from the average row length), and 2 features that estimate the padding required for the DIA and ELL formats (DIA and ELL fill-in). A training set consisting of 54 matrices from the UFL Sparse Matrix collection [23] was used. For the 100 matrices in the test set, we selected 10 matrices each from a set of 9 groups in the UFL collection at random (with the exception of the Williams group, which has only 7 matrices in the UFL collection), and generated 13 matrices related to stencils.

Linear Solvers and Preconditioners. Many large-scale scientific simulations such as computational fluid dynamics (CFD) and structural mechanics [24] involve solving partial differential equations (PDE) systems. Typically, solution to a PDE-based system involves solving the underlying sparse linear system using software toolkits [25], [26]. One of the challenges in effectively using such toolkits is the selection of an appropriate $\langle$ linear solver, preconditioner $\rangle$ combination as this selection impacts both the performance and convergence of the computation. For this experiment, we use 6 (linear solver, preconditioner) combinations from the CULA Sparse toolkit [26], which is a GPU library for solving large sparse linear systems. We select features for this benchmark based on the work by Bhowmick et al. [27]. These features reflect different numerical properties of sparse matrices such as *trace* and *1-norm*.

We use symmetric sparse matrices from the UFL Sparse Matrix collection to represent sparse linear systems. We use 26 and 100 matrices in the training and testing set, respectively.

Breadth-First Search (BFS). BFS is used as a basis for algorithms that analyze sparse relationships (such as social networks and electronic design automation) represented as graphs. Using Nitro, we select variants from a set of highly optimized BFS implementations for GPUs [28], part of a larger set of GPU primitives provided in the Back40 Library [29]. We consider a set of six variants provided in the library, which are designed for different types of input graphs. The library includes a seventh variant, named Hybrid that tries to dynamically combine the strengths of the CE-Fused and 2-Phase Fused kernels. Matching the performance of the Hybrid variant was one of our goals. We use a set of 5 graph features: number of vertices and edges, average

Benchmark	Variants	Description	Features	(#Training i/ps, #Testing i/ps)
SpMV	CSR-Vec	Performs SpMV on CSR-formatted matrices. Assigns a warp to each row.	AvgNZPerRow, RL-SD, MaxDeviation, DIA-Fillin, ELL-Fillin	(54, 100)
	DIA, ELL	Perform SpMV on DIA and ELL formatted matrices.		
	CSR-Tx, DIA-Tx, ELL-Tx	Same as above variants, but input vector cached in texture memory.		
Solvers	CG-Jacobi, CG-Bjacobi, CG-Fainv	Conjugate gradients method with Jacobi, Blocked Jacobi and Factorized Approximate Inverse preconditioners.	NNZ, Nrows, Trace, DiagAvg, DiagVar, DiagDominance, LBw, Norm1	(26, 100)
	BiCGStab-Jacobi ,BiCGStab-Bjacobi, BiCGStab-Fainv	BiConjugate gradients Stabilized method with Jacobi, Blocked Jacobi and Factorized Approximate Inverse preconditioners.		
BFS	EC-Fused, EC-Iter	Expand incoming vertex frontier, filter, and produce outgoing vertex frontier. Fused version invokes single kernel that steps through BFS iterations. Iterative version invokes a separate kernel for each BFS iteration.	AvgOutDeg, Deg-SD, MaxDeviation, Nvertices, Nedges	(20, 148)
	CE-Fused, CE-Iter	Contract incoming edge frontier, filter, and produce outgoing edge frontier.		
	2-Phase-Fused, 2-Phase-Iter	Isolates vertex expansion and edge contraction workloads into separate kernels.		
Histogram	Sort-ES, Sort-Dynamic	Sort data first, and then do a quick run-length detection. Even-Share (ES) version assigns an even share of inputs to thread blocks, dynamic uses a queue.	N, N/#Bins, SubSampleSD	(200, 1291)
	Global-Atomic-ES Global-Atomic-Dynamic	Compute Histogram using global atomic add operations.		
	Shared-Atomic-ES Shared-Atomic-Dynamic	Compute Block-level Histogram using shared memory atomicAdd, and then reduce to final Histogram.		
Sort	Merge Sort	Merge sort from ModernGPU library.	N, Nbits, NAscSeq	(120, 600)
	Locality Sort	Locality sort from ModernGPU library.		
	Radix Sort	Radix sort from CUB.		

Figure 4. A brief description of variants and list of features used for each benchmark. The last column lists the sizes of training and testing sets.

out-degree, standard deviation of the degree of each node, and deviation of the node with the highest out-degree from the average out-degree. The training set for BFS consists of a set of 20 graphs. We then test the performance of the Nitro-tuned version on 148 graphs in the DIMACS10 group in the UFL Sparse Matrix collection. We run 100 randomly-sourced BFS traversals for each graph to evaluate each variant. Further, we use the traversed edges per second (TEPS) as the optimization metric.

Histogram. A Histogram operation counts the number of observations that fall into one of a set of disjoint categories or 'bins'. Histograms are very commonly used as building blocks in more complex algorithms in a number of domains, especially image processing. We use the variants implemented in the CUDA Unbound (CUB) [30] library for this benchmark.

We evaluate three variants and two grid-mapping strategies thus giving rise to six code variants. We use 3 features: length of the input sequence, average number of elements per bin, and the standard deviation of a sub-sequence of the input sequence (SubSampleSD in Figure 4). We construct a 256-bin histogram for grayscale images, with pixel values ranging from 0 to 255. For training and testing, we use the images from the INRIA Holidays Dataset [31] (converted to grayscale). Out of the 1491 images in the dataset, 200 are used for training and the rest for testing.

Sort. Sorting is used as a building block in a myriad of algorithms and methods. We use 3 high-performance GPU sorting algorithms: Merge Sort, Locality-Optimized Segmented Sort, and Radix Sort as variants for this benchmark. The Merge and Locality Sorts are part of the ModernGPU [32] library of GPU primitives, while the Radix Sort implementation is provided in CUB [30]. We use a set of 3 features: length of the input sequence, number of bits in the input data type, and the number of ascending sub-sequences of the input.

Sorting is performed on 32 and 64-bit floating point keys. We train a combined model for both data types and report performance numbers achieved on a test set consisting of both types of data. The training set consists of 60 sequences for each data type, thus giving us a total of 120 instances. For testing, we use a total of 600 sequences, 300 for each data type. Further, each of the 300 instances is divided into 3 categories, 100 consisting of uniformly random keys, 100 consisting of reverse sorted keys, and 100 consisting of almost sorted keys. We also tried replacing the uniformly random keys with keys drawn randomly from the Standard Normal and Standard Exponential distributions, but the performance was identical. The "almost-sorted" category is generated by taking a sorted sequence and randomly swapping 20-25% of the keys. Key lengths are varied from 100K to 20M keys.

V. Results

We run these benchmarks on a system with an Intel Core i7 930 processor with 4 GB of RAM. The graphics card used is an NVIDIA Tesla C2050 (Fermi).

To evaluate the effectiveness of Nitro, we first compare the average performance of variants selected using Nitro with the best variants selected using exhaustive search. In all

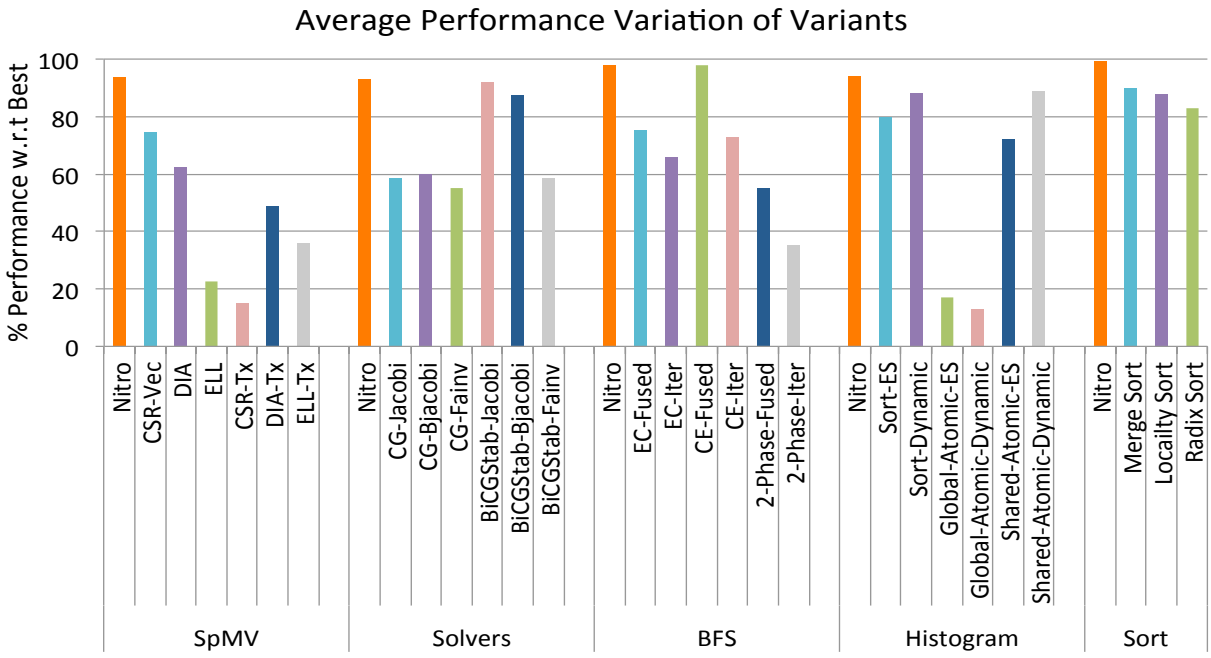

Figure 5. Performance Variation Among Variants

benchmarks, the test set we use to compare performance is much larger than the training set used to train the classifier. We do this to evaluate whether the model generalizes well to new inputs. We also evaluate the performance of the training time reduction heuristic and provide an analysis of the performance variation with respect to features.

A. *Variant Selection*

Figure 5 shows the performance of individual variants with respect to the performance achieved by the best variants (shown as 100% in the figure), on average, for each of the 5 benchmarks with their respective test sets. Also included in the figure is a comparison with the performance achieved by variants tuned by Nitro. In all benchmarks, the Nitro-tuned variants achieve within 7% of the performance achieved by the best variants.

Sparse Matrix-Vector Multiplication. The first bar in Figure 6 shows the tuning results for SpMV. On average, SpMV selected through Nitro achieved a performance 93.74% compared to the variants selected through exhaustive search. Further, we notice that over 90% of the input matrices achieve 70% or more of the performance of exhaustive search, and close to 80% of the input matrices achieve 90% or more performance.

We notice a few data points lying below the 70% mark as well. Poor performance on these matrices is mainly due to the significant performance penalty of mispredicting. In most cases, this is because DIA was chosen incorrectly, or because Texture-Cached was not chosen when it should have been. This may be improved with additional or more representative features: we currently do not have a feature designed to capture when the Texture-Cached variant should be selected.

Linear Solvers and Preconditioners. For the second benchmark in Figure 6, on average the variants selected using Nitro perform at 93.23% of the best performing variants. This average number is for 94 matrices as no variant was able to solve linear systems represented by 6 matrices, i.e., the variants did not converge to a solution. Additionally, the results indicate that of the 94 test matrices, there were 35 for which at least one variant did not converge. The Nitro version successfully selected a converging variant 33 out of the 35 times. We can thus make the following observation: Nitro not only predicts a high-performance variant, but also selects a converging one with high accuracy.

Breadth-First Search. For the third benchmark in Figure 6, the average performance of the variant selected by Nitro with respect to variants selected by exhaustive search is 97.92%.

We observed that one of CE-Fused or 2-Phase-Fused was almost always selected for all the graphs we tested on. Further, 2-Phase-Fused seemed to perform relatively well for most graphs with high average out-degrees, but poorly compared to the CE-Fused kernel for graphs with relatively low average out-degrees. Both these observations correspond with the results observed in Merrill et al. [28]. Due to the relatively simple decision boundary between variants in this experiment, Nitro selected variants were able to achieve very high performance using just 20 training data instances.

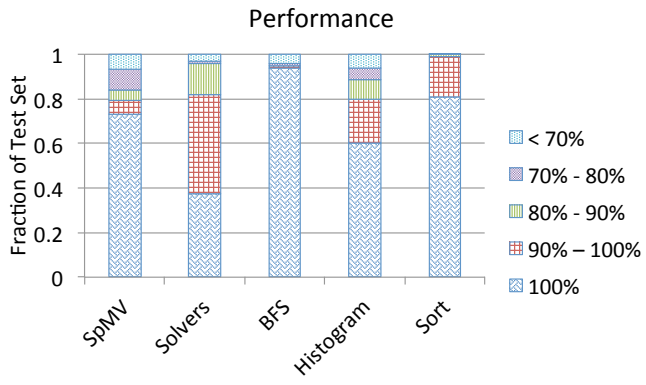

Figure 6. Performance comparison across all test inputs.

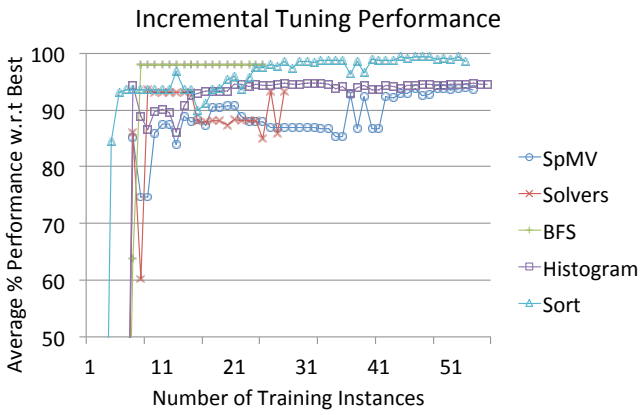

Figure 7. Convergence for active learning training heuristic.

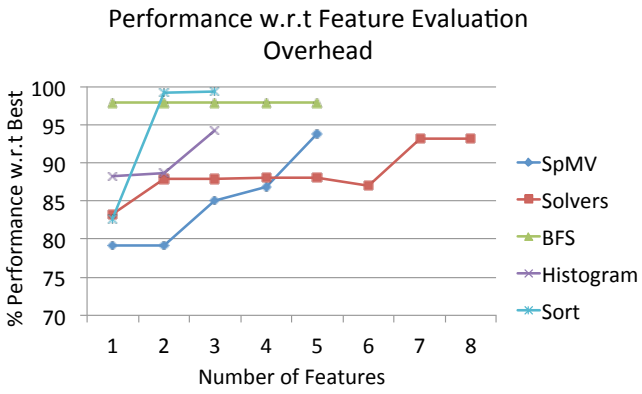

Figure 8. Performance variation as features with higher evaluation overhead are added incrementally.

The Hybrid variant proposed in Merrill et al. [28] tries to dynamically combine the strengths of the CE-Fused and 2-Phase-Fused kernels. The Nitro-tuned version was able to beat the performance of the Hybrid version by 11% on average. Even though the Hybrid kernel performs well uniformly across different inputs, we noticed that it was almost always slightly slower than the best variant for a given input (average performance was 88.14% of the best variant). This is possibly due to the dynamic nature of the Hybrid kernel.

Histogram. For the fourth benchmark in Figure 6 the average performance achieved by the variants tuned with Nitro with respect to the best variants is 94.16%. We observe that the tuned variant performs reasonably well across different input distributions. The global and shared atomic variants, however, perform well only when the data is uniformly distributed. For non-uniformly distributed data, the high latency of atomic-add operations on GPUs coupled with the high number of concurrent threads trying to update a small number of bins causes the global and shared atomic variants (especially the global atomic variant) to experience a performance drop.

Sort. The last bar in Figure 6 shows the tuning results for the Sort benchmark. The Nitro-tuned variants achieve an average performance of 99.25% with respect to the best variants.

We observed from our experiments that while Radix Sort performs exceedingly well for the 32-bit keys, its performance is surpassed by Merge and Locality Sorts in the 64-bit case. In particular, for almost sorted sequences, Locality Sort performs best. From Figure 5, it is also clear that on average, the Nitro-selected variant performs better than all the other variants, irrespective of data type.

B. Training Time Reduction

As described in Section III, Nitro supports the option of incremental tuning when there is possibly redundant training data and/or the variants take a long time to execute. Figure 7 shows that the number of iterations required by incremental tuning to reach within 90% of the performance achieved without incremental tuning is roughly 25 iterations. To match the performance achieved by using the full training set, incremental tuning takes no more than 50 iterations, and can be achieved in less than 20 iterations for all but the SpMV benchmark. Another observation from the figure is that sometimes additional training data leads to a decrease in performance, and more iterations are needed for convergence. Even with carefully chosen training data, the incremental tuning algorithm uses only a fraction of this data to achieve comparable performance to tuning on the full training set.

C. Feature Evaluation Overhead

Figure 8 shows the variation in performance as features with higher evaluation overhead are added incrementally. We notice that in case of the Sort and Solver benchmarks, removing the feature with the highest evaluation overhead (Presortedness and Left Bandwidth, respectively) has little

effect on final performance. In the case of BFS, we notice that performance depends almost entirely on the Average Out-Degree (shown as Feature 1 in the graph).

Using this pruned feature set thus results in almost negligible feature evaluation overhead for the BFS and Sort benchmarks (since we are only left with $O(1)$ features). In Histogram, the most expensive feature (Feature 3 in the graph) computes the standard deviation of a sub-sample of the input. The default size for this is 25% of the size of the input sample, or 10,000 elements, whichever is lower. From our experiments, we noticed that evaluation overhead for this feature can be brought down to less than 0.1% of the time taken by the variant on average by simply decreasing the size of the sub-sample, at the cost of slightly decreased overall performance.

In the remaining benchmarks (SpMV and Solvers), it is evident that getting peak performance requires evaluating the more expensive features. However, this cost is amortized for SpMV as we compute the feature vector only once and execute the SpMV operation multiple times. For Solvers, feature vector computation takes place only once, and is amortized over hundreds or thousands of solver iterations.

VI. Related Work

Several programmer-directed autotuning frameworks support tuning of code/algorithmic variants. Petabricks [2] supports user specification of *transforms* that are analogous to functions. Transforms are automatically composed together to form hybrid algorithms using a compiler framework and an adaptive algorithm [33]. Petabricks, however, implicitly tunes variants for the size of the input data set. Our framework, on the other hand, can tune based on any user-defined characteristic of the input data. Brewer [5] describes a code variant selection system that uses linear regression to predict the performance of individual variants based on input parameters. The variant with the lowest predicted run time is then selected. Sequoia selects variants with user guidance for recursive algorithms that target the memory hierarchy [1].

Apart from code variants, a number of systems support the expression and tuning of optimization parameters. Such systems can be adapted for code variant generation and tuning using parameterized templates which specify how to generate new variants based on the actual values of the parameters in the template (found through search). Examples of such systems include Active Harmony [4] (integrated with the CHiLL loop transformation framework [34] to generate variants), POET [35], and Orio [36]. Since parameter tuning cannot capture the algorithm variants used in our study, this work is complementary to our approach.

In addition to general-purpose frameworks, various autotuning systems and techniques have been built to aid in the development of efficient and portable applications for specific domains. Examples of such systems include ATLAS [37], PhiPAC [38], and OSKI [11] for linear algebra, [39], FFTW [40] and SPIRAL [41] for signal processing, [42], [43], [44] for stencil computations, and [45] for sorting.

The general problem of algorithm selection was first formally stated and studied by Rice in 1976 [6]. Vuduc [7] provides an evaluation of statistical learning techniques in the context of algorithm selection. Lagoudakis and Littman [8] model the algorithm selection problem as a Markov Decision Process and use Reinforcement Learning techniques to solve it. Guo proposes the use of Bayesian Networks to learn the mapping from input features to code variants [9]. Petabricks uses a bottom-up evolutionary algorithm named INCREA [33] which builds a tuned algorithm for a specific problem size by incrementally composing tuned algorithms for smaller problem sizes. Other work in this area includes [46], [47], [48], [49]. Luo et al [50] propose a system for code variant selection based on input sizes and compare the prediction performance of various classifiers. Many of these techniques can be integrated into Nitro's learning sub-system, thus replacing/augmenting the SVM-based technique currently employed.

VII. Conclusions

This paper has presented Nitro, which is a programmer-directed autotuning framework that employs supervised learning to select code variants based on features of their input data set. The support in Nitro for deriving classification models of input data sets is particularly important for irregular applications, where the best version of a computation is heavily affected by the structure of the input. On five high-performance GPU applications, variants tuned using Nitro achieve over 93% of the performance of variants selected through exhaustive search, averaged over the testing inputs. Further, we demonstrate an incremental tuning mode for Nitro that achieves substantial reduction in the training set size. With the initial framework in place, we envision a number of directions to expand on Nitro's capability. The features we use in this paper are expressed by an expert programmer, but the framework could easily support additional features that are added implicitly by the system, such as architectural features, or features derived from compiler analysis. We also plan to incorporate into Nitro optimization parameters common to most autotuning systems, and integrate it with compiler-based autotuning. Our long-term goal is a mainstream autotuning framework that supports both expert users and the general programming community.

Acknowledgements

We would like to thank NVIDIA Corporation for generous equipment donations, and members of the NVIDIA research group including Duane Merrill and Sean Baxter for discussions on tuning the BFS, Histogram, and Sort benchmarks.

This research was funded by DARPA contract HR0011-13-3-0001.

REFERENCES

[1] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan, "Sequoia: programming the memory hierarchy," in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, ser. SC '06, 2006.

[2] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe, "Petabricks: a language and compiler for algorithmic choice," in *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '09, 2009, pp. 38–49.

[3] C. Tapus, I.-H. Chung, and J. Hollingsworth, "Active harmony: Towards automated performance tuning," *Supercomputing, ACM/IEEE 2002 Conference*, pp. 44–44, Nov. 2002.

[4] A. Tiwari, C. Chen, J. Chame, M. Hall, and J. Hollingsworth, "A scalable auto-tuning framework for compiler optimization," in *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, May 2009, pp. 1–12.

[5] E. A. Brewer, "High-level optimization via automated statistical modeling," in *Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, ser. PPOPP '95, 1995, pp. 80–91.

[6] J. R. Rice, "The algorithm selection problem," *Advances in Computers*, vol. 15, pp. 65–118, 1976.

[7] R. Vuduc, J. W. Demmel, and J. A. Bilmes, "Statistical models for empirical search-based performance tuning," *Int. J. High Perform. Comput. Appl.*, vol. 18, no. 1, pp. 65–94, Feb. 2004.

[8] M. G. Lagoudakis and M. L. Littman, "Algorithm selection using reinforcement learning," in *Proceedings of the Seventeenth International Conference on Machine Learning*, ser. ICML '00. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2000, pp. 511–518.

[9] H. Guo, "A bayesian approach for automatic algorithm selection," in *Proceedings of the IJCAI Workshop on AI and Autonomic Computing: Developing a Research Agenda for Self-Managing Computer Systems*, 2003.

[10] N. Thomas, G. Tanase, O. Tkachyshyn, J. Perdue, N. M. Amato, and L. Rauchwerger, "A framework for adaptive algorithm selection in STAPL," in *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, ser. PPoPP '05, 2005, pp. 277–288.

[11] 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, 2005.

[12] D. Guo and W. Gropp, "Optimizing Sparse Data Structures for Matrix-Vector Multiply," *International Journal of High Performance Computing Applications*, vol. 25, pp. 115–131, 2011.

[13] R. Vuduc and H. Moon, "Fast Sparse Matrix-Vector Multiplication by Exploiting Variable Block Structure," in *Proceedings of the High Performance Computing and Communications, volume 3726 of LNCS*. Springer, 2005, pp. 807–816.

[14] E. Im, K. A. Yelick, and R. Vuduc, "Sparsity: Optimization Framework for Sparse Matrix Kernels," *Int. J. High Perform. Comput. Appl.*, vol. 18, no. 1, pp. 135–158, 2004.

[15] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," in *SC '09: Proc. Conference on High Performance Computing Networking, Storage and Analysis*, Nov. 2009.

[16] V. N. Vapnik, *Statistical learning theory*. Wiley, New York, 1998.

[17] C.-C. Chang and C.-J. Lin, "LIBSVM: A library for support vector machines," *ACM Transactions on Intelligent Systems and Technology*, vol. 2, pp. 27:1–27:27, 2011.

[18] M. D. Buhmann, *Radial Basis Functions*. New York, NY, USA: Cambridge University Press, 2003.

[19] B. Settles, "Active learning literature survey," University of Wisconsin–Madison, Computer Sciences Technical Report 1648, 2009. [Online]. Available: http://axon.cs.byu.edu/ martinez/classes/778/Papers/settles.activelearning.pdf

[20] A. Joshi, F. Porikli, and N. Papanikolopoulos, "Multi-class active learning for image classification," in *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, june 2009, pp. 2372 –2379.

[21] J. Reinders, *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. O'Reilly, 2007.

[22] N. Bell and M. Garland, "Generic parallel algorithms for sparse matrix and graph computations," 2009. [Online]. Available: http://code.google.com/p/cusp-library/

[23] T. Davis, "The University of Florida Sparse Matrix Collection," *ACM Transactions on Mathematical Software*, vol. 38, pp. 1:1–1:25, 2011.

[24] M. A. Heroux, P. Raghavan, and H. D. Simon, *Parallel Processing for Scientific Computing (Software, Environments and Tools)*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2006.

[25] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith, "Efficient management of parallelism in object oriented numerical software libraries," in *Modern Software Tools in Scientific Computing*, E. Arge, A. M. Bruaset, and H. P. Langtangen, Eds. Birkhäuser Press, 1997, pp. 163–202.

[26] E. Photonics and Nvidia, "CULA | sparse," http://www.culatools.com/.

[27] S. Bhowmick, B. Toth, and P. Raghavan, "Towards low-cost, high-accuracy classifiers for linear solver selection," in *Proceedings of the 9th International Conference on Computational Science: Part I*, ser. ICCS '09, 2009, pp. 463–472.

[28] D. Merrill, M. Garland, and A. Grimshaw, "Scalable GPU graph traversal," in *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '12, 2012, pp. 117–128.

[29] D. Merrill and et al., "Back40 computing," 2012. [Online]. Available: http://code.google.com/p/back40computing/

[30] D. Merrill, "Cuda unbound (cub)," http://nvlabs.github.io/cub/.

[31] H. Jégou, M. Douze, and C. Schmid, "Hamming embedding and weak geometric consistency for large scale image search," in *European Conference on Computer Vision*, ser. LNCS, A. Z. David Forsyth, Philip Torr, Ed., vol. I. Springer, oct 2008, pp. 304–317.

[32] S. Baxter, "Modern GPU," http://nvlabs.github.io/moderngpu/.

[33] J. Ansel, M. Pacula, S. Amarasinghe, and U.-M. O'Reilly, "An efficient evolutionary algorithm for solving bottom up problems," in *Annual Conference on Genetic and Evolutionary Computation*, Dublin, Ireland, July 2011.

[34] C. Chen, "Model-guided empirical optimization for memory hierarchy," Ph.D. dissertation, University of Southern California, May 2007.

[35] Q. Yi, K. Seymour, H. You, R. Vuduc, and D. Quinlan, "Poet: Parameterized optimizations for empirical tuning," in *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, 2007, pp. 1–8.

[36] A. Hartono, B. Norris, and P. Sadayappan, "Annotation-based empirical performance tuning using Orio," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Rome, Italy, 2009.

[37] J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel, "Optimizing matrix multiply using PHiPAC: A portable, high-performance, ANSI C coding methodology," in *International Conference on Supercomputing*, 1997, pp. 340–347.

[38] R. Whaley and D. Whalley, "Timing high performance kernels through empirical compilation," in *International Conference on Parallel Processing*, 2005, pp. 89–98.

[39] F. de Mesmay, Y. Voronenko, and M. Puschel, "Offline library adaptation using automatically generated heuristics," in *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, April 2010, pp. 1–10.

[40] M. Frigo and S. G. Johnson, "The fastest Fourier transform in the West," MIT Lab for Computer Science, Tech. Rep. MIT-LCS-TR728, 1997.

[41] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo, "SPIRAL: Code generation for DSP transforms," *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, vol. 93, no. 2, pp. 232– 275, 2005.

[42] M. Christen, O. Schenk, and H. Burkhart, "Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures," in *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, May 2011, pp. 676–687.

[43] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick, "Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures," in *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, ser. SC '08. Piscataway, NJ, USA: IEEE Press, 2008, pp. 4:1–4:12. [Online]. Available: http://dl.acm.org/citation.cfm?id=1413370.1413375

[44] S. Kamil, C. Chan, L. Oliker, J. Shalf, and S. Williams, "An auto-tuning framework for parallel multicore stencil computations," in *International Parallel and Distributed Processing Symposium*, 2010.

[45] X. Li, M. Garzaran, and D. Padua, "Optimizing sorting with genetic algorithms," in *Code Generation and Optimization, 2005. CGO 2005. International Symposium on*, March 2005, pp. 99–110.

[46] K. Leyton-Brown, E. Nudelman, G. Andrew, J. McFadden, and Y. Shoham, "A portfolio approach to algorithm select," in *Proceedings of the 18th International Joint Conference on Artificial Intelligence*, ser. IJCAI'03. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003, pp. 1542–1543.

[47] L. Kotthoff, I. P. Gent, and I. Miguel, "A preliminary evaluation of machine learning in algorithm selection for search problems," in *Fourth Annual Symposium on Combinatorial Search*, 2011.

[48] L. Lobjois and M. Lemaître, "Branch and bound algorithm selection by performance prediction," in *Proceedings of the Fifteenth National/Tenth Conference on Artificial Intelligence/Innovative Applications of Artificial Intelligence*, ser. AAAI '98/IAAI '98. Menlo Park, CA, USA: American Association for Artificial Intelligence, 1998, pp. 353–358.

[49] A. Guerri and M. Milano, "Learning techniques for automatic algorithm portfolio selection," in *Proceedings of the 16th Eureopean Conference on Artificial Intelligence, (ECAI 2004)*. IOS Press, 2004, pp. 475–479.

[50] L. Luo, Y. Chen, C. Wu, S. Long, and G. Fursin, "Finding representative sets of optimizations for adaptive multiversioning applications," in *In 3rd Workshop on Statistical and Machine Learning Approaches Applied to Architectures and Compilation (SMART09), colocated with HiPEAC09 conference*, 2009.