Architecture-Adaptive Code Variant Tuning

Saurav Muralidharan
University of Utah
sauravm@cs.utah.edu

Amit Roy
University of Utah
aroy@cs.utah.edu

Mary Hall
University of Utah
mhall@cs.utah.edu

Michael Garland
NVIDIA Corporation
mgarland@nvidia.com

Piyush Rai
IIT Kanpur
piyush@cse.iitk.ac.in

Abstract

Code variants represent alternative implementations of a computation, and are common in high-performance libraries and applications to facilitate selecting the most appropriate implementation for a specific execution context (target architecture and input dataset). Automating code variant selection typically relies on machine learning to construct a model during an offline learning phase that can be quickly queried at runtime once the execution context is known. In this paper, we define a new approach called architecture-adaptive code variant tuning, where the variant selection model is learned on a set of source architectures, and then used to predict variants on a new target architecture without having to repeat the training process. We pose this as a multi-task learning problem, where each source architecture corresponds to a task; we use device features in the construction of the variant selection model. This work explores the effectiveness of multi-task learning and the impact of different strategies for device feature selection. We evaluate our approach on a set of benchmarks and a collection of six NVIDIA GPU architectures from three distinct generations. We achieve performance results that are mostly comparable to the previous approach of tuning for a single GPU architecture without having to repeat the learning phase.

Categories and Subject Descriptors C.4 [*Performance of Systems*]: Modeling Techniques; D.2.8 [*Metrics*]: Performance Measures; D.3.4 [*Processors*]: Optimization

Keywords autotuning; cross-architectural tuning; input-adaptive; multi-task learning; device feature selection

ASPLOS '16, April 02-06, 2016, Atlanta, GA, USA
Copyright © 2016 ACM 978-1-4503-4091-5/16/04. . . $15.00
DOI: http://dx.doi.org/10.1145/2872362.2872411

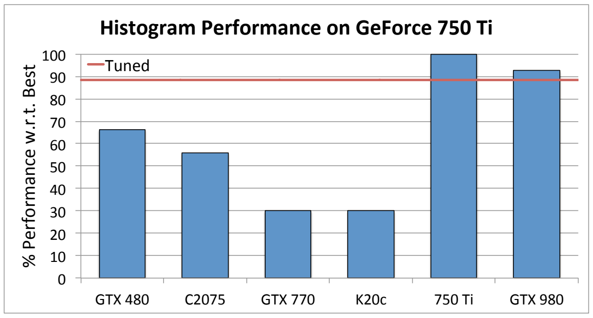


Figure 1: Histogram performance on the GeForce 750 Ti when trained on other architectures. The tuned line shows the performance of our system when trained using data from all architectures other than 750 Ti.

1. Introduction

Modern high-performance computing hardware has grown increasingly complex with the proliferation of deep memory hierarchies, hierarchical parallelism and latency vs. throughput-optimized processor cores. Achieving high performance on such systems thus often requires the use of automatic performance tuning (*autotuning*) software [6, 18, 32, 37, 38]. Autotuners systematically navigate a search space of possible implementations of a computation to find the implementation(s) that best meets a specific optimization criteria, usually performance. *Code variants* in such systems represent alternative implementations of a computation. Each code variant has the same interface, and is functionally equivalent to the other variants but may employ fundamentally different algorithms or implementation strategies.

Prior work on code variant selection and related code optimization systems have successfully employed machine

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.

learning to develop an optimization model during an offline training phase [1, 11, 16, 23, 24, 29, 30, 35]. Such a model can be queried to perform variant selection at runtime once properties of the input dataset are available. These systems, however, require the variant selection model(s) to be re-trained every time the software is installed on a new architecture or if the underlying hardware is upgraded. This training process is typically very time-consuming and heavy on system resources; we are required to evaluate each variant v for each input i when collecting the training data. This paper evaluates the following question: *Can we develop a methodology to reuse results of training on two or more source architectures to create a variant selection model for a different target architecture without training on the target architecture?* While the overall approach we present in this paper is general, we simplify the problem by focusing on only NVIDIA GPUs.

As a motivating example, consider the Histogram operation: it counts the number of observations that fall into one of a set of disjoint bins. Consider the six code variants for Histogram in the high-performance GPU CUB library [26], which are described in Figure 3. There are two variants that do not use atomic operations, two that use global memory atomics and two that use shared memory atomics. The best variant is therefore *architecture-sensitive*, based on the relative performance of atomic operations, and also *input-sensitive*, e.g., affected by input size and mean sample distribution.

Figure 1 shows performance for Histogram on the GeForce 750 Ti GPU (Maxwell), when using a variant selection model trained on six different GPU architectures. The x-axis captures results when trained on the corresponding GPU. The y-axis represents percentage performance achieved by the variant selected by a model with respect to the best-performing variant (exhaustive search), averaged across all inputs in a test dataset. From the figure, it is clear that while variant selection models trained and tested on the same architecture perform well (above 95% of exhaustive search), this is not the case when models trained on architecture X are deployed on architecture Y (X $\neq$ Y), with performance dropping to as low as 30% of exhaustive in some cases.

While an architecture-specific model yields high performance, the time-consuming training phase must be repeated for each application and target architecture. In this paper, we instead develop a system that automatically constructs code variant selection model(s) on a target architecture using only training data from a set of source architectures specified by the programmer, together with information that characterizes each architecture. On the target, no variants are executed during the model construction process, since no training data from the target is required. Our system thus enables the construction of performance-portable software that quickly and automatically adapts to both changing inputs and new hardware architectures. In Figure 1, the line labeled 'Tuned' shows performance achieved by our system trained on data from every architecture except the 750 Ti.

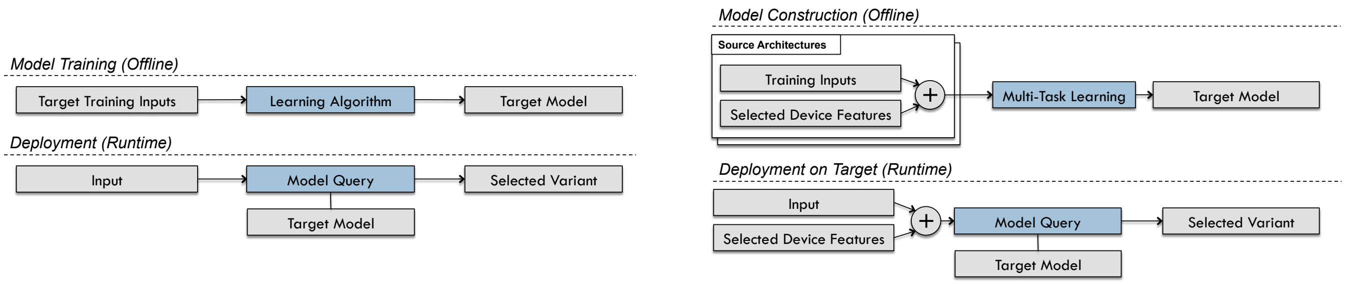
We treat the cross-architectural tuning problem as a *multi-task* [8] learning problem, where each separate task denotes an architecture. Features that characterize each architecture (hereafter referred to as *device features*) are collected automatically (a one-time operation) on each architecture. Device features not relevant to the application in question are pruned away. The resulting device features are then used in the multi-task learner to come up with variant selection model(s) for the target architecture.

This paper makes the following contributions: (1) it develops the first automated approach (to our knowledge) to cross-architecture autotuning, which uses multi-task learning to develop a model on a target architecture from training on different source architectures; (2) it summarizes a wealth of empirical data for six computations and six GPU architectures across three distinct generations that captures reasons behind architectural sensitivity to code variant selection; and, (3) it demonstrates that device feature selection is valuable in building a successful code variant selection model on new architectures, discussing the strengths and limitations of the approach.

2. System Overview

The automated system described in this paper extends the Nitro autotuning framework [29]. Nitro provides a library interface that permits expert programmers to express code variants along with meta-information that aids the system in selecting among the set of variants at runtime. Figure 2(a) illustrates the approach in Nitro. A learning algorithm – Support Vector Machine (SVM) classifier by default – constructs a code variant selection model on the target architecture as a result of an offline training phase on the same architecture. For each architecture, training data has the form $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_N, y_N)\}$, where each $\mathbf{x}_i$ represents an input feature vector and each y_i represents the best variant for that input. When presented with a new, unseen input at runtime, the model predicts the best variant to use.

Figure 2(b) shows how we have extended Nitro to support architecture-adaptive tuning. We can omit the training data collection on the target architecture by using previously-collected training data from one or more source architectures. To capture the signature of the target architecture and its relationship to the source architectures, we rely on *device features*, listed in Table 1. On NVIDIA platforms, these features are obtained in three possible ways: most are discovered instantaneously using the built-in `deviceQuery` program bundled with the CUDA toolkit. `Static` device features are easily-obtained published specifications that augment what is returned by `deviceQuery`. If any other features are needed, then `Custom` features can be added. We observed that there were no features that captured the cost of atomic operations, which vary significantly across GPU generations. Therefore,



(a) Overview of the original Nitro system.

(b) Architecture-adaptive tuning overview.

Figure 2: Comparison of the original Nitro system with architecture-adaptive tuning. When tuning across architectures, values of the device features selected through DFS are obtained on both the source (during model construction) and target (during deployment). These are then concatenated with feature values of the relevant input data point ('+' operator in the figure).

	Fermi		Kepler		Maxwell	
Feature	**480**	**C2075**	**770**	**K20c**	**750**	**980**
deviceQuery						
`global_mem (GB)`	1.5	5.2	4.0	4.7	2.0	4.0
`cuda_cores`	480	448	1536	2496	640	2048
`clock_rate (MHz)`	1401	1147	1110	706	1268	1216
`mem_clock_rate (MHz)`	1848	1566	3505	2600	2700	3505
`mem_bus_width (bits)`	384	384	256	320	128	256
`l2_cache_size (KB)`	768	768	512	1280	2048	2048
`shared_mem_per_block (KB)`	48	48	48	48	48	48
`copy_engines`	1	2	1	2	1	2
Static						
`peak_gbps`	177.4	144.0	224.0	208.0	86.4	224.0
`peak_gflops_sp`	1345	1030	3213	3520	1389	4612
`peak_gflops_dp`	168	515	134	1170	43	156
Custom						
`shared_atomic (msec)`	0.193	0.238	0.281	0.361	0.011	0.006
`global_atomic (msec)`	0.402	0.488	0.034	0.051	0.063	0.036

Table 1: Values of GPU device features for 6 architectures.

we added to Nitro two microbenchmarks that measure this; other microbenchmarks could be added to the `Custom` set as needed. Device feature values for the six GPU architectures we consider in this paper are also listed in Table 1.

We pose the problem of architecture-adaptive tuning as a *multi-task learning* (MTL) problem. MTL algorithms learn multiple tasks simultaneously to capture intrinsic relatedness between tasks. In our system, each separate architecture is represented as a task, and inter-task relationships are learned using MTL algorithms. We use *feature concatenation* for MTL, which derives the code variant selection model for the target architecture and is formally described in Section 3.1. In earlier stages of this research, we implemented and explored other MTL algorithms such as weighted kernels and probabilistic SVMs [7], but found that variant selection performance was far more affected by device feature selection than MTL algorithms.

We have discovered that using the full set of 13 device features does not yield the most accurate predictions, and which features are most relevant to code variant selection is application-specific. Therefore, our system performs *device feature selection (DFS)* to pinpoint the small number of device features relevant to the current application.

	480		C2075		770		K20c		750		980	
	H	**S**	**H**	**S**	**H**	**S**	**H**	**S**	**H**	**S**	**H**	**S**
480	1	1	0.9	0.8	0.3	-0.4	0.6	0.3	-0.8	-0.6	0.2	-0.4
C2075	0.9	0.8	1	1	0	-0.8	0.5	-0.3	-0.6	0	-0.3	-0.8
770	0.3	-0.4	0	-0.8	1	1	0.9	0.8	-0.8	-0.6	0.3	1
K20c	0.6	0.3	0.5	-0.3	0.9	0.8	1	1	-1	-1	-0.1	0.8
750	-0.8	-0.6	-0.6	0	-0.8	-0.6	-1	-1	1	1	0	-0.6
980	0.2	-0.4	-0.3	-0.8	0.3	1	-0.1	0.8	0	-0.6	1	1

Table 2: Cosine similarity between architectures for Histogram (*H*) and SpMV (*S*). Values closer to +1 indicate similarity, while values closer to -1 indicate dissimilarity.

Each code variant stresses different components of the hardware architecture, such as the DRAM subsystem, floating-point performance, parallelism, machine balance, etc. To demonstrate that device feature selection is application-specific, Table 2 approximates the similarity between architectures for two benchmarks: Histogram and Sparse Matrix-Vector Multiplication (SpMV). Each entry in the table corresponds to the *cosine-similarity* (cosine of the angle between vectors) between device feature vectors of the corresponding architectures. Thus, values closer to +1 indicate similarity, while values closer to -1 indicate dissimilarity. Note that the optimal set of device features for both the benchmarks are different, since Histogram and SpMV variants stress different components of the hardware architecture. Thus, two architectures which are very similar for the SpMV computation may be completely different for Histogram. For example, the entry corresponding to (C2075, 750) shows that for Histogram, the C2075 and 750 are quite dissimilar (a fact confirmed in Figure 1), while the same pair of architectures is relatively similar for the SpMV benchmark.

3. Tuning Process

Our system employs a two-phase device feature selection (DFS) strategy to automatically find the best-performing subset of device features (in terms of final variant selection performance) for each computation. These selected device features may then be used by a multi-task learning algorithm to automatically construct variant selection models. The following subsection describes the process of model training using the feature concatenation technique. The subsections that follow describe how the multi-task learner constructs variant selection models on the target architecture using (1) all device features; (2) device features found by profile DFS (P-DFS); and, (3) device features found by performing cross-validation search on the output of P-DFS.

3.1 Model Construction using MTL

The feature concatenation strategy for multi-task learning appends device features to input features and builds an SVM model based on this new training dataset. More formally, let there be M source architectures and N training inputs. Further, let $\{\mathbf{a}_1, \mathbf{a}_2, ..., \mathbf{a}_M\}$ denote device feature vectors for each of the M source architectures. Then, for a given source architecture s, the corresponding training set is:

$$T_s = \{([\mathbf{x}_1 \circ \mathbf{a}_s], y_{s1}), \ldots, ([\mathbf{x}_N \circ \mathbf{a}_s], y_{sN})\}$$

where $\{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$ is the set of N input feature vectors from the training set, and each y_{si} denotes the label (best variant) for training input i on architecture s; $[\circ]$ denotes vector concatenation. The full training set is then: $T = \bigcup_{s=1}^{M} T_s$, which is used to train an SVM classifier. During testing, the device features of the target architecture are concatenated with the input features before querying the model.

3.2 Utilizing the Full Set of Device Features

A straightforward solution to the architectural tuning problem is to feed the entire device feature set to the multi-task learner when it builds the variant selection model for the target. In this subsection, we describe how this naïve strategy works.

Source Architecture Side On the source architectures, when the user invokes the autotuning system, input features and corresponding variant labels are collected automatically, as in the original Nitro system [29]. This information is also recorded in a repository, to be retrieved when needed by target architectures. The device feature values for the source architecture in question are also collected and recorded in the repository.

Target Architecture Side On the target architecture, the user invokes a function in the autotuner, which automatically (1) retrieves the data collected from the source architectures from the repository; and (2) collects device feature values of the target. Each training input from the source architectures is of the form $\langle I, v \rangle$, where I represents an input feature vector and v represents the label of the best variant for that input. Using this together with device feature values for each source architecture, a variant selection model for the target architecture is constructed as explained in Section 3.1.

3.3 Profile Device Feature Selection (P-DFS)

With a restricted set of source architectures, extraneous device features can confuse the multi-task learner, as demonstrated in Section 6.2. We now describe an improvement over using the full set of device features called profile DFS (P-DFS), which uses the profiling data of the variants of a computation to predict the device features most relevant to that computation.

Application Proxies An application proxy is a small program that takes an *intensity* value ϕ as input, ranging from 0 to 5, and produces a GPU kernel with roughly ϕ `* 20%` instructions of a particular kind. The first column of Table 3 lists the application proxies used by our system. Thus, the `SP-GFLOP` proxy generates kernels with single-precision floating point instructions, the `ATOMIC` proxy generates kernels with atomic add instructions and so on. As a concrete example, when the `SP-GFLOP` proxy is provided an intensity value of 2, the proxy generates a CUDA kernel with roughly 40% single-precision floating point arithmetic instructions. The following code snippet shows the generated kernel code:

```
// 6 loads and stores, 4 floating-point instructions
A[i] = A[i+1]*beta + alpha;
A[i+1] = A[i+2]*beta + alpha;
A[i+2] = A[i+3];
```

Here, `A` is an array of type `float32`, `alpha` and `beta` are scalars (also of type `float32`), and `i` is the array index.

Each proxy P_j, where j ranges from 1 to 5 (total number of proxies) is associated with a set of device features F_j, representing the hardware component(s) that it stresses. The `ATOMIC` proxy, for example, is associated with the `shared_atomic` and `global_atomic` features. The first and last columns in Table 3 list these associations for each proxy.

Application Proxy Profiling For each proxy P_j, the system automatically collects tuples of the form $\langle C_{\phi j}, \phi \rangle$, where $C_{\phi j}$ represents the profiling data of a single run of proxy P_j, and ϕ is the intensity with which it is run. Each proxy has a subset of relevant profiling metrics, which are also listed in Table 3 (column 3). Running a proxy at every intensity from 0 to 5, we obtain a set of $\langle C_{\phi j}, \phi \rangle$ tuples that can be used to train a machine learning model. A model is built for every proxy, which can then be queried with profiling data of code variants.

Source Architecture Side The P-DFS approach requires collecting the following data on at least one source architecture: (1) profiling metrics C_v of each code variant v on each training input; and (2) profiling metrics of application proxies at different intensities $\langle C_{\phi j}, \phi \rangle$. Thus, in addition to invoking the autotuner as described in Section 3.2, the user is required to initiate profiling data collection. This automatically collects all the required profiling data and stores it in the repository.

Target Architecture Side On the target, the construction of variant selection models proceeds as in Section 3.2. How-

Proxy	Description	Profiling Metrics	Device Features
`SP-GFLOP`	Single precision floating-point	`flop_count_sp, inst_fp_32, flop_sp_efficiency`	`peak_gflops_sp, cuda_cores, clock_rate`
`DP-GFLOP`	Double precision floating-point	`flop_count_dp, inst_fp_64, flop_dp_efficiency`	`peak_gflops_dp, cuda_cores, clock_rate`
`ATOMIC`	Atomic operation latency	`atomic_transactions_per_request, atomic_transactions, l2_atomic_-transactions, l2_atomic_-throughput, atomic_throughput`	`global_atomic, shared_atomic`
`MEM-BW`	Global memory bandwidth	`l1_cache_global_hit_rate, l1_cache_local_hit_rate, gld_-transactions, gst_transactions, local_load_transactions, local_store_transactions, gld_transactions_per_request, gst_transactions_per_request, local_load_transactions_per_-request, local_store_-transactions_per_request, stall_memory_dependency, gld_-efficiency, gst_efficiency, l2_l1_read_hit_rate, l2_read_-transactions, l2_write_-transactions, dram_read_-transactions, dram_write_-transactions, l2_l1_read_-transactions, l2_l1_write_-transactions, l2_utilization`	`peak_gbps, mem_clock_rate, mem_bus_width, l2_cache_size`
`SH-MEM-BW`	Shared memory bandwidth	`shared_load_transactions, shared_store_transactions, shared_load_transactions_per_-request, shared_store_throughput`	`shared_mem_per_block`

Table 3: GPU application proxies with corresponding profiling metrics and device features.

ever, this time, only device features selected by the P-DFS system are used for training the variant selection model.

Algorithm 1 provides an overview of the P-DFS process. The profiling data for the proxies at various intensities $\langle C_{\phi j}, \phi \rangle$ is first retrieved from the repository. This is then used to construct a set of models, one for each proxy. If we denote the model for proxy P_j as λ_j, then querying λ_j with the profiling metrics of a variant C_v will yield the intensity value corresponding to P_j for the variant. By querying each proxy model using the profiling data of the variants in the computation (also retrieved from the repository), and examining the predicted intensity values, the best proxies can be found. These are recorded for each input and in the final step, a majority voting scheme is used to select a global best set of proxies. The device features associated with the winning proxies (last column of Table 3) is returned as output of the P-DFS system.

3.4 Cross-Validation Device Feature Selection (CV-DFS)

Although device features obtained as a result of P-DFS are relevant to the computation in question, there may still be

Algorithm 1 Profile Device Feature Selection

```
1:  ▷ V: Set of variants
2:  ▷ I: Set of training inputs
3:  ▷ P: Set of application proxies
4:  global_best ← $\{\}$
5:  for $v \in V$ do
6:      ▷ For each kernel in variant $v$
7:      for $k \in$ kernels[$v$] do
8:          ▷ For each training input
9:          for $i \in I$ do
10:             intensity ← $\{\}$
11:             for $p \in P$ do
12:                 ▷ Profiling data for kernel $k$ on input $i$
13:                 $t$ ← profile[$k, i$]
14:                 ▷ Predict intensity for proxy $p$ on profile
15:                 intensity[$p$] = intensity-predict($t, p$)
16:             best_proxies[$i$] = $\{x$ : intensity[$x$]is highest$\}$
17:         ▷ Add best proxies across inputs to global best
18:         global_best $\cup =$ majority-vote(best_proxies)
    return global_best
```

Algorithm 2 Cross-Validation Device Feature Selection

```
1:  ▷ S: Set of source architectures
2:  ▷ D: Set of device features from P-DFS
3:  global_best ← {}
4:  ▷ For each source architecture
5:  for $s \in$ S do
6:      best_accuracy ← 0
7:      best_set ← ∅
8:      ▷ Assign a temporary target
9:      target ← $s$
10:     sources ← $S - \{s\}$
11:     $T_s$ ← {training-data($x$): $x \in$ sources}
12:     $T_t$ ← training-data(target)
13:     for $d \in$ subsets[D] do
14:         ▷ Get device feature values of source and target
15:         $DF_s$ ← {df-values($d, x$): $x \in$ sources}
16:         $DF_t$ ← df-values($d$, target)
17:         ▷ Train MTL model using $T_s$ and $DF_s$
18:         model ← mtl-train($T_s, DF_s$)
19:         ▷ Predict and calculate accuracy w.r.t. $T_t$
20:         accuracy ← predict(model, $T_t, DF_t$)
21:         if accuracy > best_accuracy then
22:             best_accuracy ← accuracy
23:             best_set ← $d$
24:     ▷ Record best features and their frequencies
25:     global_best ∪ = best_set
26: ▷ Return the $k$ most frequently occurring features
27: return most-frequent(global_best, $k$)
```

extraneous features that confuse the variant selection model on the target. To obtain an even more pruned and relevant set of device features, we employ a cross-validation DFS (CV-DFS) strategy. An overview of CV-DFS is provided in Algorithm 2.

CV-DFS is performed on the target architecture, and does not require any extra data collection on the source architectures (over P-DFS). The algorithm proceeds by assigning one of the source architectures as a temporary target (Line 9 in Algorithm 2). Then, with the remaining source architectures, every subset of device features (currently restricted to size three) is exhaustively used to build a variant selection model for the temporary target (Line 18, **mtl-train** function) and performance of this model on the temporary target's training data is evaluated (Line 20, the **predict** function). This process is iteratively performed for each source architecture, and the k device features that perform best over all source architectures are chosen. By default, k is set to one (i.e., return the best device feature).

CV-DFS relies on the assumption that device features that yield good prediction performance on source architectures are likely to be good predictors on the target for the same computation. As demonstrated in Section 6, this assumption holds for most applications.

```python
from nitro import *
import glob

histogram = code_variant("histogram", 6)
# Record training data in store
histogram.record = True
histogram.device_id = "gtx_480"
# Create autotuner instance
tuner = autotuner("histogram")
inputs = glob.glob("training/*.jpg")
tuner.set_training_args(inputs)
# Tune for current architecture
tuner.tune([histogram])
```

Listing 1: Histogram tuning example - source architecture.

```python
from nitro import *

histogram = code_variant("histogram", 6)
histogram.profiling_based_dfs = True
histogram.search_based_dfs = True
# Create autotuner instance
tuner = autotuner("histogram")
# Build model from source data
tuner.tune_from_source([histogram])
```

Listing 2: Histogram tuning example - target architecture.

4. Implementation

The Nitro framework [29] provides C++ and Python interfaces for code variant tuning. Variants, input features, and optional constraints are specified using the C++ interface within the application, while a separate Python script is used to customize the tuning process. For the system described in this paper, we extend Nitro's Python tuning interface with additional functions and options.

The function `tune_from_source` automatically builds models for the target architecture using source training data and device feature values. We have implemented a storage system for variant training data using `Redis` [34]. The variant name, together with the device identifier is used to index into the store, where the variant training data, optional profiling data (for both the variants and proxy applications), and device feature values are kept. The `tune_from_source` function automatically retrieves the right data and builds the models. Users have the option of toggling both P-DFS (using the `profiling_based_dfs` knob) and CV-DFS (using the `search_based_dfs` knob). If P-DFS is enabled, then per-input profiling data must also be collected on at least one of the source architectures (using the `profile` function). Listings 1 and 2 provide examples of how this interface is used on the source and target sides, respectively.

5. Benchmarks

Figure 3 lists the benchmarks we use to evaluate our system's effectiveness, including a description of the set of variants, the features used, and number of inputs for training and test datasets. All of these benchmarks are derived from high-performance CUDA libraries that already included code variants. By using existing high-performance libraries, we are able to focus the experiment on the small amount of additional code required to apply our automated system to these benchmarks. 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.

Histogram Histograms are very commonly used as building blocks in a number of domains, especially image processing. We use the variants implemented in the high-performance CUDA Unbound (CUB) library [26]. We evaluate three variants and two grid-mapping strategies, thus giving rise to six code variants. We use three features. 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 [21] (converted to grayscale). Out of the 1491 images in the dataset, 200 are used for training and the rest for testing.

Sparse Matrix-Vector Multiplication (SpMV) SpMV is used in many iterative methods for solving large-scale linear systems. For this experiment, we use the variants provided by the CUSP library [4]. We use 5 features and a training set consisting of 54 matrices from the UFL Sparse Matrix collection [15]. 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.

Sort 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 [3] library of GPU primitives, while the Radix Sort implementation is provided in CUB [26].

Sorting is performed on 32 and 64-bit floating point keys. We train a combined model for both data types and report performance 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. 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.

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. Variants are selected from a set of highly optimized BFS implementations for GPUs described in [28], part of a larger set of GPU primitives provided in the Back40 Library [25]. We consider a set of six variants provided in the library, which are designed for different types of input graphs. We use a set of 5 features. The training set for BFS consists of a set of 20 graphs and the test set consists of all the 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 traversed edges per second (TEPS) as the optimization metric.

Linear Solvers and Preconditioners Many large-scale scientific simulations such as computational fluid dynamics (CFD) and structural mechanics [20] involve solving partial differential equations (PDE) systems. Typically, solutions to a PDE involve solving the underlying sparse linear system using software toolkits [2, 31]. 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 $\langle$linear solver, preconditioner$\rangle$ combinations from the CULA Sparse toolkit [31], which is a GPU library for solving large sparse linear systems. Features used for this benchmark are based on the work by Bhowmick et al. [5]. We use symmetric sparse matrices from [15] to represent sparse linear systems.

Matrix Transposition In-place transposition of square matrices is a well-studied problem. Transposition of a non-square matrix is a much more involved process, requiring $O(mn \log mn)$ work. Catanzaro et al. [10] describe a set of in-place matrix transposition algorithms which perform the operation in $O(mn)$ time. These algorithms are packaged as an open-source library [9]. We use four variants from this library for our experiment: two for general row-to-column and column-to-row transposition, and another two specialized for skinny matrices. We use four features related to the dimensions of the matrix. Matrix dimensions are chosen from a uniform-random distribution with the constraint that the matrix fits in the memory of the GTX 480 GPU (the GPU with lowest memory capacity). The matrices are populated with 64-bit double precision values. 194 such matrices are used for training and 1000 for testing.

6. Results

We run our experiments on six NVIDIA GPUs characterized by the device features in Table 1: (1) GeForce GTX 480, (2) Tesla C2075, (3) GeForce GTX 770, (4) Tesla K20c, (5) GeForce 750 Ti, and (6) GeForce GTX 980. As shown in Table 1, these graphics cards span three GPU architecture families: Fermi, Kepler, and Maxwell. We use CUDA Toolkit version 6.5 for our experiments[1]. The host system is an Intel Core i7-4770 CPU (3.4 Ghz) with 32 GB of RAM. GPU profiling metrics are collected using the `nvprof` tool. Each profiling metric is normalized with respect to the total number of issued instructions in the GPU kernel.

[1] Except for Solvers, which requires CUDA 6.0 (due to CULA).

Benchmark	Variants	Description	Features	Description	(#Training, #Testing) I/Ps
SpMV	CSR, CSR-VEC	Performs SpMV on CSR-formatted matrices. CSR assigns a thread to each row. CSR-Vec assigns a warp to each row.	`AvgNZPerRow, RL-SD, MaxDeviation` `DIA-Fillin, ELL-Fillin`	Features related to row length. Fillin ratio for DIA and ELL formats.	(54, 100)
	DIA, ELL	Perform SpMV on DIA and ELL formatted matrices.			
Solver	CG-Jacobi, CG-Bjacobi, CG-Fainv	Conjugate gradients method with Jacobi, Blocked Jacobi and Factorized Approximate Inverse preconditioners	`NNZ, #Rows` `Trace` `DiagAvg, DiagVar, DiagDominance` `LBw` `Norm1`	Number of nonzeros and rows. Trace of a matrix. Features related to diagonal elements of a matrix. Left bandwidth of a matrix. 1-norm of a matrix.	(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-Iterative	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` `#Vertices, #Edges`	Features related to graph out-degree. Number of vertices and edges.	(20, 138)
	CE-Fused, CE-Iterative	Contract incoming edge frontier, filter, and produce outgoing edge frontier.			
	2-Phase-Fused 2-Phase-Iterative	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`	Sequence of length and average length. Standard deviation of sub-sample.	(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`	Input size.	(120, 600)
	Locality Sort	Locality sort from ModernGPU library.	`NBits`	32 or 64 bits.	
	Radix Sort	Radix sort from CUB.	`#AscSeq`	# Ascending sub-sequences.	
Transpose	R2C, C2R	R2C, C2R variants from inplace library	`M, N` `RowMajor` `CoPrime`	Number of rows, columns. Is in row-major layout. Are M and N co-prime.	(194, 1000)
	Skinny R2C Skinny C2R	R2C, C2R specialization for skinny matrices			

Figure 3: Variants and features used for each benchmark. The last column lists the sizes of training and testing sets.

6.1 Architecture Sensitivity of Benchmarks

We first ask the question whether architecture differences significantly impact code variant selection. For this purpose, we identify the best variant (found through exhaustive search) for each input in the testing set across all benchmarks and architectures. Figure 4 provides a measurement of the architectural sensitivity of each benchmark. Here, the x-axis is the set of benchmarks, and the y-axis is the percentage of test inputs for which at least one architecture selects a different best variant than the others. In other words, it is the percentage of test inputs for which the exact same variant of the benchmark was not selected across all architectures. Figure 5 is similar, except it depicts architectural sensitivity of benchmarks within GPUs of each generation - namely, Fermi, Kepler, and Maxwell. Figure 5 shows that differences in variant selection are usually less pronounced within the same architectural family, but not always. Further data is shown in Table 4, where each sub-table represents a benchmark and a row represents the distribution of variant selection (via exhaustive search) across all test inputs for a particular architecture. We use these two figures and the table in the remainder of this subsection.

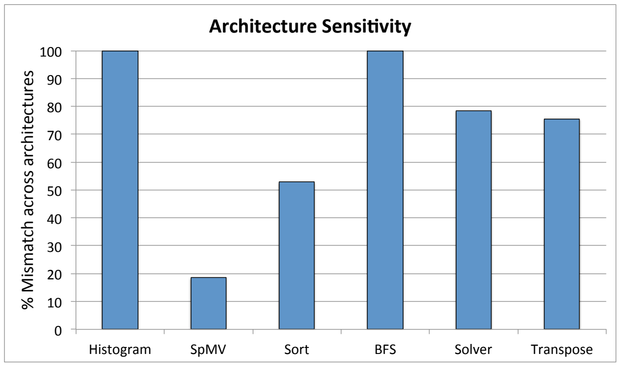


Figure 4: Architecture-sensitivity of each benchmark. The y-axis represents the percentage of test inputs for which at least one architecture selects a different best variant than the others.

From one generation to the next, new architecture features and machine configurations may dramatically affect

	SpMV			
	CSR	CSR-VEC	DIA	ELL
GTX 480	0.00	50.52	19.59	29.90
C2075	0.00	54.64	19.59	25.77
GTX 770	0.00	49.48	19.59	30.93
K20c	3.09	48.45	18.56	29.90
750 Ti	4.12	45.36	20.62	29.90
GTX 980	0.00	56.70	18.56	24.74

	BFS					
	EC-Fused	EC-Iter	CE-Fused	CE-Iter	2P-Fused	2P-Iter
GTX 480	0.00	0.70	90.91	4.90	1.40	2.10
C2075	0.00	0.71	90.00	7.86	1.43	0.00
GTX 770	0.71	59.29	0.71	37.86	0.00	1.43
K20c	24.64	0.00	75.36	0.00	0.00	0.00
750 Ti	13.04	0.00	86.96	0.00	0.00	0.00
GTX 980	4.29	0.00	94.29	0.00	1.43	0.00

	Transpose			
	R2C	C2R	Skinny R2C	Skinny C2R
GTX 480	44.10	55.10	0.00	0.80
C2075	45.50	53.80	0.00	0.70
GTX 770	42.50	56.90	0.00	0.60
K20c	25.10	74.30	0.00	0.60
750 Ti	45.00	54.40	0.00	0.60
GTX 980	32.90	66.40	0.00	0.70

	Solver					
	CG-J	CG-BJ	CG-FAI	BiCGStab-J	BiCGStab-BJ	BiCGStab-FAI
GTX 480	12.90	15.05	6.45	35.48	23.66	6.45
C2075	13.98	12.90	6.45	32.26	30.11	4.30
GTX 770	6.45	10.75	13.98	35.48	23.66	9.68
K20c	11.83	13.98	5.38	36.56	24.73	7.53
750 Ti	18.28	11.83	6.45	34.41	16.13	12.90
GTX 980	12.90	16.13	7.53	37.63	17.20	8.60

	Sort		
	Locality	Merge	Radix
GTX 480	3.50	27.17	69.33
C2075	7.50	47.00	45.50
GTX 770	2.50	26.00	71.50
K20c	1.67	36.00	62.33
750 Ti	33.50	4.50	62.00
GTX 980	39.00	17.17	43.83

	Histogram					
	Sort-ES	Sort Dynamic	GA-ES	GA-Dynamic	SA-ES	SA-Dynamic
GTX 480	0.64	36.30	0.00	0.00	7.13	55.93
C2075	0.00	33.73	0.00	0.00	26.92	39.34
GTX 770	0.08	73.56	4.49	0.40	0.24	21.23
K20c	0.00	85.18	0.08	0.00	4.49	10.26
750 Ti	0.00	0.00	0.00	0.00	97.84	2.16
GTX 980	0.00	0.00	0.00	0.00	0.16	99.84

Table 4: Variant selection histograms across different benchmarks and architectures. Each sub-table represents the distribution of variant selections across test data for a particular benchmark.

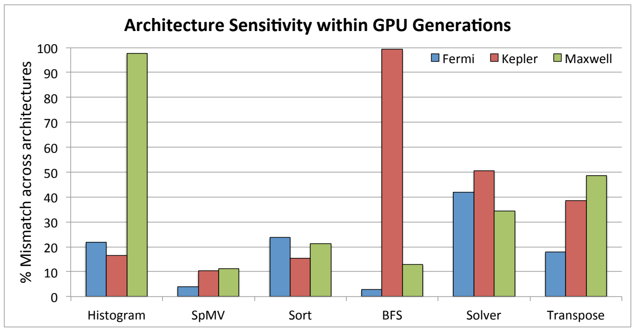


Figure 5: Architecture-sensitivity within GPUs of the same generation.

variant selection (e.g., support for atomic operations). But within a single generation, different selections are usually attributed to differences in (1) raw performance metrics (clock speed, memory bandwidth, floating point performance, etc.); or, (2) parallelism (number of cores). These architecture differences are captured in the device features of Table 1. From Figures 4 and 5 and Table 4, we see that Histogram reflects significant differences both across and within an architecture generation. The Maxwell generation devices (GTX 980 and 750 Ti) use the shared-atomic variants (SA-ES and SA-Dynamic) almost exclusively due to their low latency of shared memory atomics. However, these two devices rarely select the same shared-atomic variant, with the GTX 980 preferring SA-Dynamic and 750 Ti preferring SA-ES for most inputs. The Dynamic variants treat the input as a queue and atomically dequeue work in tiles for processing. Due to the reliance of these variants on atomics, the GTX 980 prefers them compared to the 750 Ti (the GTX 980's performance on atomics is nearly twice that of the 750 Ti, as Table 1 shows). The Kepler and Fermi devices predominantly use the Sort-Dynamic, SA-ES and SA-Dynamic variants, with the Kepler devices (GTX 770 and K20c) preferring the sorting-based variant over the shared-atomic ones. We believe that the slightly lower performance of shared atomics on Kepler when compared to the Fermi devices (GTX 480 and C2075) is the reason for this.

For BFS, most of the differences arise on the GTX 770 architecture. Specifically, the EC-Iterative and CE-Iterative variants are rarely selected by any architecture except the GTX 770. As described in Figure 3, the Iterative variants invoke a separate kernel for each BFS kernel, while the Fused versions use a single kernel to step through BFS iterations. Notice that `l2_cache_size` is a relevant device feature for BFS (Table 5, second column) and the GTX 770 has the lowest L2 cache size of all GPUs (Table 1). Since doing more work in a single kernel invocation typically increases L2 cache usage, we suspect that this is the reason for the GTX 770 preferring the Iterative variants over Fused ones.

Sort and Transpose exhibit architecture sensitivity, but not to the extent shown by Histogram and BFS and mostly across generations. The Maxwell generation of devices prefers to pick Locality sort over Merge sort, when compared to devices from other generations. The lower cost of atomic operations on Maxwell is most likely the reason for

Benchmark	Best Device Features	Proxies predicted by P-DFS	Best Feature by CV-DFS
Histogram	peak_gbps, shared_atomic, mem_bus_width	MEM-BW, ATOMIC	shared_atomic
SpMV	peak_gbps, mem_speed	MEM-BW, SH-MEM-BW	peak_gbps
Sort	global_atomic, l2_cache_size, shared_atomic	MEM-BW, ATOMIC	shared_atomic
BFS	global_atomic, shared_atomic, l2_cache_size, peak_gbps	MEM-BW, ATOMIC, SH-MEM-BW	shared_atomic
Solvers	global_atomic, shared_atomic, l2_cache_size, peak_gbps	MEM-BW, ATOMIC, SH-MEM-BW	shared_atomic
Transpose	global_atomic, shared_atomic, l2_cache_size, peak_gbps	MEM-BW, SH-MEM-BW	peak_gbps

Table 5: Best device features for each benchmark, proxies predicted by P-DFS, and the best features chosen by CV-DFS.

this, as Locality sort uses a dynamic work queue from which tasks are peeled off atomically. For Transpose, the bigger devices from the Kepler and Maxwell generations (the K20c and the GTX 980, respectively), tend to slightly prefer the C2R variant over R2C compared to other cards.

Finally, we notice from Table 4 that for the SpMV and Solver benchmarks, variants tend to be picked uniformly across architectures. We believe the primary reason for this is the fact that SpMV and Solver variants are optimized for various sparsity patterns of the input matrix and not necessarily for architecture-specific features; thus making them predominantly input-dependent. We were able to confirm this for the SpMV variants in the CUSP library by analyzing their source code, but not for the related Solver variants from CULA, which are closed-source.

6.2 Prediction Performance

First we look at how well device feature selection detects the relevant features for each benchmark. Table 5 shows the best subset of device features found by exhaustive search and by cross-validation search for each benchmark in the second and fourth columns, respectively. The third column shows the application proxies predicted by P-DFS for each benchmark. This exhaustive search finds the subset that yields best prediction accuracy on the target's test data. Since cross-validation DFS may predict a different subset of device features for every target, the last column of the table shows the device feature that occurs most frequently among all targets. We notice that P-DFS correctly predicts the proxies relevant to each benchmark. For example, it predicts that atomics are relevant to Histogram and BFS. Also, cross-validation search, guided by proxies found by P-DFS, discovers most of the important device features or nearby ones found via exhaustive search for all benchmarks. Another interesting observation is that although all the benchmarks we consider are predominantly memory bandwidth-bound, some benchmarks such as Histogram and BFS contain variants that rely on the use of global and shared-memory atomics. This reinforces our earlier point that the magnitude of architectural similarity is a function of the device features relevant to a benchmark's variants, and is not the same across all benchmarks.

Now we examine how well the different variant selection models derived from multi-task learning compare in their effectiveness against the original Nitro system (training and testing performed on the target architecture) and exhaustive search. In Figure 6, the benchmarks appear on the x-axis, with each bar representing a different target architecture (the remaining 5 architectures are used as sources). The y-axis shows percentage performance achieved compared to exhaustive search, as defined in the previous subsection. Bars labeled *Full Set* represent performance achieved when multi-task learning uses all device features, while the *P-DFS* and *CV-DFS* bars represent performance achieved by using device features selected by the profile DFS, and profiling followed by cross-validation DFS, respectively. While performing cross-validation DFS, we restrict the maximum size of each device feature subset to 3, since we found that increasing this beyond 3 rarely resulted in performance improvements. Also, we use the default value of 1 for the CV-DFS parameter k (number of most frequently occurring device features) in our evaluation. While we discovered that higher values suited certain benchmarks (for example, Histogram performs 2.5% better on average across all architectures when k is set to 2), we avoid varying k on a per-benchmark basis to remain consistent.

We expected the original Nitro bar would be an upper bound on performance, as it is training on the target architecture. Indeed we see a modest performance loss for Histogram. Performance is comparable for Transpose and Sort for all architectures, and BFS for Fermi and Maxwell generations but not Kepler. The reasons for these deviations in performance were explained in Section 6.1, but effectively they indicate instances where the learning phase did not see similar scenarios. Surprisingly, multi-task learning actually outperforms the original Nitro for the Solver and SpMV benchmarks on some architectures. This is a significant result considering the fact that we performed no training runs on the

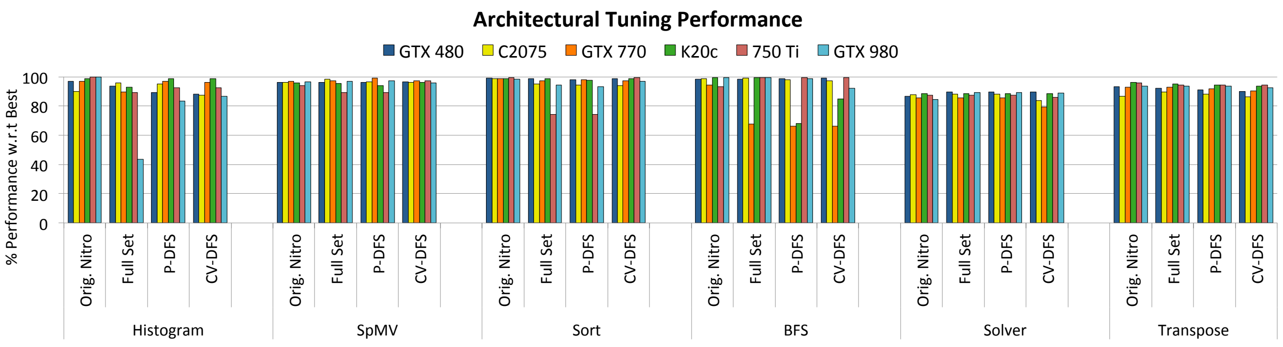


Figure 6: Device feature selection performance.

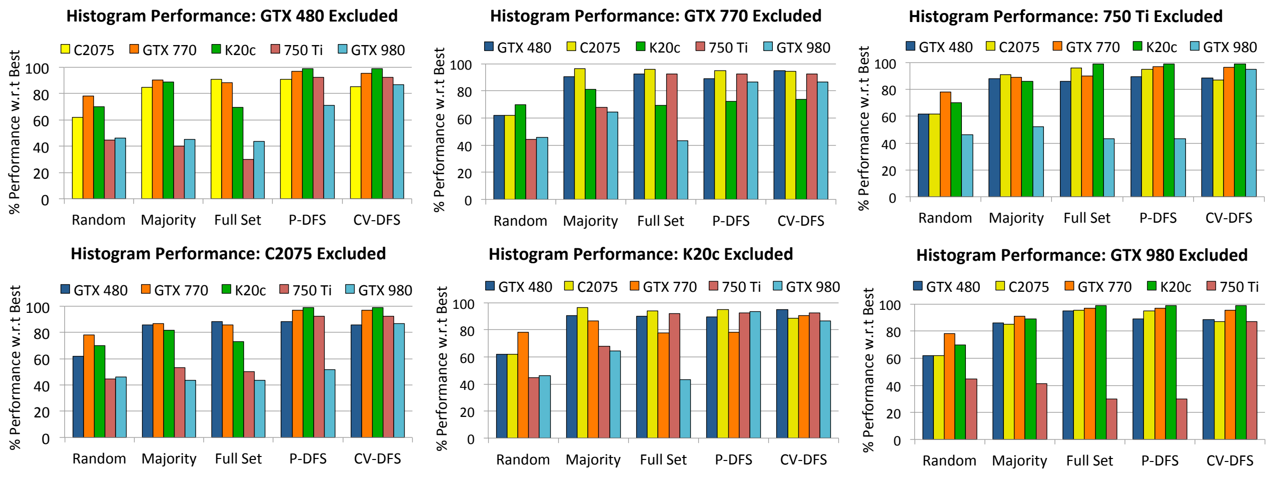


Figure 7: Device feature selection performance for Histogram on a restricted set of architectures.

target. It is an indication that multi-task learning is inferring useful relationships between similar architectures, thus effectively increasing the amount of training data available for model training compared to using only one architecture.

Now consider the differences between the three DFS strategies. Cross-validation yields the best performance for Histogram, SpMV, and Sort on almost all architectures, and is comparable to the other two DFS approaches for Solver and Transpose. The full set is preferable for the K20c version of BFS. The effect of using incorrect device features is more pronounced on a restricted set of source architectures. Space does not permit us to present our system's performance on all source:target combinations for all the benchmarks. However, to demonstrate how sensitive the performance is to the correct feature set, we perform the same experiment as above for Histogram, but iteratively remove one architecture from the total set - resulting in 4 source architectures instead of 5. This seemingly small change has substantial effects on performance.

Figure 7 shows the results for this experiment. Here, each sub-figure shows the performance of MTL with different device feature sets when a specific GPU is excluded from the list of architectures. In this experiment, we also compare against two simpler reference schemes: random selection and majority vote (*Random* and *Majority* in Figure 7, respectively). *Random* simply chooses a valid variant uniformly at random for each test input. It indicates the extent of input sensitivity, as it works well when variants have similar performance across inputs. We report the average of 1000 runs in this case for consistent results. *Majority* chooses the most frequently predicted variant among all source architectures for a given input as the predicted variant for the target. To accomplish this, a variant selection model for each source architecture is built separately using the original Nitro system. Since these two schemes do not make use of any architectural characteristics, their performance (especially on a set of restricted architectures) can be used to indicate and quantify the importance of device feature selection.

For all the source:target combinations, we notice a marked improvement in performance for *P-DFS* and *CV-DFS* over *Full Set*, demonstrating the importance of device feature selection. Further, the performance of *CV-DFS* is at least comparable to *P-DFS*, and often significantly better, especially on Fermi and Maxwell. In comparison, *Random* and

	P-DFS	CV-DFS
Histogram	4.70	100.50
SpMV	3.36	12.42
Sort	4.14	56.61
BFS	3.89	30.27
Solver	4.59	36.80
Transpose	4.26	48.24

Table 6: Device feature selection overhead (time in seconds).

Majority fare relatively poorly. In particular, the tendency of Maxwell devices (750 Ti and GTX 980) to strongly prefer the shared atomic variants over others (Table 4) seems to confuse the majority vote scheme. This is confirmed by the fact that removal of either of these devices improves *Majority* performance on all devices except the other device in the same generation. CV-DFS proves to be much more robust, and shows consistent performance across devices, even when devices from the same generation are removed. Overall, we notice that while majority vote performs well in simple cases, knowledge of architectural characteristics via device features is critical for robust performance.

6.3 Device Feature Selection Overhead

Table 6 shows the overhead incurred by P-DFS and CV-DFS[2] . As the table shows, P-DFS is fastest, since it primarily involves construction of the models for the various proxies, followed by querying the models on the profiling data of the code variants. CV-DFS takes longer, since all subsets of size $<= 3$ must be evaluated by the algorithm. Note that CV-DFS takes less time in comparison to gathering training data from source architectures, as we do not evaluate each $\langle I, v\rangle$ pair.

The CV-DFS strategy has a number of parameters that can be adjusted by the user. The value of these parameters can greatly affect the time taken to execute CV-DFS. Users can adjust the training subset size, feature subset size and the number of source architectures to use for CV. Reducing the values of any of these aforementioned parameters significantly reduces CV-DFS execution time. In our experiments we used the full training set, and feature subset sizes from 1 to 3 on all the source architectures.

6.4 Summary

Overall, multi-task learning produces results comparable to training on the target architecture in most cases, and even better results in a few cases. It falls short when the training data fails to capture a sufficiently similar scenario, and it improves from additional training data available from multiple sources. Finally, we observe that device feature selection improves performance particularly when less training data is available, and that in such cases, CV-DFS produces superior results but introduces more overhead than other approaches.

[2] Since the repository is stored on the local network, we do not include communication overheads.

7. Related Work

Performance counters have been used to predict and guide code tuning and compiler optimizations. Cavazos et al. [11] use performance counters to determine good compiler optimization settings. Machine learning is used to learn relationships between performance counter and optimal code optimization settings. Our framework, on the other hand, uses machine learning to build a relationship between performance counters and best device feature subsets, which are subsequently used in the cross-architectural tuning pipeline. Another system introduced by Parello et al. [30] uses performance counter data to systematically optimize programs by identifying performance anomalies. This system uses a decision tree to iteratively fix performance issues by applying optimization schemes to remedy the performance anomalies encountered.

Machine learning has been extensively used in guiding performance optimizations, as heuristics and exhaustive search are not practical. Supervised classification has been used to predict unroll factors to improve performance [35]. This problem can be seen as a variant selection problem where the selection depends on features extracted from the code itself. Our work specifically looks at variants whose performance depends on input dataset.

Apart from machine learning-guided optimizations, there has been prior work on input adaptivity. Ding et al. [16] address this problem by extending the PetaBricks language [1] to support input sensitivity. The language enables users to define domain-specific features of the input. The PetaBricks autotuner generates a number of implementations of the algorithm (each optimized for a specific input size). These implementations are used as variants that the machine learning algorithm predicts based on the given input features. Our framework focuses on streamlining the process of selecting variants and adapting selection across architectures and hence does not automatically generate variants but rather lets the user define the variants. G-ADAPT [23] is a framework that discovers decisions for GPU code optimizations. This system was subsequently adapted to handle input sensitivity. The G-ADAPT framework achieves this by employing machine learning (Regression Trees) to build a mapping from input to GPU code optimizations. This is achieved by an iterative search for the set of near-optimal optimization parameters for a given input. Both of the above frameworks address the input adaptivity problem but do not support cross-architectural adaptation.

Magni et al. [24] address the tuning of OpenCL code across architectures by applying a *thread-coarsening* transformation to the code. A machine learning technique is employed to predict the optimal coarsening factor for these transformations. Our framework does not apply transformations to the code but rather works with an existing set of variants and does not require training on the target architecture.

A number of systems support the tuning of optimization parameters. Such systems can aid in 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. Such systems include Active Harmony [36] (integrated with the CHiLL loop transformation framework [12] to generate variants), POET [39], Orio [19], Sequoia [33], the X-Language [17] and [27].

A number of frameworks aid in the development of efficient and portable applications for specific domains. Examples of such systems include ATLAS [38], PhiPAC [6], and OSKI [37] for linear algebra, FFTW [18] and SPIRAL [32] for signal processing, and [13],[14],[22].

In summary, to our knowledge we are the first to adapt code variant selection across architectures without re-training, formulating this as a multi-task learning problem.

8. Conclusions

This paper has presented a novel approach to cross-architecture autotuning, which uses multi-task learning to develop a model on a target architecture from training on different source architectures. On a set of benchmark applications and a collection of six NVIDIA GPUs from three distinct architecture generations, we achieve performance results comparable to the previous approach of tuning for a single architecture without having to repeat the learning phase, demonstrating the promise of multi-task learning for addressing performance portability across architectures. We view this work on variant selection as an initial step towards a more general approach to learning an optimization model on one set of resources and adapting to a different set of resources at runtime. Many questions remain: improving models for outliers, examining very different architectures, and other autotuning problems such as parameter selection. Tuning across different architecture classes such as CPU and GPU is particularly challenging, as higher-level device features (e.g. `gflops/gbps` ratio) and profiling metrics that remain valid across architecture classes must be used. These challenges will become increasingly important to future architectures, as complexity grows and systems become more dynamic.

Acknowledgments

We would like to thank NVIDIA Corporation for generous equipment donations, and members of the NVIDIA research group including Duane Merrill and Albert Sidelnik for valuable discussions. We would also like to thank the reviewers who suggested additional experiments to improve the paper. This research was funded by DARPA contract HR0011-13-3-0001.

References

[1] 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*, PLDI '09, pages 38–49, New York, NY, USA, 2009. ACM.

[2] 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*, pages 163–202. Birkhäuser Press, 1997.

[3] S. Baxter. Modern GPU library. `http://nvlabs.github.io/moderngpu/`.

[4] 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.

[5] 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*, ICCS '09, pages 463–472, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-01969-2.

[6] J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC: A portable, high-performance, ANSI C coding methodology. In *Proceedings of the 11th International Conference on Supercomputing*, ICS '97, pages 340–347, New York, NY, USA, 1997. ACM. ISBN 0-89791-902-5.

[7] E. V. Bonilla, F. V. Agakov, and C. K. I. Williams. Kernel multi-task learning using task-specific features. In *Proceedings of the 11th International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2007.

[8] R. Caruana. Multitask learning. *Mach. Learn.*, 28(1):41–75, July 1997. ISSN 0885-6125.

[9] B. Catanzaro. In-place matrix transposition. `https://github.com/bryancatanzaro/inplace`.

[10] B. Catanzaro, A. Keller, and M. Garland. A decomposition for in-place matrix transposition. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 193–206, New York, NY, USA, 2014. ACM.

[11] J. Cavazos, G. Fursin, F. Agakov, E. Bonilla, M. F. P. O'Boyle, and O. Temam. Rapidly selecting good compiler optimizations using performance counters. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '07, pages 185–197, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2764-7.

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

[13] M. Christen, O. Schenk, and H. Burkhart. PATUS: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, IPDPS '11, pages 676–687,

Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-0-7695-4385-7.

[14] K. Datta, S. Kamil, S. Williams, L. Oliker, J. Shalf, and K. Yelick. Optimization and performance modeling of stencil computations on modern microprocessors. *SIAM Rev.*, 51(1): 129–159, Feb. 2009. ISSN 0036-1445.

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

[16] Y. Ding, J. Ansel, K. Veeramachaneni, X. Shen, U.-M. O'Reilly, and S. Amarasinghe. Autotuning algorithmic choice for input sensitivity. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, 2015.

[17] S. Donadio, J. Brodman, T. Roeder, K. Yotov, D. Barthou, A. Cohen, M. J. Garzarán, D. Padua, and K. Pingali. A language for the compact representation of multiple program versions. In *Proceedings of the 18th International Conference on Languages and Compilers for Parallel Computing*, LCPC'05, pages 136–151, Berlin, Heidelberg, 2006. Springer-Verlag.

[18] M. Frigo and S. G. Johnson. The fastest fourier transform in the west. In *Proceedings of the 1998 International Conference on Acoustics, Speech, and Signal Processing, ICASSP '98*, 1997.

[19] A. Hartono, B. Norris, and P. Sadayappan. Annotation-based empirical performance tuning using Orio. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, IPDPS '09, pages 1–11. IEEE Computer Society, 2009.

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

[21] H. Jégou, M. Douze, and C. Schmid. Hamming embedding and weak geometric consistency for large scale image search. In A. Z. David Forsyth, Philip Torr, editor, *European Conference on Computer Vision*, volume I of *LNCS*, pages 304–317. Springer, oct 2008.

[22] 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 (IPDPS)*, 2010.

[23] Y. Liu, E. Z. Zhang, and X. Shen. A cross-input adaptive framework for GPU program optimizations. In *Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing*, IPDPS '09, pages 1–10, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-1-4244-3751-1.

[24] A. Magni, C. Dubach, and M. F. P. O'Boyle. A large-scale cross-architecture evaluation of thread-coarsening. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 11:1–11:11, New York, NY, USA, 2013. ACM.

[25] D. Merrill. Back40 Computing, . `http://code.google.com/p/back40computing/`.

[26] D. Merrill. CUDA Unbound (CUB), . `http://nvlabs.github.io/cub/`.

[27] D. Merrill, M. Garland, and A. Grimshaw. Policy-based tuning for performance portability and library co-optimization. In *Proc. Innovative Parallel Computing (InPar 2012)*, May 2012.

[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*, PPoPP '12, pages 117–128, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1160-1.

[29] S. Muralidharan, M. Shantharam, M. Hall, M. Garland, and B. Catanzaro. Nitro: A framework for adaptive code variant tuning. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, IPDPS '14, pages 501–512. IEEE Computer Society, 2014.

[30] D. Parello, O. Temam, A. Cohen, and J. Verdun. Towards a systematic, pragmatic and architecture-aware program optimization process for complex processors. In *Proceedings of the ACM/IEEE SC2004 Conference on High Performance Networking and Computing, Pittsburgh, PA, USA*, 2004.

[31] E. Photonics and NVIDIA. CULA | sparse. `http://www.culatools.com/`.

[32] 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"*, 93(2):232– 275, 2005.

[33] M. Ren, J. Y. Park, M. Houston, A. Aiken, and W. J. Dally. A Tuning Framework for Software-Managed Memory Hierarchies. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 280–291, October 2008.

[34] S. Sanfilippo and P. Noordhuis. Redis. `http://redis.io`.

[35] M. Stephenson and S. Amarasinghe. Predicting unroll factors using supervised classification. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '05, pages 123–134, Washington, DC, USA, 2005. IEEE Computer Society.

[36] 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*, pages 1–12, May 2009.

[37] R. Vuduc, J. W. Demmel, and K. A. Yelick. OSKI: A Library of Automatically Tuned Sparse Matrix Kernels. *Journal of Physics: Conference Series*, 16(1):521, 2005.

[38] R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1–2):3 – 35, 2001.

[39] 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*, pages 1–8, 2007.