ABSTRACTIONS AND STRATEGIES FOR ADAPTIVE PROGRAMMING

by

Saurav Muralidharan

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

School of Computing

The University of Utah

December 2016

The University of Utah Graduate School

STATEMENT OF DISSERTATION APPROVAL

The dissertation of **Saurav Muralidharan**

has been approved by the following supervisory committee members:

Mary W. Hall	, Chair	**05/16/2016** Date Approved
Ganesh Gopalakrishnan	, Member	**05/16/2016** Date Approved
Matt Flatt	, Member	**05/16/2016** Date Approved
Matthew Brendon Might	, Member	**05/16/2016** Date Approved
Michael Garland	, Member	**05/19/2016** Date Approved

and by **Ross T. Whitaker** , Chair/Dean of

the Department/College/School of **Computing**

and by David B. Kieda, Dean of The Graduate School.

ABSTRACT

Emerging trends such as growing architectural diversity and increased emphasis on energy and power efficiency motivate the need for code that adapts to its execution context (input dataset and target architecture). Unfortunately, writing such code remains difficult, and is typically attempted only by a small group of motivated expert programmers who are highly knowledgeable about the relationship between software and its hardware mapping. In this dissertation, we introduce novel abstractions and techniques based on automatic performance tuning that enable both experts and nonexperts (application developers) to produce adaptive code.

We present two new frameworks for adaptive programming: Nitro and Surge. Nitro enables expert programmers to specify code variants, or alternative implementations of the same computation, together with meta-information for selecting among them. It then utilizes supervised classification to select an optimal code variant at runtime based on characteristics of the execution context. Surge, on the other hand, provides a high-level nested data-parallel programming interface for application developers to specify computations. It then employs a two-level mechanism to automatically generate code variants and then tunes them using Nitro. The resulting code performs on par with or better than handcrafted reference implementations on both CPUs and GPUs.

In addition to abstractions for expressing code variants, this dissertation also presents novel strategies for adaptively tuning them. First, we introduce a technique for dynamically selecting an optimal code variant at runtime based on characteristics of the input dataset. On five high-performance GPU applications, variants tuned using this strategy achieve over 93% of the performance of variants selected through exhaustive search. Next, we present a novel approach based on multitask learning to develop a code variant selection model on a target architecture from training on different source architectures. We evaluate this approach on a set of six benchmark applications and a collection of six NVIDIA GPUs from three distinct architecture generations. Finally, we implement support for combined code variant and frequency selection based on multiple objectives, including power and energy ef-

ficiency. Using this strategy, we construct a GPU sorting implementation that provides improved energy and power efficiency with less than a proportional drop in sorting throughput.

To my parents and wife.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would first like to thank my advisor, Prof. Mary Hall, for her support and guidance throughout the course of my Ph.D. She has taught me most of what I know about research and technical writing, and I am confident that the skills I have learned from her will serve me well throughout my career.

I am thankful to my Ph.D. committee members, Prof. Ganesh Gopalakrishnan, Prof. Matthew Flatt, Prof. Matthew Might, and Dr. Michael Garland for their guidance and the many productive discussions we've had. I'd like to especially thank Michael for supervising a number of my research projects over the span of four years; I have benefited immensely from his feedback and guidance. I would also like to thank Prof. Hari Sundar for his help with the multiobjective tuning project.

My current and past labmates in the CTOP group, Anand, Protonu, Axel, Suchit, Manu, Amit, Tharindu, Khalid, Huihui, and Tuowen, have been good friends and collaborators. I would like to thank Manu and Amit in particular for their immense help with preparing my publications.

I spent two wonderful summers at NVIDIA Research, and would like to thank the various colleagues and friends I had there, including Bryan, Albert and Duane for their mentorship and help with the Surge project.

I'd like to thank the many friends I made in Salt Lake City, especially Arijit, Suchit, Anand, Nil, Nikhil, Prasanna, Sriram, Shreyas, Sharath, Manju, Meghana, Anusua, and Piyush for making my Ph.D. years fun and memorable.

I am grateful to Ann, Karen, and the School of Computing front desk staff for helping me out with various administrative tasks related to my Ph.D. and international student status; they have always been ready to help whenever I needed them.

Finally, I would like to thank my parents and wife for their unconditional support, love, and patience. They have made a number of sacrifices to help me get through my Ph.D. years smoothly, and I will be forever grateful to them.

This work was funded by Defense Advanced Research Projects Agency (DARPA) contract HR0011-13-3-0001.

CHAPTER 1

INTRODUCTION

With parallel architectures becoming increasingly complex and diverse, and energy and power efficiency also gaining importance, programmers are forced to continuously rewrite and reoptimize code as architectures and optimization objectives change. Unfortunately, this manual approach is time consuming, demands considerable knowledge of low-level architectural details, and is likely not portable. Instead, we believe that code targeting current and future parallel architectures must have the ability to intelligently and automatically adapt to changing *execution contexts* (input dataset and target architecture); additionally, it must meet multiple, possibly conflicting, higher level optimization objectives such as performance and energy/power efficiency.

A number of approaches for writing adaptive code, targeting various programmer expertise levels, have been proposed in the literature. High-level domain-specific programming systems such as Halide [1] and Elixir [2] decouple the specification of computations from their low-level implementations. This enables optimized implementations to be generated automatically, letting users focus on the computation itself. At the other end of the spectrum, there are new techniques and frameworks for automatic performance tuning (*autotuning*, for short) targeting expert programmers [3]–[6]. Such systems lift some of the burden off expert programmers, who can now focus on writing high-performance implementations, as opposed to spending effort on making the code adaptive.

In this dissertation, we present a cohesive framework for writing adaptive code that provides suitable abstractions for both experts and application developers, and incorporates a host of novel techniques for handling input adaptivity, architecture adaptivity, and multi-objectivity.

1.1 Abstractions for Adaptive Programming

Adaptive programming is the process of writing code that intelligently adapts to changing execution contexts and optimization objectives. A commonly employed mechanism in

adaptive programming 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. Given a computation and a set of code variants implementing it, one way of achieving adaptivity is to select the optimal variant for a given execution context and optimization objective. However, the question of expressing code variants still remains; in other words, what is the right level of abstraction for specifying code variants given a programmer's expertise? In this section, we study this issue in more detail and outline the contributions that this dissertation makes to address it.

1.1.1 Supporting Expert Programmers

This class of users demand very high levels of performance, and are highly knowledgeable about the relationship between software and its hardware mapping. They typically prefer to write high-performance code variants by hand, and are seeking mechanisms to make their code adaptive. To support such users, this dissertation introduces Nitro, a new programmer-directed code variant tuning system. In addition to code variants, Nitro lets programmers express metainformation for variant selection, such as how to calculate *features* or characteristics of the input datasets and target architecture, and representative training input datasets. Nitro also includes a tuning interface to optionally customize the tuning process.

1.1.2 Supporting Application Developers

In contrast to expert programmers, application developers prioritize clean, maintainable code over raw performance. Consequently, low-level abstractions for code variant expression and tuning are unlikely to be adopted by this group. This dissertation presents Surge, a nested data-parallel programming system that decouples the high-level specification of computations from their implementation details. This enables Surge to automatically generate a search space of code variants, which are subsequently tuned using Nitro.

1.2 Adaptive Code Variant Selection.

Once code variants are specified, optimal ones among them must be automatically selected depending on factors of the execution context. The fact that some of this information, such as characteristics of the input dataset, is not known until runtime makes this problem harder. Additionally, the selected variants must meet high-level, possibly conflicting, optimization objectives such as performance and energy efficiency. In this

section, we define input-adaptive, architecture-adaptive, and multiobjective tuning; we also briefly outline the contributions that this dissertation makes in these areas.

1.2.1 Input Adaptivity

Given a set of code variants implementing a computation, input-adaptive tuning finds the optimal one corresponding to a given input dataset. While some autotuning systems such as Sequoia [7] and PetaBricks [8] support input-adaptive code variant tuning, what is missing from these frameworks is more general metainformation that can be used to select variants, beyond input dataset 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 dataset may significantly impact selection of the best code variant, and is not known until runtime. As a motivating example, consider two GPU sparse matrix-vector multiplication (SpMV) variants from the CUSP library: ELL and CSR-Vector [9]. Their performance on the NVIDIA GeForce GTX 480 GPU is shown in Figure 1.1. Here, the x-axis represents number of matrix rows and the y-axis shows performance in GFLOP/s. As the figure shows, none of the variants is uniformly the best across all inputs; instead, the best variant changes with input.

One approach to input-adaptive code variant selection is to build a statistical *Model* that maps characteristics of the input dataset to the appropriate variant. In this work, we use supervised learning in an offline training phase to infer a model that maps from features of the input dataset to variants. The model is then used to select optimized code variants for new, unseen inputs. We also implement an incremental tuning mode based on active

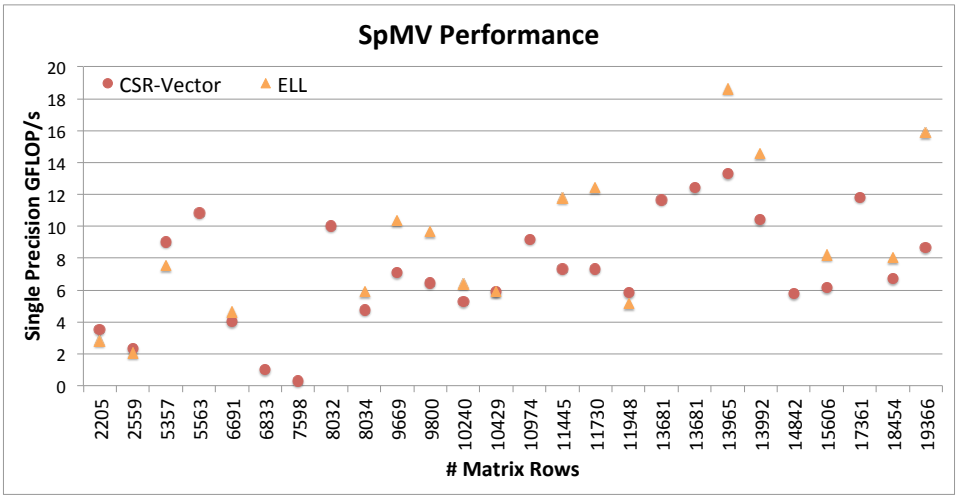


Figure 1.1: Performance of SpMV code variants on the NVIDIA GeForce GTX 480 GPU.

learning [10] for reducing the number of inputs required for training the model.

1.2.2 Architecture Adaptivity

As described above, we rely on a model-based strategy for input-adaptive code variant selection. These models, however, require retraining 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 for each input when collecting the training data. This work 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?* In other words, can we come up with an input- and architecture-adaptive code variant selection strategy?

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 six code variants for Histogram in the high-performance GPU CUB library [11]. 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 1.2 shows performance for Histogram on the GeForce 750 Ti GPU (Maxwell generation), 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 work, we instead develop a strategy to automatically construct 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 are required. Our strategy thus enables the construction of performance-portable

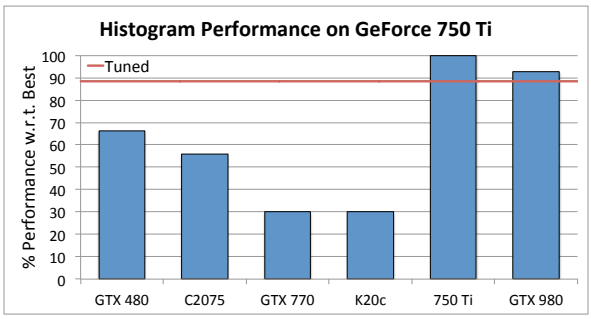


Figure 1.2: Histogram performance on the GeForce 750 Ti when trained on other architectures. The tuned line shows the performance of our strategy when trained using data from all architectures other than 750 Ti.

software that quickly and automatically adapts to both changing inputs and new hardware architectures. In Figure 1.2, the line labeled 'Tuned' shows performance achieved by our strategy trained on data from every architecture except the 750 Ti.

1.2.3 Multiobjective Tuning

Higher power consumption and associated heat dissipation in HPC systems is forcing a corresponding increase in operating costs. On the other end of the spectrum, battery life is increasingly becoming a concern on smaller embedded devices. Optimizing code variants for power and energy efficiency, while also ensuring minimal degradation in performance, is thus becoming critically important. One way of achieving this goal is through multi-objective optimization, which aims to find a set of solutions (in this case, code variants) that satisfy a set of (possibly conflicting) optimization criteria. Dynamic voltage and frequency scaling (DVFS) is another approach that has proven effective for reducing energy and power consumption, especially on GPUs [12], [13]. Given an execution context, the ability to predict the optimal frequency, in addition to code variant, would thus be useful.

In this work, we present a mechanism for users to define custom aggregated optimization metrics. This enables code variant selection based on multiple objectives, such as performance, energy consumption, etc. Further, we describe techniques for combined ⟨code variant, frequency⟩ selection, enabling the use of DVFS to further reduce energy and power consumption of code variants with minimal performance loss.

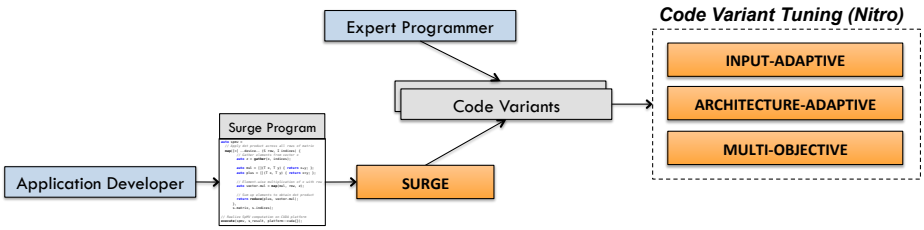


Figure 1.3: Overview of the contributions of this dissertation.

1.3 Contributions

The contributions of this dissertation are outlined below, and summarized in Figure 1.3.

1. *A framework for code variant tuning:* We describe Nitro, a programmer-directed code variant tuning framework targeted at expert users. Nitro allows code variants to be conveniently specified, together with metainformation to aid in selecting among them. As shown in Figure 1.3, it acts as a substrate for implementing all code variant tuning strategies described in this dissertation.

2. *Input-adaptive tuning:* We introduce a strategy for input-adaptive code variant selection based on supervised classification. Further, we demonstrate an incremental tuning mode based on active learning that achieves substantial reduction in the training set size. On five high-performance GPU applications, tuned variants achieve over 93% of the performance of variants selected through exhaustive search, averaged over the testing inputs.

3. *Architecture-adaptive tuning:* We present a novel approach based on multitask learning to develop a code variant selection model on a target architecture from training on different source architectures. Additionally, we introduce two techniques for pruning features that characterize each architecture and demonstrate their importance. Finally, we present performance results on a set of six benchmark applications and a collection of six NVIDIA GPUs from three distinct architecture generations.

4. *Multiobjective tuning:* We implement support for combined ⟨code variant, core clock frequency⟩ selection based on multiple objectives, including power and energy efficiency. In particular, we demonstrate how to build a sorting implementation for the NVIDIA Jetson TK1 and Tesla K80 GPUs that provides improved energy and power efficiency with less than a proportional drop in sorting throughput.

5. *A tunable programming system:* We introduce Surge, a nested data-parallel programming system that decouples the high-level specification of computations from their

low-level hardware implementations using two first-class language constructs named *schedules* and *policies*. Surge is then able to automatically generate code variants from these specifications and tune them using Nitro, as shown in Figure 1.3. For five real-world benchmarks expressed in Surge, we demonstrate performance that is on-par or better than handcrafted reference implementations on both CPUs and GPUs.

1.4 Dissertation Roadmap

The remainder of this dissertation is organized as follows: we first introduce Nitro and describe techniques for code variant tuning with respect to input adaptivity in Chapter 2. In Chapters 3 and 4, we describe strategies for architecture-adaptive and multiobjective tuning, respectively. Next, in Chapter 5, we describe Surge, including its programming interface, and code generation and tuning infrastructure. Finally, Chapter 6 discusses relevant prior research on autotuning techniques and high-level parallel programming systems, and Chapter 7 concludes.

CHAPTER 2

INPUT-ADAPTIVE TUNING

One approach to input-adaptive code variant selection is to build a statistical *model* that maps characteristics of the input dataset to the appropriate variant. Such a model can be queried to perform variant selection at runtime once properties of the input dataset are available. In previous work on the *algorithm selection problem* [14], statistical learning techniques are used to select among a set of different algorithms [15]–[17]. To date, however, no general-purpose framework enables users to specify and tune arbitrary code variants and also customize the tuning process.

This chapter describes a new programmer-directed autotuning system called Nitro. It focuses on (1) how code variants and metainformation for variant selection are expressed in Nitro, and (2) underlying system support that selects the most appropriate variant for a given input dataset. 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 an 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.

2.1 Automating Code Variant Selection

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 [18]–[22]. Unfortunately, the structure of the matrix is usually not known until runtime, 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 [22] 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 dataset at runtime 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 runtime, 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 metainformation 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.

2.1.1 Nitro System Overview

Figure 2.1 provides a high-level overview of the Nitro system, which consists of two parts: the Nitro Library, implemented using C++ Templates (Figure 2.1a), and the Nitro Autotuner, written in Python (Figure 2.1b). 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 2.1b) 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)* [23], a widely used machine learning algorithm to build the model from an offline training phase on the *Training Input* so that it can be consulted at runtime given the feature vector of a new input. We use the publicly available `libSVM` [24] for this purpose.

2.1.2 Nitro Library Constructs

Table 2.1 provides a summary of the constructs available in Nitro for expressing variants and their associated features and constraints. Figure 2.2 provides Nitro code for SpMV to illustrate these constructs, as described in the following paragraphs.

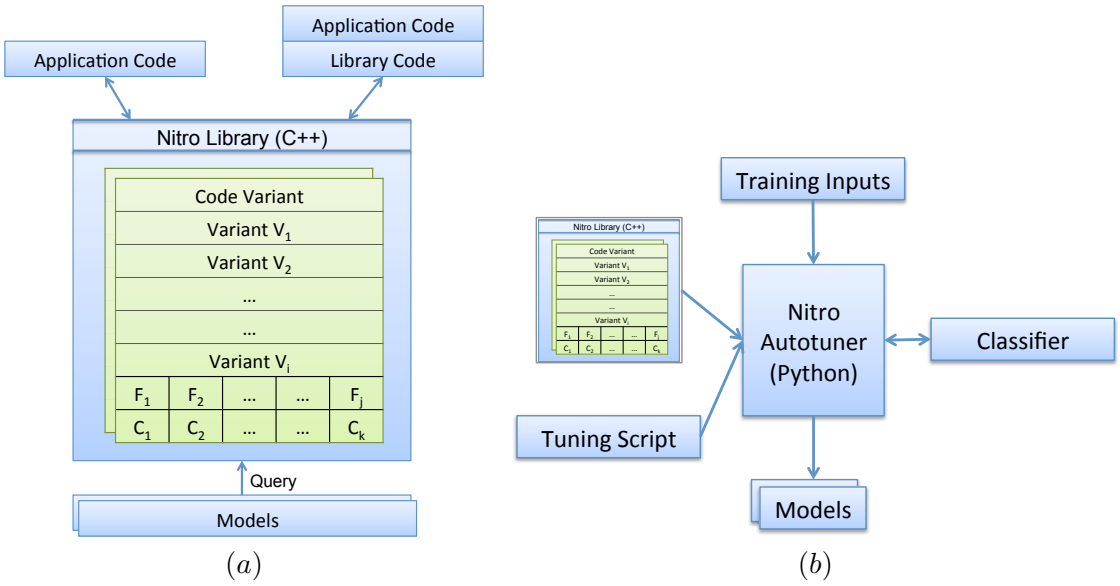


Figure 2.1: Overview of the Nitro system. (a) The production version of the library/application. The C++ library is used to define variants, features, and constraints. Calling the variant evaluates the input features at runtime and queries the accompanying model to select the right variant to execute 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 2.1: 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.

```
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 2.2: Example Nitro Library interface for SpMV.

2.1.2.1 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 2.2, 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 2.2. Variants are added to the `code_variant` object using the `add_variant` function, which accepts a pointer to the 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 2.2, 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* [22].

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.

2.1.2.2 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 2.2, 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 2.2.3 is asynchronous feature evaluation; asynchronous feature evaluation is enabled by calling the `fix_inputs` function before calling `operator()`.

2.1.2.3 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 2.2, the constraint `__dia_cutoff` ensures that the `__dia` variant does not get executed if the constraint evaluates to false.

2.1.3 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 2.2 shows the various configuration options available. Most of these options have a default value, and the only essential information that must be provided is the training input dataset and the functions to be tuned. The remaining functionality allows the expert user to optionally control the tuning process as desired.

Figure 2.3 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

Table 2.2: 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 2.3: Example Nitro Autotuner interface for SpMV.

file also enables us to use the C++ template mechanism to selectively generate relevant code.

2.2 The Nitro Autotuner

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.

2.2.1 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) [25] 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.

2.2.2 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 [26], 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 [10]. At each iteration, Nitro updates the model.

When using incremental tuning, Nitro requires a stopping criterium to determine when the number of training inputs is sufficient to construct an accurate model. As shown in Table 2.2, 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 2.4.2). Alternatively the accuracy threshold with respect to the test input is useful if 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 work, we were able to achieve considerable reductions in training times using this strategy for incremental tuning (see Section 2.4.2).

2.2.3 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 [27] library.

2.3 Benchmarks

Table 2.3 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 datasets. All of these benchmarks are derived from high-performance CUDA libraries that already included code variants. Further, for each benchmark, the best performing code

Table 2.3: A brief description of variants and list of features used for each benchmark. The last column lists the sizes of training and testing sets.

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.		

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.

2.3.1 Sparse Matrix-Vector Multiplication (SpMV)

As described in Section 2.1, 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 [9] to provide the code variants for SpMV. We use 3 features related to the matrix row lengths (average nonzeros 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 [28] 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.

2.3.2 Linear Solvers and Preconditioners

Many large-scale scientific simulations such as computational fluid dynamics (CFD) and structural mechanics [29] involve solving partial differential equations (PDE) systems. Typically, solution to a PDE-based system involves solving the underlying sparse linear system using software toolkits [30], [31]. One of the challenges in effectively using such toolkits is the selection of an appropriate ⟨ linear solver, preconditioner ⟩ 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 [31], 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. [32]. 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.

2.3.3 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 [33], part of a larger set of GPU primitives provided in the Back40 Library [34]. 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 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 traversed edges per second (TEPS) as the optimization metric.

2.3.4 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) [11] 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 subsequence of the input sequence (SubSampleSD in Table 2.3). 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 [35] (converted to grayscale). Out of the 1491 images in the dataset, 200 are used for training and the rest for testing.

2.3.5 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 [36] library of GPU primitives, while the Radix Sort implementation is provided in CUB [11]. 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 subsequences 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.

2.4 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 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.

2.4.1 Variant Selection

Figure 2.4 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.

2.4.1.1 Sparse Matrix-Vector Multiplication

The first bar in Figure 2.5 shows the tuning results for SpMV. On average, SpMV selected through Nitro achieved a performance of 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

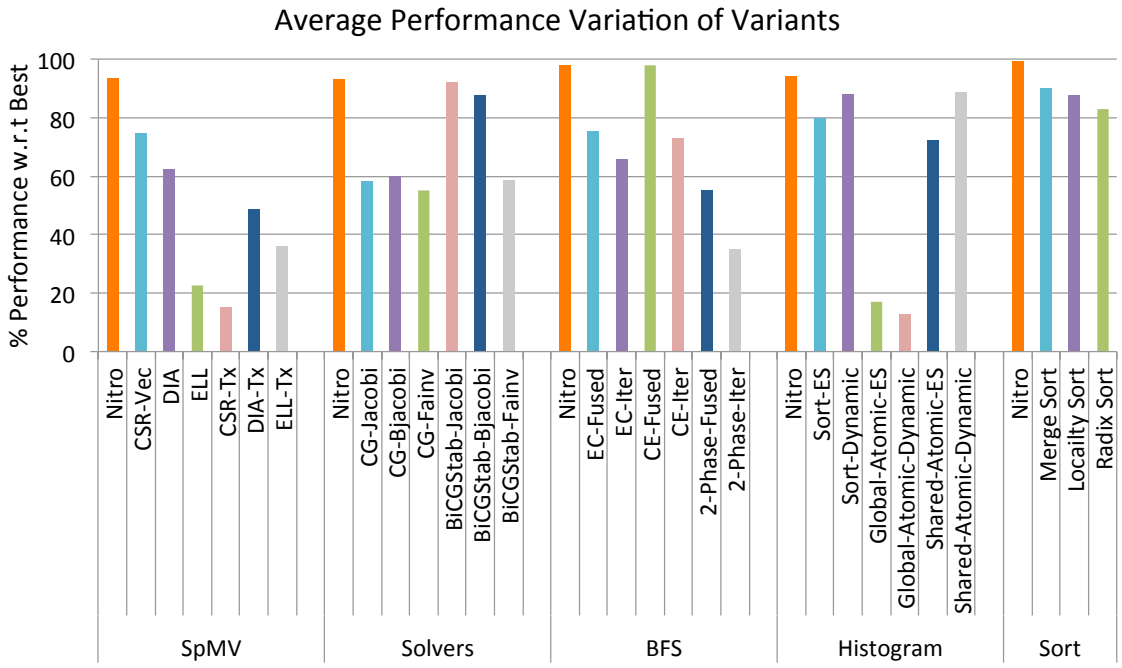


Figure 2.4: Performance variation among variants.

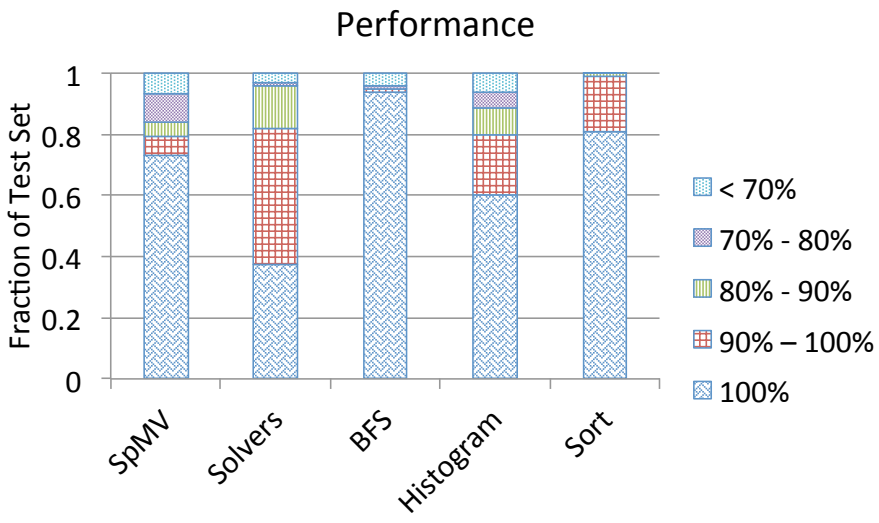


Figure 2.5: Performance comparison across all test inputs.

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.

2.4.1.2 Linear Solvers and Preconditioners

For the second benchmark in Figure 2.5, 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.

2.4.1.3 Breadth-First Search

For the third benchmark in Figure 2.5, 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. [33]. 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.

The Hybrid variant proposed in Merrill et al. [33] 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.

2.4.1.4 Histogram

For the fourth benchmark in Figure 2.5 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 are uniformly distributed. For nonuniformly 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.

2.4.1.5 Sort

The last bar in Figure 2.5 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 2.4, it is also clear that on average, the Nitro-selected variant performs better than all the other variants, irrespective of data type.

2.4.2 Training Time Reduction

As described in Section 2.2, 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 2.6 shows that the number of iterations required by incremental tuning to reach within 90% of the performance achieved without incremental tuning is roughly 25. 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

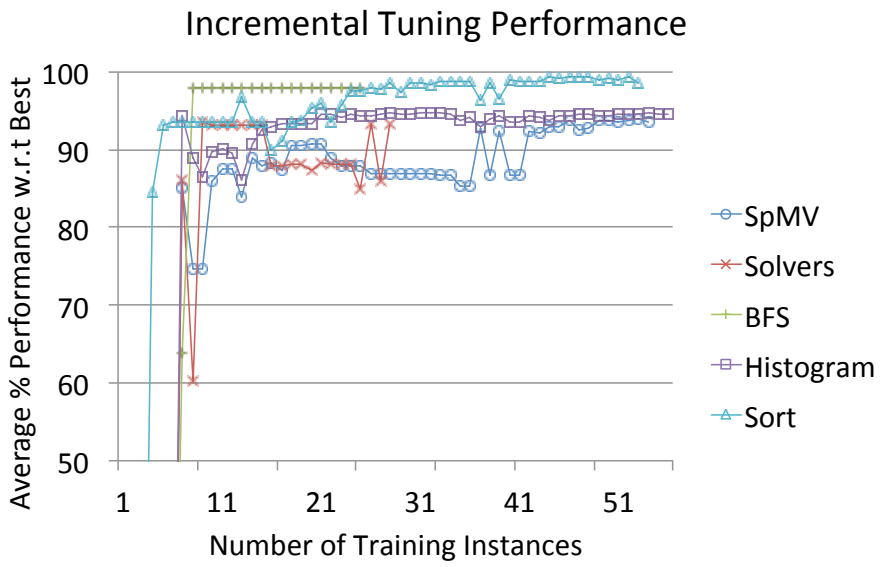


Figure 2.6: Convergence for active learning training heuristic.

benchmark. Another observation from the figure is that sometimes additional training data lead 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.

2.4.3 Feature Evaluation Overhead

Figure 2.7 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 subsample 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 subsample, at the cost of slightly decreased overall performance.

In the remaining benchmarks (SpMV and Solvers), it is evident that getting peak

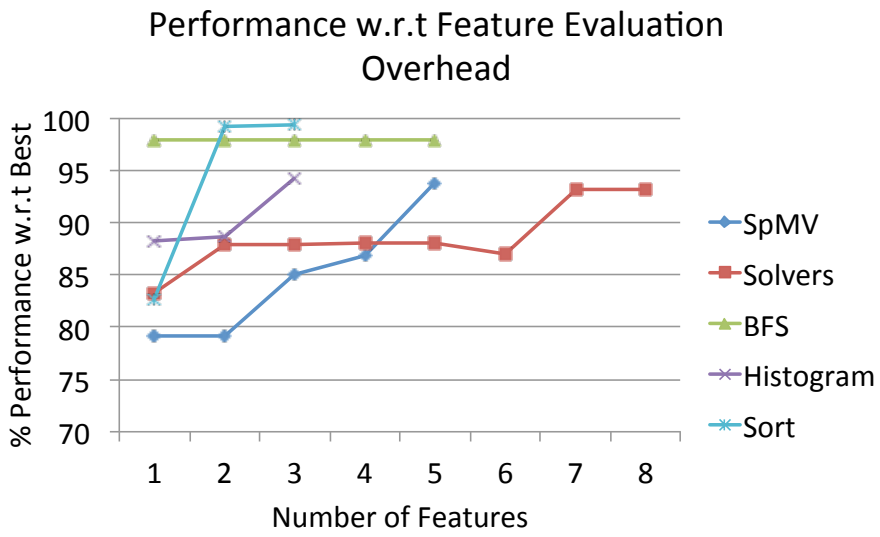


Figure 2.7: Performance variation as features with higher evaluation overhead are added incrementally.

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.

2.5 Summary

This chapter has presented Nitro, a programmer-directed autotuning framework that permits the expression of code variants, together with metainformation for selecting among them. Using the Nitro platform, we implemented a strategy that employs supervised learning to select code variants based on features of their input dataset. The support in Nitro for deriving classification models of input datasets 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 our strategy achieve over 93% of the performance of variants selected through exhaustive search, averaged over the testing inputs. Further, we demonstrate an incremental tuning mode based on active learning that achieves substantial reduction in the training set size.

CHAPTER 3

ARCHITECTURE-ADAPTIVE TUNING

Input-adaptive code variant selection schemes such as the one described in Chapter 2 rely on an offline training phase for model construction. However, such model(s) must be retrained 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 work evaluates whether we can 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 work is general, we simplify the problem by focusing on only NVIDIA GPUs.

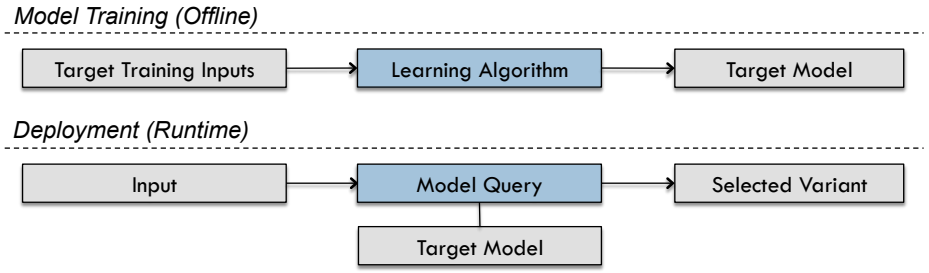
In this chapter, we develop a strategy to automatically construct code variant selection model(s) on a target architecture using only training data from a set of source architectures specified by the programmer, together with information that characterizes each architecture. We treat the cross-architectural tuning problem as a *multitask learning* [37] 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 multitask learner to come up with variant selection model(s) for the target architecture.

3.1 System Overview

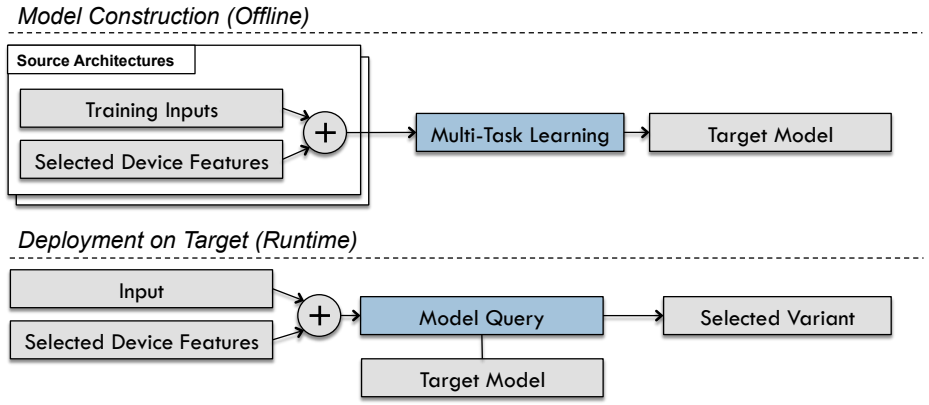
The automated system presented in this chapter extends the Nitro autotuning framework described in Chapter 2. Figure 3.1(a) illustrates the approach for input-adaptive tuning in Nitro. For each architecture, training data have the form $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_N, y_N)\}$, where each $\mathbf{x}_i$ 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 3.1(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 3.1. 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, we added to



(a) Overview of input-adaptive tuning in Nitro.



(b) Architecture-adaptive tuning overview.

Figure 3.1: Comparison of input-adaptive tuning in Nitro 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).

Table 3.1: Values of GPU device features for 6 architectures.

	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

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 work are also listed in Table 3.1.

We pose the problem of architecture-adaptive tuning as a *multitask 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 intertask 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 3.2.1. In earlier stages of this research, we implemented and explored other MTL algorithms such as weighted kernels and probabilistic SVMs [38], 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.

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 3.2 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 1.2), while the same pair of architectures is relatively similar for the SpMV benchmark.

3.2 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 multitask 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 multitask 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.

Table 3.2: Cosine similarity between architectures for Histogram (H) and SpMV (S). Values closer to +1 indicate similarity, while values closer to -1 indicate dissimilarity.

	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

3.2.1 Model Construction Using MTL

The feature concatenation strategy for multitask 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.

3.2.2 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 multitask learner when it builds the variant selection model for the target. In this subsection, we describe how this naïve strategy works.

3.2.2.1 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. 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.

3.2.2.2 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 3.2.1.

3.2.3 Profile Device Feature Selection (P-DFS)

With a restricted set of source architectures, extraneous device features can confuse the multitask learner, as demonstrated in Section 3.5.2. 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.

3.2.3.1 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 3.3 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 3.3 list these associations for each proxy.

3.2.3.2 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 3.3 (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.

Table 3.3: GPU application proxies with corresponding profiling metrics and device features.

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

3.2.3.3 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 3.2.2, the user is required to initiate profiling data collection. This automatically collects all the required profiling data and stores it in the repository.

3.2.3.4 Target Architecture Side

On the target, the construction of variant selection models proceeds as in Section 3.2.2. However, 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 $\langle C_{\phi j}, \phi \rangle$ are first retrieved from the repository. These are 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 3.3) are returned as output of the P-DFS system.

3.2.4 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 extraneous features that confuse the variant selection model on the target. To obtain an even more pruned and relevant set of device features, we

Algorithm 1 Profile Device Feature Selection

```
▷ V: Set of variants
▷ I: Set of training inputs
▷ P: Set of application proxies
global_best ← {}
for v ∈ V do
    ▷ For each kernel in variant v
    for k ∈ kernels[v] do
        ▷ For each training input
        for i ∈ I do
            intensity ← {}
            for p ∈ P do
                ▷ Profiling data for kernel k on input i
                t ← profile[k, i]
                ▷ Predict intensity for proxy p on profile
                intensity[p] = intensity-predict(t, p)
            best_proxies[i] = {x : intensity[x]is highest}
        ▷ Add best proxies across inputs to global best
        global_best ∪ = majority-vote(best_proxies)
return global_best
```

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 perfor-

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 ∈ 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 ∈ sources}
12:    T_t ← training-data(target)
13:    for d ∈ subsets[D] do
14:        ▷ Get device feature values of source and target
15:        DF_s ← {df-values(d, x): x ∈ 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)
```

mance on source architectures are likely to be good predictors on the target for the same computation. As demonstrated in Section 3.5, this assumption holds for most applications.

3.3 Implementation

As described in Chapter 2, the Nitro framework 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 chapter, 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` [39]. 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 3.1 and 3.2 provide examples of how this interface is used to tune a Histogram computation with 6 code variants on the source and target sides, respectively. The `record` flag (line 6 in Listing 3.1), when set, instructs the system to record training data and device feature values in the store during the tuning process. In Listing 3.2, the call to `tune_from_source` automatically retrieves these data for all the source architectures and builds a model for the target architecture.

```
1  from nitro import *
2  import glob
3
4  histogram = code_variant("histogram", 6)
5  # Record training data in store
6  histogram.record = True
7  histogram.device_id = "gtx_480"
8  # Create autotuner instance
9  tuner = autotuner("histogram")
10 inputs = glob.glob("training/*.jpg")
11 tuner.set_training_args(inputs)
12 # Tune for current architecture
13 tuner.tune([histogram])
```

Listing 3.1: Histogram tuning example - source architecture.

```
1  from nitro import *
2
3  histogram = code_variant("histogram", 6)
4  histogram.profiling_based_dfs = True
5  histogram.search_based_dfs = True
6  # Create autotuner instance
7  tuner = autotuner("histogram")
8  # Build model from source data
9  tuner.tune_from_source([histogram])
```

Listing 3.2: Histogram tuning example - target architecture.

3.4 Benchmarks

Table 3.4 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.

3.4.1 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 [11]. 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 [35] (converted to grayscale). Out of the 1491 images in the dataset, 200 are used for training and the rest for testing.

3.4.2 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 [9]. We use 5 features and a training set consisting of 54 matrices from the UFL Sparse Matrix collection [28]. 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.

Table 3.4: Variants and features used for each benchmark. The last column lists the sizes of training and testing sets.

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`	Features related to row length.	(54, 100)
	DIA, ELL	Perform SpMV on DIA and ELL formatted matrices.	`DIA-Fillin, ELL-Fillin`	Fillin ratio for DIA and ELL formats.	
Solver	CG-Jacobi, CG-Bjacobi, CG-Fainv	Conjugate gradients method with Jacobi, Blocked Jacobi and Factorized Approximate Inverse preconditioners	`NNZ, #Rows`	Number of nonzeros and rows.	(26, 100)
	BiCGStab-Jacobi BiCGStab-BJacobi BiCGStab-Fainv	BiConjugate gradients Stabilized method with Jacobi, Blocked Jacobi and Factorized Approximate Inverse preconditioners	`Trace`	Trace of a matrix.	
			`DiagAvg, DiagVar, DiagDominance`	Features related to diagonal elements of a matrix.	
			`LBw`	Left bandwidth of a matrix.	
			`Norm1`	1-norm of a matrix.	
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`	Features related to graph out-degree.	(20, 138)
	CE-Fused, CE-Iterative	Contract incoming edge frontier, filter, and produce outgoing edge frontier.	`#Vertices, #Edges`	Number of vertices and edges.	
	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`	Sequence of length and average length.	(200, 1291)
	Global-Atomic-ES Global-Atomic-Dynamic	Compute Histogram using global atomic add operations.	`SubSampleSD`	Standard deviation of sub-sample.	
	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`	Number of rows, columns.	(194, 1000)
	Skinny R2C Skinny C2R	R2C, C2R specialization for skinny matrices	`RowMajor`	Is in row-major layout.	
			`CoPrime`	Are M and N co-prime.	

3.4.3 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 [36] library of GPU primitives, while the Radix Sort implementation is provided in CUB [11].

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.

3.4.4 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 [33], part of a larger set of GPU primitives provided in the Back40 Library [34]. 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 five 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.

3.4.5 Linear Solvers and Preconditioners

Many large-scale scientific simulations such as computational fluid dynamics (CFD) and structural mechanics [29] involve solving partial differential equations (PDE) systems. Typically, solutions to a PDE involve solving the underlying sparse linear system using software toolkits [30], [31]. One of the challenges in effectively using such toolkits is the selection of an appropriate ⟨linear solver, preconditioner⟩ combination, as this selection impacts both the performance and convergence of the computation. For this experiment, we use six ⟨linear solver, preconditioner⟩ 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. [32]. We use symmetric sparse matrices from [28] to represent sparse linear systems.

3.4.6 Matrix Transposition

In-place transposition of square matrices is a well-studied problem. Transposition of a nonsquare matrix is a much more involved process, requiring $O(mn \log mn)$ work. Catanzaro et al. [40] 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 [41]. 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.

3.5 Results

We run our experiments on six NVIDIA GPUs characterized by the device features in Table 3.1: (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 3.1, these graphics cards span three GPU architecture families: Fermi, Kepler, and Maxwell. We use CUDA Toolkit version 6.5 for our experiments (except for Solvers, which requires CUDA 6.0 due to CULA). 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.

3.5.1 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 3.2 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 3.3 is similar, except it depicts architectural sensitivity of benchmarks within GPUs of each generation - namely, Fermi, Kepler, and Maxwell.

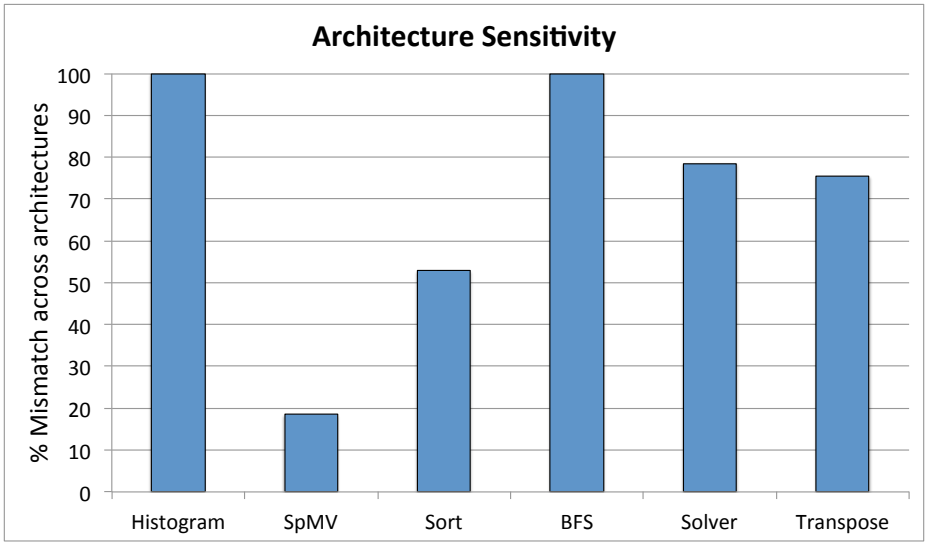


Figure 3.2: Architecture-sensitivity of each benchmark. The y-axis represents the percentage of test inputs for which at least one architecture selects a different best variant than the others.

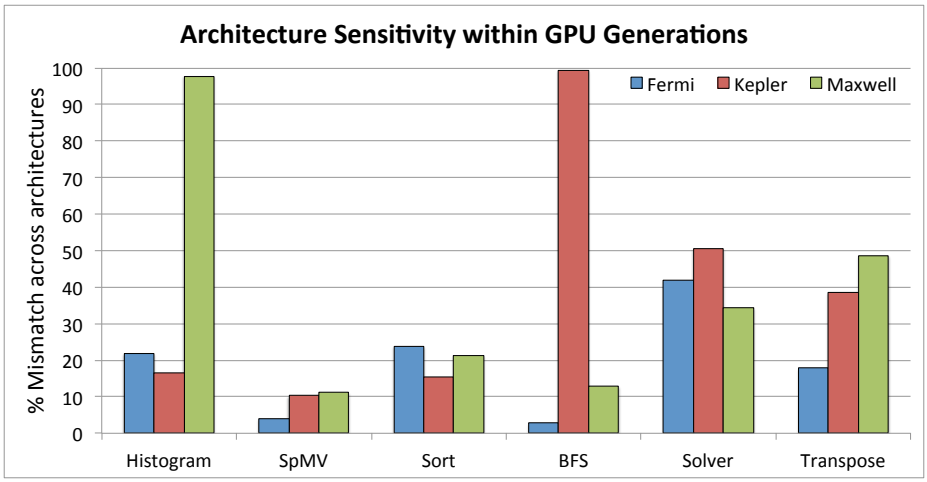


Figure 3.3: Architecture-sensitivity within GPUs of the same generation.

Figure 3.3 shows that differences in variant selection are usually less pronounced within the same architectural family, but not always. Further data are shown in Table 3.5, where each subtable 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.

From one generation to the next, new architecture features and machine configurations may dramatically affect variant selection (e.g., support for atomic operations). But within a single generation, different selections are usually attributed to differences in (1) raw

Table 3.5: Variant selection histograms across different benchmarks and architectures. Each subtable represents the distribution of variant selections across test data for a particular benchmark.

	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

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 3.1. From Figures 3.2 and 3.3 and Table 3.5, 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 3.1 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 Table 3.4, 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 3.6, second column) and the GTX 770 has the lowest L2 cache size of all GPUs (Table 3.1). 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 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 3.5 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

Table 3.6: Best device features for each benchmark, proxies predicted by P-DFS, and the best features chosen by CV-DFS.

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`

CULA, which are closed-source.

3.5.2 Prediction Performance

First we look at how well device feature selection detects the relevant features for each benchmark. Table 3.6 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 multitask learning compare in their effectiveness against the original Nitro system (training and testing performed on the target architecture) and exhaustive search. In Figure 3.4, 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 multitask learning uses all device features, while

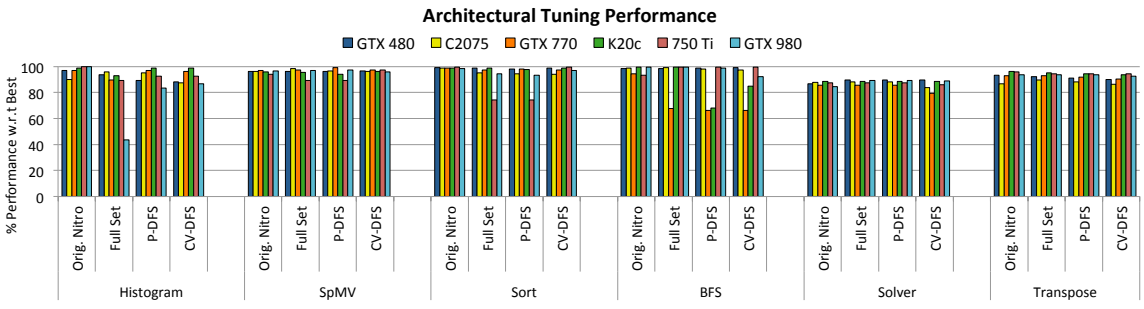


Figure 3.4: Device feature selection performance.

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 3.5.1, but effectively they indicate instances where the learning phase did not see similar scenarios. Surprisingly, multitask 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 target. It is an indication that multitask 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 3.5 shows the results for this experiment. Here, each subfigure 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 3.5, respectively).

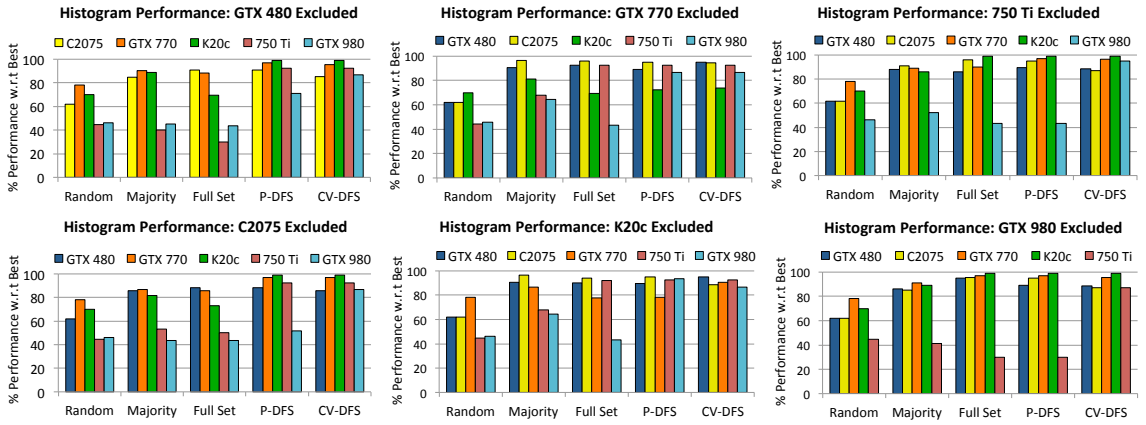


Figure 3.5: Device feature selection performance for Histogram on a restricted set of architectures.

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 *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 3.5) 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.

3.5.3 Device Feature Selection Overhead

Table 3.7 shows the overhead incurred by P-DFS and CV-DFS (since the repository is stored on the local network, we do not include communication overheads). 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.

3.5.4 Results Summary

Overall, multitask 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 fail 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 fewer training data are available, and that in such cases, CV-DFS produces superior results but introduces more overhead than other approaches.

3.6 Summary

This chapter has presented a novel approach to cross-architecture autotuning, which uses multitask learning to develop a model on a target architecture from training on different

Table 3.7: Device feature selection overhead (time in seconds).

	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

source architectures. We demonstrated that device feature selection is valuable in building a successful code variant selection model on new architectures, and discussed the strengths and limitations of the approach. Finally, 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 multitask 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.

CHAPTER 4

TUNING FOR ENERGY AND POWER EFFICIENCY

Multiobjective optimization refers to the problem of optimizing a vector $\vec{f} = \langle f_1, ..., f_r \rangle$ of optimization functions of r criteria. While a number of techniques and algorithms for multiobjective optimization have been proposed in the literature, there have been relatively few autotuning frameworks that take multiobjectivity into account [42]–[44]; among these, none support tuning with respect to the input dataset and target architecture.

In this chapter, we describe two strategies for supporting multiobjective code variant tuning in Nitro. First, we describe an interface for defining *custom aggregation metrics*. These enable users to consolidate various optimization criteria such as performance and energy/power efficiency into a single metric, which Nitro then uses to automatically select among code variants during the training phase. Second, we present a strategy for selection of core clock frequencies in addition to code variants.

To evaluate the effectiveness of our approach, we use GPU sorting as a case study; specifically, we tune two high-performance GPU sorting implementations and up to 25 different core clock frequencies on two distinct GPU architectures. The resulting tuned implementation demonstrates improved energy and power efficiency with less than proportional loss in sorting throughput.

4.1 Multiobjective Tuning in Nitro

An important characteristic of multiobjective optimization problems is that there may not exist a single alternative which is optimal for all the criteria. Solving a multiobjective optimization problem involves computing a set of alternatives, each representing a trade-off among the optimization criteria. The alternatives in this set fulfill two conditions: (1) they cannot be further improved without worsening at least one of the optimization functions, and (2) none of them is better than the others for all the objective functions. The alternatives in this set are referred to as *nondominated solutions*, and the set itself is defined as the *Pareto*

set. The Pareto set, when only the values of the optimization functions are considered, is referred to as the *Pareto frontier*.

Existing multiobjective tuning systems typically convert the multiobjective problem into a single-objective one using one of two methods: (1) optimize for one particular objective, but only return solutions that ensure that certain constraints are not violated for the other objectives, and (2) compute an aggregation of optimization functions (using weights, for example) and optimize the resulting single-objective function. Some other systems try to directly approximate the Pareto frontier, and rely on the user to select the final solution from this space [42]. Since Nitro must compute an optimal solution for each training input, we avoid techniques such as the latter that rely on continuous user involvement.

4.1.1 Extensions to Autotuning Interface

We add support for multiobjectivity in Nitro through the metric aggregation technique described above. The new interface requires each code variant to report the correct measurements, which may include performance, power readings, etc., to Nitro (as opposed to only performance). Next, the programmer provides a function g, which, given the values of the various optimization functions, computes the aggregated output. Nitro is then able to automatically use g to consolidate the various measurements reported by code variants into a single metric and find the best variant corresponding to each training input. Listing 4.1 shows an example aggregation function named `tput_over_e` for sorting. Given a list of sorting variants with their corresponding throughput (T) and energy consumption (E) values, this function returns the variant with the lowest value of the aggregated metric $\frac{T}{E}$.

4.1.2 Combining Code Variant and Frequency Selection

Dynamic voltage and frequency scaling (DVFS) has been successfully employed to improve the energy and power efficiency of GPU applications [12], [13]. In this work, we introduce a small set of extensions to Nitro that enables it to predict the best frequency corresponding to a given input, in addition to the best variant. Programmers first specify valid frequencies using the `tune_frequency` function in the autotuning interface. Nitro then automatically inserts code to set up the correct frequency before a variant is executed, and collects training data in the form $T = \{(\mathbf{x}_1, \langle v_1, f_1 \rangle), \ldots, (\mathbf{x}_M, \langle v_M, f_M \rangle)\}$, where each $\mathbf{x}_i$ represents an input feature vector and each $\langle v_i, f_i \rangle$ pair represents the best variant and frequency for that input. T is then split into $T_v = \{(\mathbf{x}_1, v_1), \ldots, (\mathbf{x}_M, v_M)\}$, and $T_f = \{(\mathbf{x}_1, f_1), \ldots, (\mathbf{x}_M, f_M)\}$, which are used to construct variant and frequency selection models, respectively. At runtime, both the models are queried to obtain the best ⟨variant,

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

gpu_sort = code_variant("gpu_sort", 3)
# Set up aggregation function
gpu_sort.set_aggregation_function(tput_over_e)
...

# The aggregation function takes a list of (i, x1, x2, ...)
# tuples, where i is the variant index and x1, x2, ... are
# the value(s) to be optimized. The best tuple is returned.
def tput_over_e(ls):
  # Find the variant with lowest value of
  # throughput/energy. Here, x[1] is the throughput,
  # and x[2] is energy consumption.
  return min(ls, key = lambda x: float(x[1])/float(x[2]))
```

Listing 4.1: Sample aggregation function for optimizing sorting throughput and energy consumption.

frequency⟩ pair for the given input.

4.2 Energy and Power-Efficient GPU Sorting

We evaluate the effectiveness of our system using GPU sorting as a case study. The following algorithms are used as code variants:

- *Merge Sort:* Recursively split a list in half, sorting each half, and then merge the two sorted lists together. The implementation we use is part of the ModernGPU [36] library of GPU primitives.
- *Radix Sort:* Successively group keys by individual digits that have the same position and value. The implementation is from the CUB Library [11].

Additionally, we vary core clock frequencies on the GPUs we consider, as described in Sections 4.1.2 and 4.3. Lastly, we use the same input features as in the Sort benchmark in Chapter 2 (see Table 2.3).

4.2.1 Aggregated Metrics for Sorting

As described in Section 4.1, performance, power and energy measurements are combined into aggregated metrics for code variant and frequency selection. For sorting, we have chosen to use a set of metrics based on sorting throughput T (measured in keys per second), energy consumption E (in Joules), and maximum power draw P (in Watts). We explore which leads to the best reduction in energy or power with the least impact on performance. These metrics are as follows: $\frac{T}{E}$, $\frac{T}{E^2}$, $\frac{T^2}{E}$, $\frac{T}{P}$, $\frac{T}{P^2}$, and $\frac{T^2}{P}$. The *best* ⟨variant, frequency⟩ pair

is thus the one with the highest value for the given metric. These metrics were selected because they capture the relationship between throughput and energy or power. Further, it is straightforward to build a model for code variant selection by consolidating on a single metric. They also reflect a different assignment of priorities on the various optimization criteria. For example, $\frac{T^2}{E}$ prioritizes the maximization of throughput over reduction in energy consumption; in contrast, $\frac{T}{E}$ gives equal priority to both. In our experiments, we will show how the choice of metric affects the balance between performance and power/energy efficiency.

4.3 Experimental Methodology

As described in Section 4.2, we select among two GPU sorting variants: merge sort (from the ModernGPU library) and radix sort (from the CUB library). We now describe the target architectures, core clock frequencies, and input datasets used in our experiments.

4.3.1 Target Architectures

We run our experiments on two different NVIDIA GPUs: the Tesla K80 and the Jetson TK1. While the K80 is among the highest performing GPUs available today, the TK1 is representative of lightweight, low-power embedded GPUs.

4.3.1.1 NVIDIA Tesla K80

The NVIDIA Tesla K80 GPU has 26 GPU streaming multiprocessors, for a total of 4992 cores, and 24 GB of memory. It supports 25 core clock frequency settings ranging from 562 Mhz to 875 Mhz in 13 Mhz increments. Two memory frequencies are also supported, but we do not adjust memory frequency in these experiments because the lower memory frequency of 324 Mhz is far lower than the peak of 2505 Mhz and is therefore going to perform poorly in a bandwidth-limited algorithm such as sort. For frequency adjustments and energy/power measurements on the K80, we created two new libraries: `gpu_freqlib` [45], and `gpu_powermon`, respectively, based on the NVIDIA Management Library (NVML) interface. Power readings were sampled at 10 Hz. The K80 also uses a mechanism called *AutoBoost* to dynamically vary clock frequencies to fill available power headroom; we disable this feature for more consistent data collection.

4.3.1.2 NVIDIA Jetson TK1

We also measured sorting performance on an NVIDIA Jetson TK1; the Jetson is low power and lightweight, consisting of a single GPU streaming multiprocessor (Kepler gen-

eration) with 192 cores, and four-plus-one ARM cores, where the fifth ARM core is used as a master processor. It has a unified DRAM of 2 GB, which is shared between CPUs and GPUs, and separate cache structures for CPU and GPU. The software installation uses CUDA 6.5, with version 6.5.35 of the `nvcc` compiler. The power and energy reported for Jetson are physical measurements using the Yokogawa WT310 digital multimeter. We measure the voltage drop across a known precision resistance in series with the device under test (DUT). With a known resistance and measured voltage on that resistance, the current can be obtained with the equation $I = V/R$. Here, the resistance is 0.020 Ohms, with a 1% variation. To determine the power, we use the equation $P = IV$, where I is the value calculated above, and V is 12.19V. The Jetson has 14 core clock frequencies ranging from 72-852MHz.

4.3.2 Input Data

As the performance of sort is dependent on its input data, we use a variety of data types, distributions, and sizes in our experiments. We consider two data types: integer (32-bit) and long integer (64-bit). Input sequence sizes range from 5K up to 25M keys. Finally, we use three different input distributions in our experiments: `uniform, Gaussian, and zipf`. In total, we generate 54 and 66 sequences for the training and test datasets, respectively, and ensure that there are no overlapping points between the two sets.

4.4 Experimental Results

We first analyze the effects of frequency adjustments on variant performance and energy/power efficiency. Figure 4.1 shows the results of this experiment for a fixed test input sequence of 10 million 64-bit (long) integers with `uniform` distribution. Here, the x-axis represents supported core frequencies on the Jetson TK1 and the y-axis shows throughput (left subfigure), energy-efficiency (middle subfigure), or maximum power draw (right subfigure). We notice that throughput and maximum power draw increase for both variants as frequencies go up. Energy efficiency tends to go up until 396 MHz and then slightly decreases at higher frequencies. From our experiments, we noticed that the shape of variant curves changed substantially with input characteristics for both the training and test sets. Further, for a given metric, the best variant for an input is often different for different frequencies, as seen in Figure 4.1.

Next we look at how frequency selection is affected by the choice of optimization objective. We evaluate this by exhaustively searching for the best frequency corresponding to each input in our test dataset. Figure 4.2 shows the distribution of frequency selections for

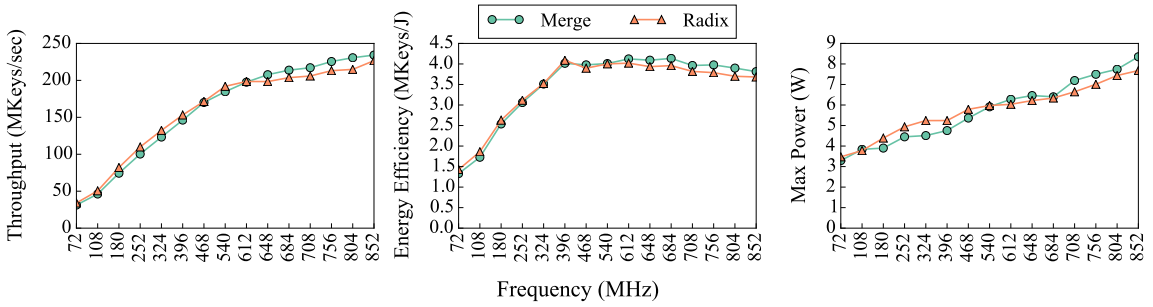


Figure 4.1: Variation in throughput (keys sorted per second), energy efficiency (keys sorted per Joule), and maximum power draw of code variants as frequency increases. Results are for an input sequence of 10M elements, `long` datatype and `uniform` distribution on the Jetson TK1.

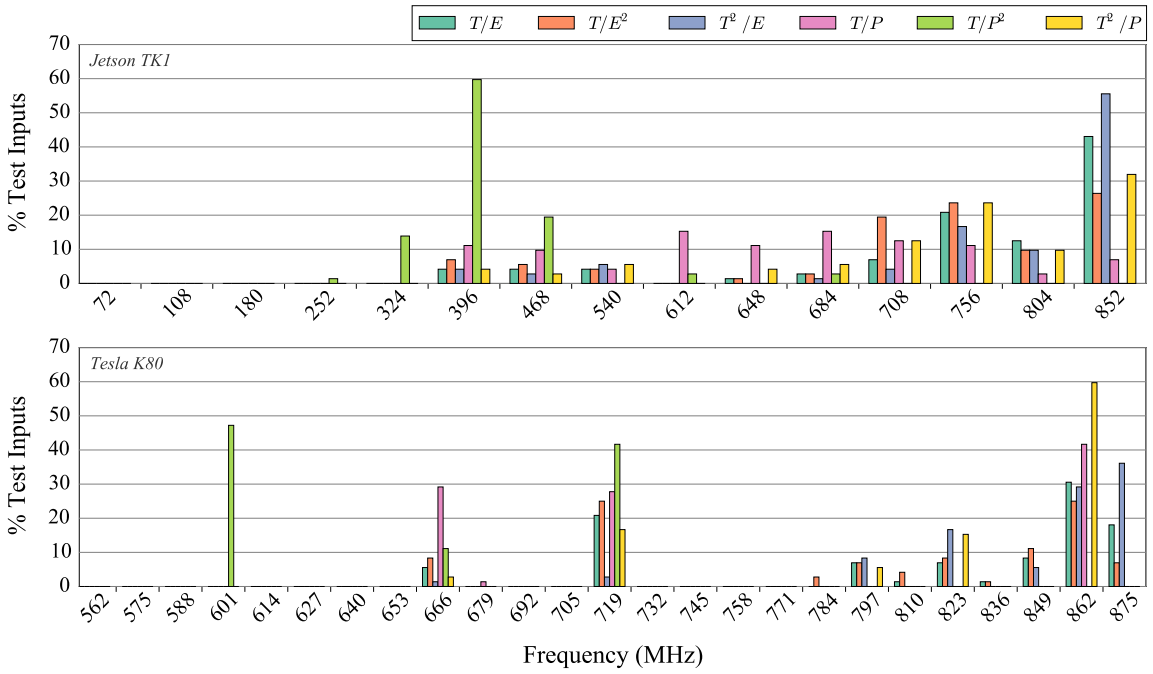


Figure 4.2: Distribution of frequencies selected via exhaustive search on the Jetson TK1 (top) and Tesla K80 (bottom) for various optimization objectives.

the six optimization objectives described in Section 4.2 on the Jetson TK1 (top) and Tesla K80 (bottom). Here, the x-axis represents frequency and the y-axis is the percentage of test inputs for which a given frequency is selected. Each bar represents a different optimization objective. We notice that frequency selections are relatively more spread out on the Jetson TK1. It is also interesting to note that 13 of the 25 frequencies are never selected by any optimization objective on the Tesla K80. On both architectures, the highest frequency is preferred for a number of inputs when throughput optimization is given importance (T^2 in

the numerator), as expected. Further, we notice that $\frac{T}{P^2}$ prefers lower frequencies while $\frac{T}{E^2}$ prefers mid to higher frequencies. This indicates that lower frequencies consume less power and also that optimizing for energy efficiency does not automatically optimize for maximum power draw and vice versa.

Finally, Figures 4.3 and 4.4 show the prediction performance of the various ⟨variant, frequency⟩ selection models on a subset of our test dataset for both architectures. The subfigures report throughput (top), energy efficiency (middle), or maximum power draw (bottom) for the selected ⟨variant, frequency⟩ pair for a given input size on the x-axis. We also include numbers for radix and merge sort running at the highest supported frequencies (852 MHz for Jetson TK1 and 875 MHz for Tesla K80) as reference points. For better readability, we focus on 32-bit integer (`int`) type and `uniform` distribution in these figures; additionally, all values are normalized with respect to radix sort at the highest frequency for easier comparison across subfigures. The average throughput, energy efficiency, and maximum power draw (across the full test set) with respect to radix and merge sort is also presented in Table 4.1. From the figures and table, we notice that the power-optimizing models ($\frac{T}{P}$ and $\frac{T}{P^2}$) provide lower maximum power draw (especially for larger input sizes); on average, from Table 4.1, we notice that the savings in power with respect to any one variant is greater than the corresponding loss in throughput for these models. This can be beneficial in datacenters, for example, which place a high importance on power reduction. The remaining models demonstrate improved energy efficiency with respect to the reference variants while ensuring on-par or better throughput, as shown in Table 4.1. This indicates that running variants at the highest supported frequency may not always yield the best energy efficiency. Overall, we demonstrate that input-adaptive ⟨variant, frequency⟩ selection models can provide improved energy and power efficiency with less than a proportional drop in throughput.

4.5 Summary

In this work, we have demonstrated how Nitro can be straightforwardly extended to support multiobjective tuning and combined ⟨variant, frequency⟩ selection. We have also presented the first approach to systematically reducing the energy and power requirements for node-level sorting on two distinct GPU architectures. Specifically, we demonstrated

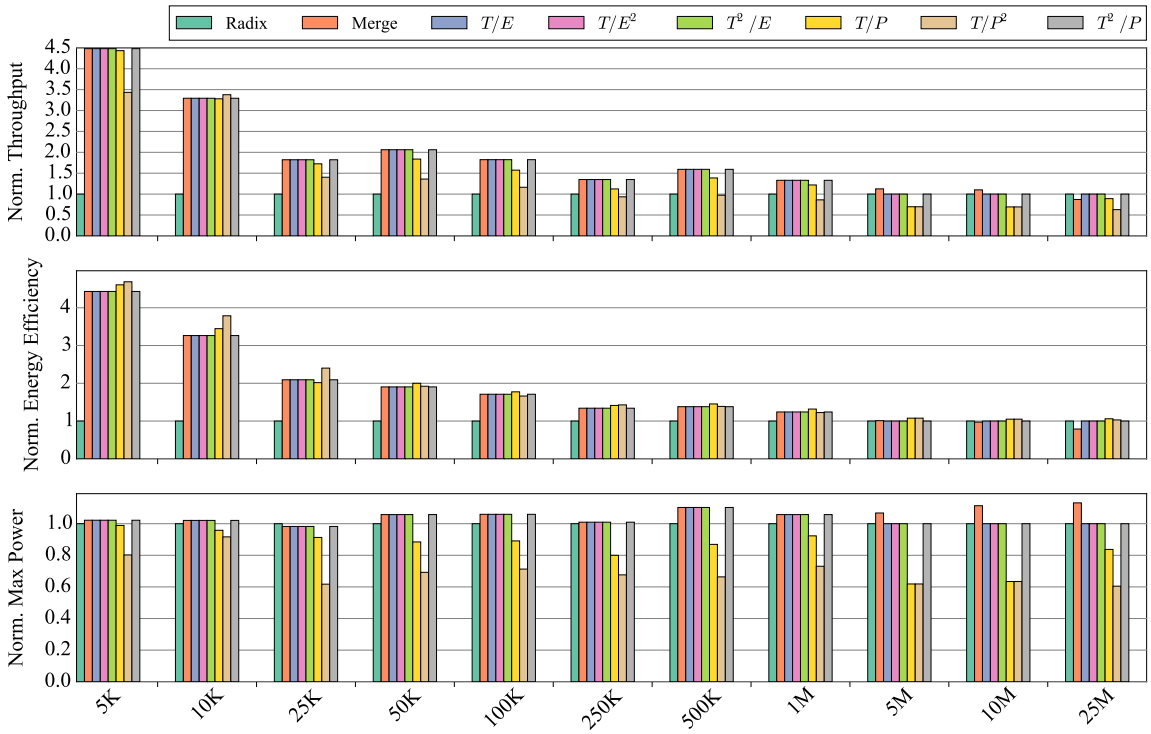


Figure 4.3: Throughput (top), energy efficiency (middle), and maximum power (bottom) on the Jetson TK1 for radix and merge sort, and for variants selected for each optimization objective. Values are normalized with respect to radix sort. Inputs are of type ⟨`int, uniform`⟩.

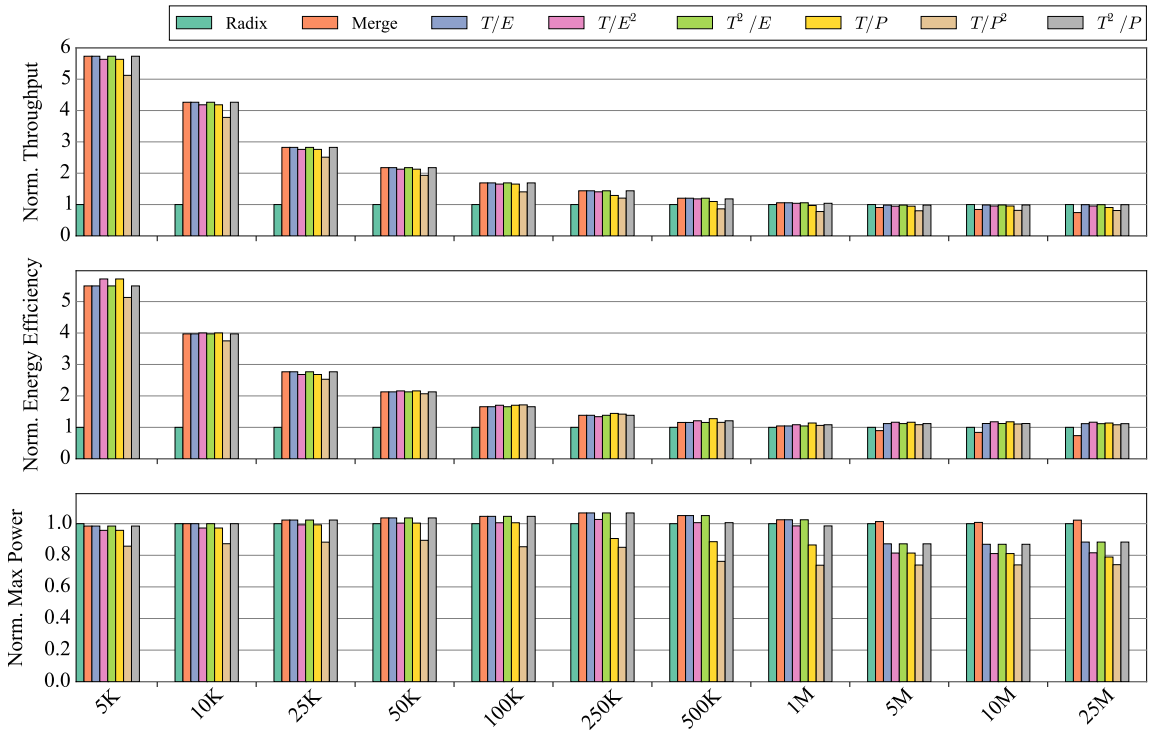


Figure 4.4: Throughput (top), energy efficiency (middle), and maximum power (bottom) on the Tesla K80 for radix and merge sort, and for variants selected for each optimization objective. Values are normalized with respect to radix sort. Inputs are of type ⟨`int, uniform`⟩.

Table 4.1: Throughput (T), energy efficiency (E) and maximum power draw (P) for the variants and frequencies selected by the constructed models with respect to fixed radix and merge sort (at highest frequencies). Values are averaged over all test inputs.

	Jetson TK1																	
	T/E			T/E^2			T^2/E			T/P			T/P^2			T^2/P		
	T	E	P	T	E	P	T	E	P	T	E	P	T	E	P	T	E	P
Radix	2.16	2.37	0.96	2.10	2.33	0.93	2.17	2.36	0.97	1.99	2.38	0.83	1.53	2.35	0.65	2.17	2.36	0.97
Merge	1.01	1.04	0.98	0.99	1.04	0.95	1.02	1.04	0.99	0.91	1.07	0.85	0.70	1.05	0.66	1.02	1.04	0.99
	Tesla K80																	
	T/E			T/E^2			T^2/E			T/P			T/P^2			T^2/P		
	T	E	P	T	E	P	T	E	P	T	E	P	T	E	P	T	E	P
Radix	3.68	3.59	1.00	3.61	3.64	0.95	3.68	3.57	1.01	3.59	3.66	0.93	3.19	3.52	0.83	3.67	3.60	0.99
Merge	1.03	1.08	0.96	1.01	1.11	0.91	1.04	1.07	0.98	0.99	1.12	0.89	0.87	1.07	0.80	1.03	1.09	0.95

improvements in energy and power efficiency with less than proportional loss in sorting throughput. The techniques and algorithms presented in this chapter are easily translatable to a range of data-intensive applications and we expect that this work will serve as a guideline for the development of energy and power-efficient algorithms for a wider array of problems in the near future.

CHAPTER 5

A TUNABLE PROGRAMMING SYSTEM

So far, this dissertation has focused on Nitro, which targets expert programmers. In this chapter, we introduce a new programming system, *Surge*, that aims to simplify adaptive programming for application developers and nonexpert programmers. Surge supports decoupling the high-level specification of computations from their implementation details using first-class constructs. This separation enables it to easily generate a *search space* of multiple low-level, architecture-specific implementations from the same specification. Expert users or autotuners are then able to navigate this search space to find the best implementation for a given execution context.

Surge consists of a programming interface, implemented as a C++ library, and separate code generation and autotuning subsystems. The programming interface is based on *nested data parallelism*, which is a generalization of flat data-parallelism; in nested data parallelism, subcomputations of a data-parallel computation may themselves be data-parallel [46]. It is a powerful abstraction for expressing a variety of parallel computations; further, the algorithmic hierarchy in nested data-parallelism maps naturally to modern processors, many of which have hierarchically organized execution and storage resources (such as GPUs).

The separation between nested data-parallel specification and implementation is achieved using two constructs in the programming interface: *schedules* and *policies*. These represent different levels of abstraction with respect to code generation: schedules provide semantic constraints on the implementation of data-parallel operators, independent of hardware-specific details, while policies encapsulate a set of *optimization parameters* that govern low-level code generation on various hardware platforms. This two-level design makes targeting new platforms easier, and provides a systematic way of automatically generating a search space of valid implementations, which we then navigate using an autotuner. The resulting system is easy to use, but still provides on par or better performance than manually optimized implementations for a variety of computations.

5.1 Programming Interface

Surge exposes a nested data-parallel programming interface, implemented as a C++ library. Programs written using this interface can be compiled to platform-specific code using a standard C++ compiler. Table 5.1 lists the currently supported data-parallel operators, and Listing 5.1 shows an example of how they may be used to express sparse matrix-vector multiplication (SpMV). A basic `sequence` type, denoting a view over contiguous one-dimensional data, is also provided. More complex types of sequences can be built up using operators such as `nest`, as described in Table 5.1. In Listing 5.1, for example, `s_matrix` and `s_indices` are nested sequences (represented internally using the CSR matrix format) constructed from the flat one-dimensional sequences `nonzeros` and `column_indices`, respectively, and the same row offset sequence `row_offsets`; each element of `s_matrix` and `s_indices` thus represents a single row of nonzeros and corresponding column indices of the original matrix.

For each computation in the program, Surge builds an *expression sequence* to capture the nesting structure of its data-parallel operators. Each element of an expression sequence E is a single data-parallel operator, and given two elements e and f in E, f follows e in the sequence (represented as $e \triangleright f$) only if the operator corresponding to f is nested within that of e in the computation. In the SpMV example (Listing 5.1), the outermost `map` (line 6) iterates over matrix rows, and the `reduce` operator on line 14 operates on these rows and is nested within the `map`. The corresponding expression sequence is therefore `map`$\triangleright$`reduce`. Note that the `gather` and `map` on lines 10 and 12, respectively, are not part of the nesting structure; instead they are arguments to `reduce`, and are fused into it as explained in Section 5.3.

An expression sequence is an abstract entity and can be realized in hardware in multiple ways. For example, on CUDA, two implementations of the SpMV expression sequence can be obtained by either assigning each iteration of the outermost `map` to a thread, or to a logical CUDA warp (power-of-2 contiguous group of threads that are at most the physical warp size); the former corresponds to the CSR-Scalar implementation and the latter to CSR-Vector, as described in Bell et al. [47]. To bind an expression sequence to a concrete hardware implementation, Surge introduces two new constructs in the interface: schedules and policies. A schedule, when associated with a data-parallel operator, enforces a semantic constraint on the implementation of that operator; these constraints can then be systematically relaxed to obtain platform-independent implementation strategies. Policies, on the other hand, provide fine-grained control over low-level, platform-specific implementation details by encapsulating the parameters that drive code generation. By exposing schedules

Table 5.1: Current data-parallel operators in Surge. Parameters in square brackets are optional.

Operator	Description
`map(f, s`$_1$`,...,s`$_n$`)`	Produces sequence (`f(s`$_1$`[0], ..., s`$_n$`[0]), f(s`$_1$`[1], ..., s`$_n$`[1]), ...`)
`reduce(`$\oplus$`, s, p)`	Produces the result `(p` $\oplus$ `s[0]` $\oplus$ `s[1]` $\oplus$ `...)` for operator $\oplus$
`reduce_by_key(`$\oplus$`, s, k, p)`	Performs a segmented reduction of sequence s with key sequence k and prefix p
`scan(f, s, p)`	Produces sequence `y` s.t. `y[0] = p` and `y[i] = f(y[i-1],s[i-1])` for operator `f`
`gather(s, idx)`	Produces sequence `y` s.t. `y[i] = s[idx[i]]`
`scatter(s, idx, d)`	Updates sequence `d` s.t. `d[idx[i]] = s[i]`
`range(s, e, [stride])`	Produces sequence with values ranging from `s` to `e` with stride `stride`
`replicate(v, len)`	Synthesizes sequence `s` of length `len` s.t. `s[i] = v` for all `i`
`zip(s`$_1$`,...,s`$_n$`)`	Produces sequence `x` s.t. `x[0]=`$\langle$`s`$_1$`[0],...,s`$_n$`[0]`$\rangle$`, x[1]=`$\langle$`s`$_1$`[1],...,s`$_n$`[1]`$\rangle$`, ...`
`split(s, l)`	Produces nested sequence `x` from `s` s.t. each sub-sequence of `x` is a tile of `s` of size `l`
`join(s`$_1$`,...,s`$_n$`)`	Produces sequence (`x`$_1$`,..., x`$_n$$\rangle_i$) s.t. `x`$_1$ $\in$ `s`$_1$, `x`$_2$ $\in$ `s`$_2$`,...` & $i \in [0, \prod_{i=1}^{n} len(s_i))$
`striding(s, stride)`	Produces strided sequence from `s` of stride `stride`
`reverse(s)`	Produces the reversed sequence of `s`
`nest(s, i)`	Produces a nested sequence from `s`, with subsequence offsets `i`

```
1  // Create nested sequences s_matrix and s_indices
2  auto s_matrix = nest(nonzeros, row_offsets);
3  auto s_indices = nest(column_indices, row_offsets);
4  auto spmv =
5    // Apply dot product across all rows of matrix
6    map([=] (S row, I indices) {
7          auto mul = [](double x, double y) {return x*y;};
8          auto plus = [](double x, double y) {return x+y;};
9          // Gather elements from vector x
10         auto z = gather(x, indices);
11         // Element-wise multiplication of x with row
12         auto vector_mul = map(mul, row, z);
13         // Sum up elements to obtain dot product
14         return reduce(plus, vector_mul);
15       },
16       s_matrix, s_indices);
17 // Realize SpMV computation
18 execute(spmv, s_result);
```

Listing 5.1: Surge code for Sparse Matrix-Vector Multiplication (SpMV). `S` and `I` are defined as `decltype(s_matrix[0])` and `decltype(s_indices[0])`.

and policies in the programming interface, as opposed to embedding them deep in the code generation infrastructure, both autotuners and expert programmers are able to easily experiment with multiple implementations for a computation. We describe schedules and policies in more detail in Section 5.2.

For performance-portable code generation, decoupling computations from their implementations alone is not enough: it is equally important to be able to reason about implementation strategies for computations in a purely platform-independent manner; targeting a new platform then reduces to the problem of finding ways to customize these strategies for that platform. Surge achieves this by keeping the concepts of schedule and policy separate. In contrast, a system that generates platform-specific implementations directly from high-level specifications (regardless of whether they are decoupled or not) must reimplement its entire code generation infrastructure for each new platform.

Invoking the `execute` function initiates the process of binding the expression sequence to a concrete implementation. It has the following form:

```
execute(expr, [destination, platform, schedule, policy])
```

Here, `expr` is the nested data-parallel computation, and the optional `destination` argument specifies where to copy the results of the computation. The `platform` argument is used to specify the target hardware platform. Surge currently supports two platforms: GPUs and x86 CPUs through CUDA C++ and OpenMP, respectively. If this argument is left unspecified, it defaults to CUDA C++. Schedules and policies may be specified through the `schedule` and `policy` parameters, respectively; in this work they are inferred automatically via autotuning (as described in Section 5.2). The values of the parameters `platform`, `schedule`, and `policy` determine a unique implementation for the computation in `expr`. Once specified, `expr` is automatically compiled to either CUDA C++ or OpenMP code using static metaprogramming (described in Section 5.3) and a standard C++ compiler.

5.2 Code Generation and Autotuning

The Surge code generator analyzes nested data-parallel computations in the program and, for each one, systematically enumerates the set of semantically valid schedules and policies. The code generator is implemented as a set of Python modules. Figure 5.1 provides an overview of the code generation process.

5.2.1 Computation Analysis

The job of the analyzer is to extract the expression sequence and platform information for each computation in the program. We avoid using a full-fledged C++ parser for this, and

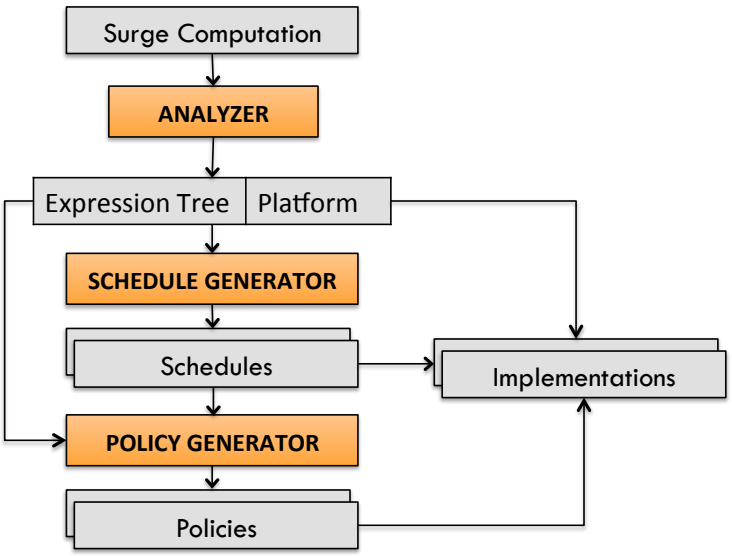


Figure 5.1: Overview of the Surge code generator.

instead rely on a lightweight *introspection* mechanism. The analyzer recompiles the input program with the macro `INTROSPECTION_MODE` defined; this instructs Surge to pretty-print the expression sequence and platform information of the computation instead of evaluating it. The resulting program is run and its output is parsed by the analyzer. Since expression sequence elements are encoded as templated types (as described in Section 5.3), static metaprogramming is used to recursively traverse the expression sequence and print out its information.

5.2.2 Schedule Enumeration

Schedules are defined in terms of *execution resources*, which are platform-specific units capable of carrying out a data-parallel operation. Examples of execution resources include threads, CUDA warps, OpenMP thread-pools, etc. Surge currently supports three schedules: `independent`, `cooperative`, and `sequential` (Table 5.2). Schedules are nested to correspond with the nesting structure of the associated expression sequence. For the SpMV code, an example valid schedule is `independent▷cooperative`: the outermost `map` can process its elements independently, whereas the inner `reduce` (vector dot product) requires a cooperation stage among threads if implemented in parallel.

Schedule enumeration refers to the process of discovering the set of valid schedules for a given expression sequence and platform. It consists of two phases: (1) schedule construction and rewriting, and (2) platform-specific pruning. Before describing schedule enumeration, we first introduce the *schedule rewrite rules*, which make it possible to transform one schedule

Table 5.2: List of Surge schedules.

Schedule	Description
`independent`	Permits the use of multiple execution resources working in parallel
`cooperative`	Permits multiple resources, but they may additionally coordinate with each other
`sequential`	Permits the use of a single thread

to another in a well-defined manner:

```
independent → sequential
cooperative → sequential
```

We define the *strength* binary relation over the set of schedules as follows: A schedule s_1 is said to be stronger than schedule s_2 iff. s_2 can be obtained from s_1 by following the rewrite rules in Surge. Operators implemented using a schedule s can always be implemented using any schedule weaker than s. For example, a `map` operator implemented using the `independent` schedule can always also be implemented using the weaker `sequential` schedule. On nested schedules, rewrite rules are applied one at a time on individual elements.

For a given schedule s and the set of valid Surge rewrite rules RR, the set of weaker schedules $w(s)$ obtained by following rewrite rules is given by

$$w(s) = \{x : (s \rightarrow x) \in RR^{+}\}$$

where RR^{+} denotes the transitive closure of RR.

5.2.2.1 Schedule Construction and Rewriting

The first step of schedule enumeration is inferring the *strongest* nested schedule for the given expression sequence - this is the schedule construction phase. Elements of the input expression sequence are traversed in order, and a *schedule lookup table* (shown in Table 5.3) is consulted to obtain the corresponding element in the strongest schedule's sequence. The strongest obtained schedule, say s, is then rewritten to obtain $w(s)$, the set of all its weaker schedules. Figure 5.2 depicts schedule construction and rewriting for the SpMV example from Listing 5.1, and Figure 5.3 visualizes how the generated schedules may be implemented on the CUDA platform.

5.2.2.2 Platform-specific Rewriting

While generated schedules are guaranteed to be semantically valid, they may not always be directly *implementable* on the given platform. For example, since CUDA only supports

Table 5.3: Schedule lookup table for Surge operators.

Operator(s)	Strongest Schedule
`map, gather, scatter, range, replicate, zip, join, striding, cyclic, reverse`	`independent`
`split, nest`	`independent▷independent`
`reduce, reduce_by_key, scan`	`cooperative`

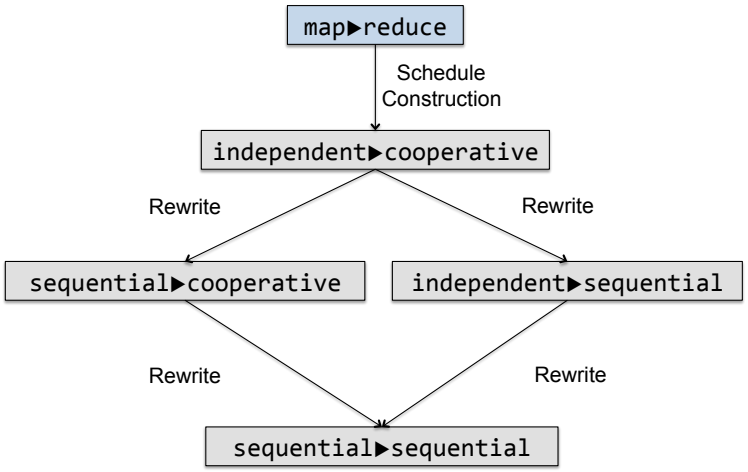


Figure 5.2: SpMV schedule construction and rewriting.

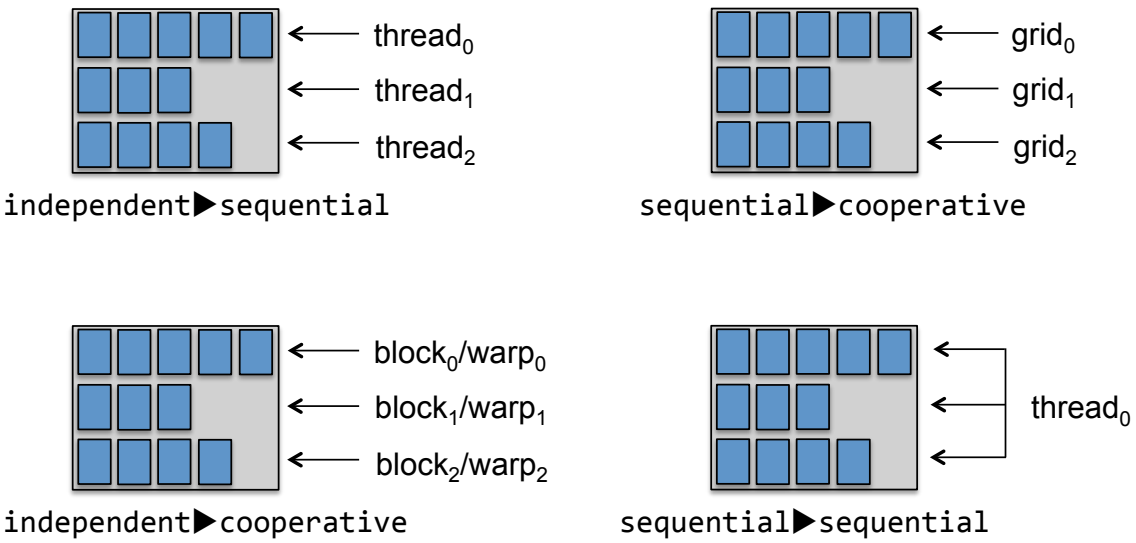


Figure 5.3: How various SpMV schedules may be implemented in CUDA. In this example, the input matrix (gray boxes) has 12 nonzeros (blue boxes) and 3 rows.

two levels of parallelism on a single GPU (at the thread block or warp level and at the thread level), any schedule after nesting level 2 must be `sequential`. Surge thus defines a set of *schedule constraints* for each platform, all of which must be satisfied by each generated schedule. If a constraint failure occurs for schedule s, it is transformed to a new schedule s' using rewrite rules such that s' satisfies all constraints. Note that such a transformation always exists: since schedules of nesting depth n form a bounded lattice under the strength relation, with the weakest element being $\texttt{sequential}_1 \triangleright \texttt{sequential}_2 \triangleright \ldots \triangleright \texttt{sequential}_n$, a nested operator is always implementable on one thread as a set of nested sequential loops.

5.2.3 Policy Enumeration

A policy for an expression sequence E consists of a set of *global* and *local* parameters; the former affect the implementation of the entire computation, while the latter that of the associated element of E. An example of a global parameter is `warp_size` in CUDA, which specifies the number of threads in a logical CUDA warp, while that of a local parameter is `omp_schedule`, which specifies the OpenMP loop schedule to use. Table 5.4 lists the parameters currently supported by Surge.

As shown in Figure 5.1, the final stage of the code generation process is policy enumeration. Let S be the set of valid schedules produced for expression sequence E and platform B. The policy enumerator, for each $s \in S$, generates a set of platform-specific *optimization parameter values* that dictates how the tuple $\langle E, B, s \rangle$ is implemented in hardware. We now describe the two phases of policy enumeration: optimization parameter inference, and search space generation.

Table 5.4: List of tunable parameters.

Parameter(s)	**Type**	**Platform**
`block_size_x, block_size_y, logical_warp_size`	Global	CUDA
`grain_size, block_reduce_algo, block_scan_algo, block_scan_grain_size`	Local	CUDA
`num_threads, enable_nesting`	Global	OpenMP
`omp_schedule, chunk_size`	Local	OpenMP
`execution_resource, enable_unroll`	Local	CUDA/OpenMP

5.2.3.1 Optimization Parameter Inference.

The set of valid optimization parameters T is given by:

$$T = \bigcup_{s \in S} (G_{(E,B,s)} \cup \texttt{Get-Parameters}(E, B, s))$$

where G is the set of global parameters, and `Get-Parameters` (shown in Algorithm 3) is a function that returns the set of valid local parameters for each $\langle E, B, s \rangle$ tuple. The inferred parameters for the schedules in the SpMV example (see Figure 5.2) are shown in Table 5.5.

5.2.3.2 Search Space Generation

Each parameter $t \in T$ can take on a set of values. If r is a function that takes a parameter as input and outputs the list of its valid values, then the search space is $\prod_{t \in T} r(t)$, where $\prod$ denotes a Cartesian product. However, since not all points in this space may be valid on the given platform, search space generation is followed by a pruning phase that discards points that violate platform-specific constraints. For example, while the maximum dimension size of a CUDA thread block along the x and y axes is 1024 each, the total number of threads per block (product of x and y axis dimensions) is also restricted to 1024 on current GPUs such as the NVIDIA Tesla K20c.

5.2.4 Autotuning

The code generation approach from the previous section produces a search space of implementations (code variants) for each computation. To automate the selection of which variant is most appropriate for a given execution context, we rely on Nitro for autotuning. The Surge framework and its interaction with Nitro are depicted in Figure 5.4. Since nested data-parallelism specifically operates on multidimensional (nested) data, having the ability to adapt to input data characteristics is valuable. Surge infers three features automatically

Algorithm 3 Parameter Inference

```
1:  function GET-PARAMETERS(E, B, s)
2:      #E: Expression sequence of form O1▷O2▷...▷On▷φ
3:      #s:  Schedule sequence of form s1▷s2▷...▷sn▷φ
4:      #B: Platform
5:      if E = φ then return ()
6:      else
7:          t ← parameter-set[O1, s1, B]
8:          E' ← O2▷O3▷...▷On▷φ
9:          s' ← s2▷s3▷...▷sn▷φ
10:         return t ∪ GET-PARAMETERS(E', B, s')
```

Table 5.5: Inferred parameters for each SpMV schedule. The subscripts denote nesting depths.

Schedule	**Inferred Parameters (CUDA)**	**Inferred Parameters (OpenMP)**
independent▷cooperative	block_size_x, block_size_y, logical_warp_size, grain_size$_1$, block_reduce_algo$_2$	num_threads, enable_nesting, omp_schedule$_1$, chunk_size$_1$, omp_schedule$_2$, chunk_size$_2$
independent▷sequential	block_size_x, block_size_y, logical_warp_size, grain_size$_1$, enable_unroll$_2$	num_threads, enable_nesting, omp_schedule$_1$, chunk_size$_1$, enable_unroll$_2$
sequential▷cooperative	block_size_x, block_size_y, logical_warp_size, block_reduce_algo$_2$	num_threads, enable_nesting, omp_schedule$_2$, chunk_size$_2$, enable_unroll$_1$
sequential▷sequential	block_size_x, block_size_y, logical_warp_size, enable_unroll$_1$, enable_unroll$_2$	enable_unroll$_1$, num_threads, enable_nesting, enable_unroll$_2$

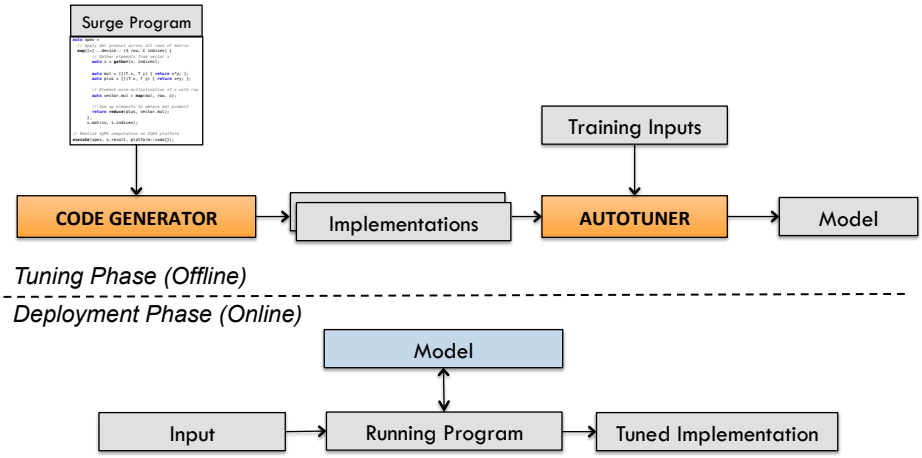


Figure 5.4: Overview of the Surge framework and its interaction with Nitro.

by analyzing the structure of the first input sequence used by the computation: (1) length of the input sequence, (2) aspect ratio for uniformly nested sequences, and (3) average row length for irregularly nested sequences (obtained through the `nest` operator, for example).

5.3 Translation to Target-Specific Code

The Surge programming interface is a domain-specific embedded language (DSEL) [48] with C++ as the host. The primary entities in the programming interface, namely operators, schedules, policies, and platforms, are all implemented as types to enable static metaprogramming. In particular, Surge overloads operators to act as type constructors, as in the expression template idiom [49], to construct the expression sequence at compile-time. This enables the hardware realization of operators to be deferred until an appropriate implementation context (platform, schedule, and policy) is available. As a concrete example, consider `map`: it is recorded as the type `transformed_sequence<F, S1, S2, ...>`, where `F` is the function being applied to every element of sequences S_i. Since a `map`'s iterations can be executed independently, `transformed_sequence` correspondingly provides the subscript operator to realize each individual iteration independently. Thus, in Listing 5.1, the `spmv` variable on line 4 is a `transformed_sequence`, and `spmv[0]` calls the lambda function defined on line 6 with arguments `(s_matrix[0], s_indices[0])`; this returns an object of type `reduced_sequence`, corresponding to the `reduce` operator on line 14.

With the expression sequence constructed, a set of *nested computation kernels*, defined for each platform, is used to realize computations. Each kernel implements a set of $\langle E, B, S, P \rangle$ tuples, where E is the expression sequence, B is the platform, S is the schedule, and P is the policy. Representing schedules and policies as separate types enables us to utilize

the C++ *substitution failure is not an error (SFINAE)* idiom and function overloading to define both generic and highly specialized computation kernels conveniently. For example, Listing 5.2 shows a simple kernel that realizes any valid operator bound with the `independent` schedule on CUDA. In contrast, the type signature of a kernel specialized for the tuple ⟨`reduce_by_key`▷_, `CUDA`, `cooperative`▷`independent`, _⟩ is as follows:

```
template<typename S, typename D, typename policy>
__global__ void nested_kernel(S src, D dest,
                cooperative<independent<>>, policy,
                mpl::enable_if_t<mpl::sequence_traits::
                  is_seg_reduced<S>::value>* = 0);
```

and for the tuple ⟨_, `CUDA`, `independent`▷`cooperative`, ⟨`execution_resource=cuda_warp`, ···⟩⟩ is as follows:

```
template<typename S, typename D, typename policy>
__global__ void nested_kernel(S src, D dest,
                independent<cooperative<>>, policy,
                mpl::enable_if_t<std::is_same<typename
                  policy::sub_policy::execution_resource,
                  res::cuda::warp>::value>* = 0)
```

5.3.1 Targeting New Architectures

Implementing support for a new architecture involves defining a new platform type, and corresponding tunable parameters and nested computation kernels. As described above, the two-layered schedule+policy approach provides a great amount of flexibility while implementing new computation kernels: programmers can start with fairly generic kernels, and then specialize incrementally. This separation also reduces the effort required to add automatic code generation support, as every phase of the code generator need not be re-

```
1  // Implements tuple <_, CUDA, independent, _>
2  template<typename S, typename D, typename policy>
3  __global__ void nested_kernel(S src, D dest,
4                  independent<>, policy) {
5    using iteration_type =
6      typename S::template iteration_type<
7      D, platform::cuda, res::cuda::thread,
8      mpl::global_policy_t<policy>,
9      mpl::sub_policy_t<policy, 1>>;
10
11   const int idx = blockIdx.x*blockDim.x+threadIdx.x;
12   const int grid_size = gridDim.x*blockDim.x;
13
14   iteration_type iterator(src, dest);
15   for(int i = idx; i < src.size(); i += grid_size)
16     iterator[i];
17   iterator.finalize();
18 }
```

Listing 5.2: Sample CUDA nested computation kernel for the `independent` schedule.

implemented; instead, only the platform-specific schedule rewriting and policy enumeration phases need to be implemented for the new platform, as described in Section 5.2.

5.3.2 Operator Fusion

Deferred realization permits operators to be fused together. Consider the case when an operator O_p is an argument to operator O_c; for example, in Listing 5.1, `gather` (on line 10) is an argument to `map` (line 12), which in turn is an argument to `reduce` (line 14). If O_p can be realized using independent iterations, then Surge fuses each iteration of O_p (producer) with that of O_c (consumer) and thus eliminates the need for temporary storage required to realize O_p.

5.4 Benchmarks

We express five benchmark applications in Surge and evaluate their performance on both multicore CPUs and GPUs. To better model a range of real-world applications, the benchmarks are of varying complexity, and are drawn from diverse domains such as linear algebra, machine learning, and physical simulation. Table 5.6 lists our benchmarks, together with the nested operators used, and details about the reference implementations. The remainder of this section describes each benchmark in detail.

5.4.1 Reduction and Scan

For our first set of benchmarks, we implement parallel reduction and parallel prefix scan (scan for short) in Surge. Reduction and scan are fundamental parallel computing primitives that are widely used as building blocks for more complex algorithms [55]. Given a sequence

Table 5.6: List of benchmarks with description, their core computation(s) and details about reference implementations.

Benchmark	Description	Core Computation(s)	Reference Implementation
`Reduction`	Parallel reduction	`map`▷`reduce`	Thrust 1.8.2 [50]
`Scan`	Parallel prefix scan	`map`▷`scan`, `map`▷`reduce`	Thrust 1.8.2 [50]
`SpMV`	Sparse matrix-vector multiplication	`map`▷`reduce`	GPU: CUSP 0.5.1 [9], CPU: MKL 11.2 [51]
`K-Means`	K-Means clustering using LLoyd's algorithm [52]	`reduce_by_key`▷`map`, `map`▷`reduce`	Catanzaro et al. [40]
`CoMD`	Co-design molecular dynamics proxy application	`map`▷`reduce`, `map`	ExMatEx CoMD 1.1 [53], [54]

of input elements $x_0, x_1, x_2, ..., x_N$, a prefix element p, and a binary operator $\oplus$, the output of a reduction is the scalar value $x = p \oplus x_0 \oplus x_1 \oplus ... \oplus x_N$ while that of prefix scan is the sequence $y_0, y_1, y_2, ..., y_N$, where $y_0 = p$ and each $y_i = y_{i-1} \oplus x_{i-1}$.

In the Surge implementation (Listings 5.3 and 5.4), the input sequence is first split into evenly sized tiles which are reduced or scanned in parallel to yield a set of partial results (or partials). These are processed to obtain the final result of the reduction or scan. For the computation of partials, the in-built `reduce` and `scan` operators are instantiated within a `map`, yielding a nested data-parallel algorithm.

5.4.2 Sparse Matrix-Vector Multiplication (SpMV)

SpMV is a critical operation that is used in many iterative methods for solving large-scale linear systems. For this benchmark, we implement the SpMV computation in Surge, as shown in Listing 5.1. The sparse matrix and column indices (`s_matrix` and `s_indices` on line 16) are represented as nested sequences, which are internally stored in a compressed sparse row (CSR) analogue. The outermost `map` (line 6) processes each row of the matrix. Inside the body of the lambda that processes a single row, the correct elements of the vector are first gathered (line 10), multiplied on an element-wise basis with the current matrix row (line 12), and finally summed up to yield the dot product of that row (line 14). Note that the `gather` and `map` on lines 10 and 12 are automatically fused into the `reduce` on line 14, eliminating temporaries (see Section 5.3 for a description of operator fusion).

```
1  auto plus =
2    [](value_t a, value_t b) { return a + b; };
3
4  // Split original flat sequence (s) into C tiles
5  auto s_tiled = split(s, tile_size);
6
7  typedef decltype(s_tiled[0]) row_t;
8  auto row_reduce =
9    [=](row_t row) { return reduce(plus, row); };
10
11 // Compute per-tile reductions
12 execute(map(row_reduce, s_tiled), s_partials);
13
14 // Reduce partials into s_result[0]
15 auto s_partials_tiled = split(s_partials, C);
16 execute(map(row_reduce, s_partials_tiled), s_result);
```

Listing 5.3: Surge code for parallel reduction.

```
auto plus =
  [](value_t a, value_t b) { return a + b; };

// Split original flat sequence (s) into C tiles
auto s_tiled = split(s, tile_size);
typedef decltype(s_tiled[0]) row_t;

// Compute per-tile reductions
execute(
  map([=](row_t tile) {
        return reduce(plus, tile);
      }, s_tiled),
  s_partial_reductions);

// Prefix sum over partial reductions
auto s_partials_tiled = split(s_partial_reductions, C)
execute(
  map([=](row_t tile) {
        return scan(plus, tile);
      }, s_partials_tiled),
  s_partial_scans);

// Compute full prefix sum by seeding from reduction
execute(
  map([=](row_t tile, T prefix) {
        return scan(plus, tile, prefix);
      }, s_tiled, s_partial_scans[0]),
  s_result);
```

Listing 5.4: Surge code for parallel prefix scan.

5.4.3 K-Means Clustering

K-Means clustering is an important algorithm commonly used in fields such as computer vision and signal processing. The problem is defined as follows: given a set of N data points in $D-$dimensional space R^D, and an integer k, determine a set of k points in R^D, called *centroids*, so as to minimize the mean squared distance from each data point to its nearest centroid. We implement a popular heuristic for k-means clustering called Lloyd's algorithm [52]. Additionally, we use the strategy outlined by Catanzaro et al. [40] and rewrite the distance computation $||x-y||^2$ as $x \cdot x + y \cdot y - 2 \cdot x \cdot y$, where x denotes a point and y a centroid. This refactorization lifts the $x \cdot x$ computation out of the main k-means loop and enables the use of vendor-optimized GEMM library calls to efficiently compute $x \cdot y$.

Given an initial set of k means, Lloyd's algorithm proceeds by alternating between two steps: (1) relabeling: assign each point to the cluster with the nearest centroid, where the distance between points is the Euclidean distance; and (2) centroid recalculation: calculate the new cluster centroids as the mean of the values of points in the new clusters. While the Surge k-means implementation consists of five computations, we focus on the more

```
1  // Use a tiled sequence for centroids and points.
2  auto s_centroids = split(s_centroids_flat, d);
3  auto s_points = split(s_data_flat, d);
4
5  //Bring all labels with the same value together
6  thrust::sort_by_key(s_labels.begin(), s_labels.end(),
7                      s_indices.begin());
8
9  auto s_points_x = gather(s_points, s_indices);
10 auto s_prefix = replicate(0, d);
11
12 using point_t = decltype(s_points_x[0]);
13
14 auto plus = [](value_t a, value_t b) { return a + b; };
15
16 execute(
17      reduce_by_key([=] (point_t x, point_t y) {
18                      return map(plus, x, y);
19                    }, s_points_x, s_labels, s_prefix),
20      s_centroids);
```

Listing 5.5: Surge code for k-means centroid recalculation.

expensive centroid recalculation step, the code for which is shown in Listing 5.5. Here, the centroids and data points are stored as tiled sequences (with tile size D) and are obtained by applying a `split` on flat 1-D sequences stored in row-major format (`s_centroids_flat` and `s_points_flat`). Each element of `s_labels`, say `s_labels[i]`, initially contains the label (index) of the closest centroid for data point `i`. We sort `s_labels` to bring all labels of the same centroid together, and store the corresponding point indices in `s_indices` (which initially holds `range(0,` $N-1$`)`). The sum of the points belonging to each centroid can now be obtained through a segmented reduction of `s_points` (permuted through `s_indices`) with key `s_labels`. Since each point is itself in D-dimensional space, we use `reduce_by_key`▷`map` (line 17) to accomplish this. A simple scaling step (not shown in the Listing) then divides the resulting points by their correct counts to obtain the new set of centroids.

5.4.4 Co-design Molecular Dynamics Proxy (CoMD)

CoMD is a molecular dynamics proxy application that is part of the ExMatex project [53]. The workloads seen in the reference CoMD application are representative of those in classical molecular dynamics applications, which is to identify all pairs of atoms under a radius cutoff, and compute the force between these pairs. While the reference implementation supports methods of computing Lennard-Jones (LJ) and Embedded Atom Method (EAM) potentials, we only consider the EAM potential method in this work.

For this benchmark, we express two algorithms for the EAM potential method for computing interatom forces in Surge. One computes the forces directly (Listing 5.6), while

```
1  auto s_space = split(s_boxes, tile_size);
2  auto s_count_range = range(0, s_space.size());
3
4  typedef decltype(s_space[0]) tile_t;
5
6  // Compute partial reductions into s_result
7  execute(
8    map([=](tile_t tile) {
9      real_t etot = 0.;
10
11     // Loop over neighboring atoms,
12     // update force and compute energy.
13     ...
14
15     return etot;
16   }, s_space, s_count_range),
17   s_result);
18
19 // Reduce partials
20 real_t etot =
21   thrust::reduce(s_result.begin(), s_result.end());
```

Listing 5.6: Surge code for CoMD inter-atom force calculation (direct version).

```
1  auto s_domain = range(0, atoms_list.n);
2
3  // eam_force_functor updates the
4  // given atom's force, energy and position.
5  eam_force_functor f(sim, atoms_list);
6
7  // In-place execute
8  execute(map(f, s_domain));
```

Listing 5.7: Surge code for CoMD inter-atom force calculation (redistributed version).

the other performs a domain-specific redistribution of atoms to expose more parallelism before computing the forces (Listing 5.7). The direct method splits the input atom space into evenly-sized tiles and computes partial energy values for each tile. The partials are then reduced to obtain the final energy value. In the redistributed version (Listing 5.7), the `map` on line 8 applies the `eam_force_functor`, which updates a given atom's force, energy, and position, to each atom. Note that both algorithms are expressed in a platform-neutral way and are targetable on both hardware platforms. To enable selection between these two algorithms, we specify them as algorithmic variants using Nitro. We thus obtain a two-level selection process where the algorithm is first selected, followed by the implementation of that algorithm for the target platform.

5.5 Evaluation

In this section, we demonstrate performance and productivity results for the five benchmarks described in Section 5.4. For each benchmark, both CPU and GPU implementations are automatically generated, and the performance of the best ones for each platform (found through autotuning) are compared against hand-written reference implementations for that platform.

5.5.1 Methodology and Hardware Platforms

Our evaluation was run on two hardware platforms: (1) a dual-socket, 32-core Intel Xeon E5-2698 v3 CPU (Haswell) running at 2.30 Ghz, and (2) an NVIDIA Tesla K20c GPU (Kepler generation). The NVIDIA CUDA compiler (NVCC) 7.5 was used, with g++-4.8.2 as the host (CPU) compiler. The `-O3` flag was specified. During CPU results collection, the `KMP_AFFINITY` environment variable was set to `granularity=fine,scatter`. All implementations were run for 100 timing iterations to collect consistent results. Unless otherwise specified, double precision floating-point numbers were used in our evaluation.

Once the benchmarks were specified, Surge automatically generated valid implementations for the desired platform, tuned them, and produced the SVM models. Table 5.7 shows the tuning information for each benchmark, including the features inferred automatically by Surge (as explained in Section 5.2.4), number of training and testing inputs, and size of the search space (number of distinct implementations generated by Surge). When Nitro builds its model in the offline training phase, it automatically finds the best variant corresponding to each training input using exhaustive search; the maximum number of such unique variants across all computations in a benchmark is listed in the last two columns of Table 5.7. We observe that although the initial search space is fairly large, the set of variants for a computation that perform well on a given platform is relatively small.

5.5.1.1 Training and Testing Inputs

As mentioned in Section 5.2.4, the training inputs for Nitro, due to their domain-specific nature, must be provided by the programmer. For reduction, scan, k-means, and CoMD, we generated synthetic inputs for both training and testing; for SpMV, we used sparse matrices from the UFL Sparse Matrix Collection [56]. To obtain representative training sets, we start with a large pool of inputs for each benchmark and use Nitro's active learning heuristic [57] to automatically prune it down and obtain the final training set. The test sets are mutually distinct from the training set. The third column of Table 5.7 shows the number of training and test inputs for each benchmark.

Table 5.7: Features used, number of training and test inputs, size of search space, and number of variants for each benchmark.

Benchmark	Inferred Features	#Inputs		Search Space Size		#Variants	
		#Training	#Testing	GPU	CPU	GPU	CPU
`Reduction`	`#tiles, aspect_ratio`	GPU:8, CPU:6	8	42	36	4	3
`Scan`	`#tiles, aspect_ratio`	GPU:7, CPU:5	8	90	72	4	3
`SpMV`	`#rows, avg_rowlen`	GPU:10, CPU:6	13	42	36	5	4
`K-Means`	`#elements, #tiles, aspect_ratio`	GPU:5, CPU:7	7	40	12	3	5
`CoMD`	`#elements, #tiles, aspect_ratio`	GPU:7, CPU:7	7	48	60	4	3

5.5.2 Performance Results

Figures 5.5 to 5.7 show performance results for our benchmarks on both hardware platforms. In each graph, points on the x-axis represent different inputs from the test set, while the y-axis shows performance in terms of throughput. For each benchmark, we show the performance achieved by the reference and tuned implementations. Note that the performance data shown for the tuned implementations include feature evaluation and SVM model query time. We also include a comparison with the performance achievable if the best implementation among all the generated ones were found via exhaustive search for each test input (lines and bars labeled 'Exhaustive'). The average speedups (over the test set) achieved by the tuned Surge implementation over reference implementations are listed in the second and third columns of Table 5.8.

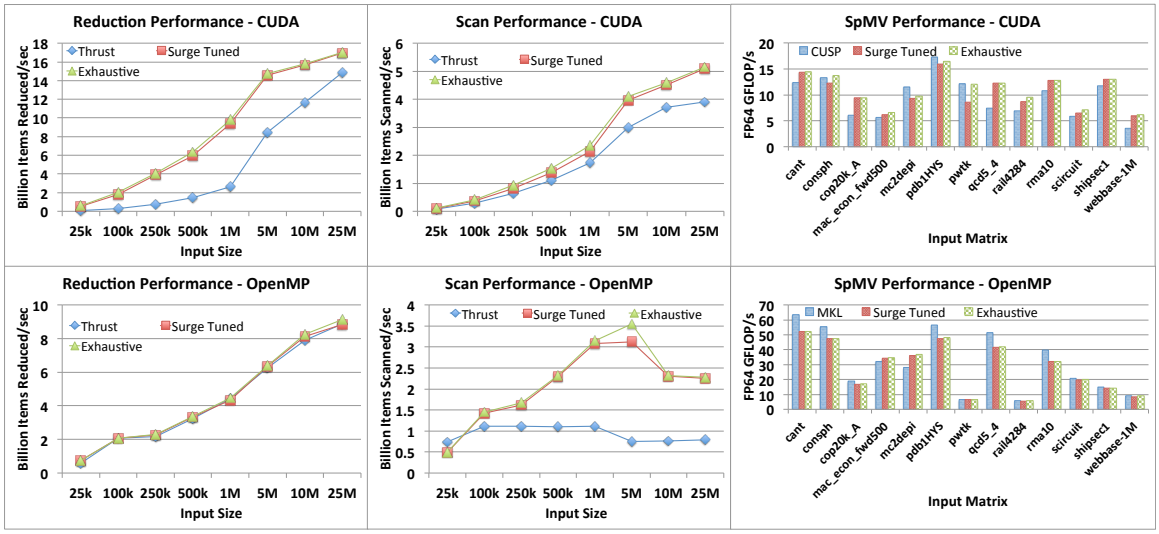


Figure 5.5: Reduction, Scan and SpMV Performance on CUDA and OpenMP.

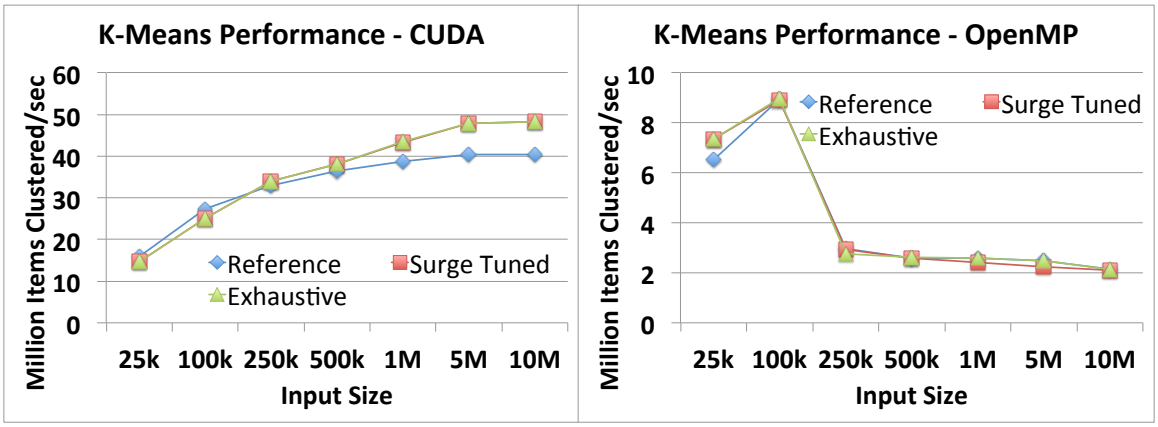


Figure 5.6: K-Means Performance on CUDA and OpenMP.

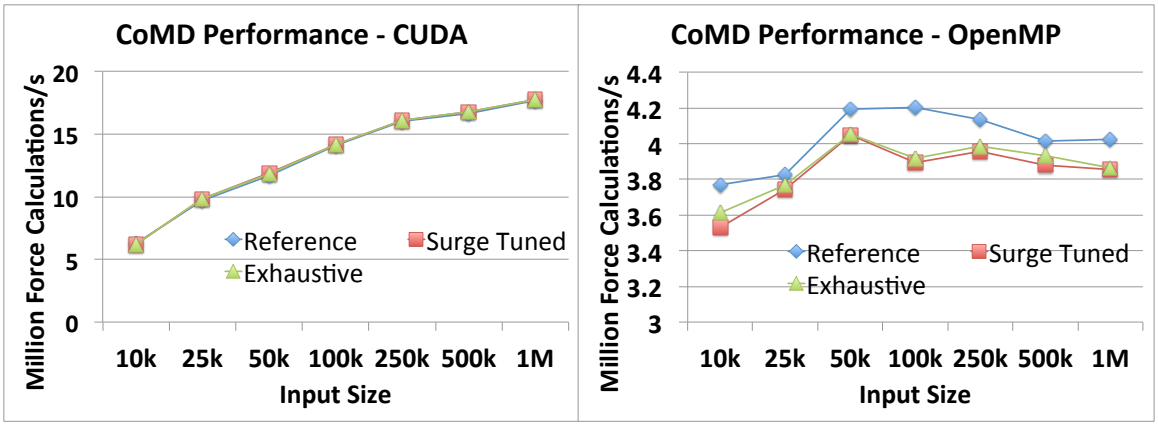


Figure 5.7: CoMD Performance on CUDA and OpenMP.

5.5.2.1 Reduction and Scan

Figure 5.5 shows the performance of reduction and scan. The tuned version either matches or significantly outperforms the reference implementations on both platforms for all test inputs. The performance is especially good on the GPU for small input sizes, where a CUDA warp-based reduction or scan is automatically selected.

5.5.2.2 SpMV

Figure 5.5 shows the performance of the Surge SpMV code (shown in Listing 5.1). We compare against the CUSP CSR Vector and Intel Math Kernel Library (MKL) CSR implementations on the GPU and CPU respectively. The ability to vary logical warp sizes proves to be crucial to obtaining good performance on the GPU, as matrices with smaller average row lengths perform best with smaller logical warp sizes. On matrices with relatively large average row lengths (for example, `rail4284`), the preferred execution resource on CUDA

Table 5.8: Average speedups over GPU and CPU reference implementations, and source lines of code (SLOC) required for Surge, and GPU and CPU reference implementations.

Benchmark	Speedup		SLOC		
	GPU	CPU	Surge	GPU	CPU
Reduction	3.67	1.04	8	88	51
Scan	1.26	2.28	19	96	63
SpMV	1.17	0.93	11	55	unknown
K-Means	1.05	0.98	43	125	71
CoMD	1.01	0.95	71, 78	91	74

seems to be blocks, as opposed to warps. On the CPU, the tuned version performs between 80.5% and 128.8% of highly tuned MKL code.

5.5.2.3 K-Means

For this benchmark, the dimension of each vector is set to 32 and the number of clusters to 10. The algorithm is run for 100 iterations. The Surge version of K-Means uses a `reduce_by_key▷map` operator at the core. On CUDA, the number and size (along both the x- and y-dimensions) of the CUDA blocks turn out to be the most important parameters, and tuning them enables us to beat the reference implementation for larger input sizes.

5.5.2.4 CoMD

Figure 5.7 shows the performance numbers for CoMD. On the CPU, we see that the direct approach, which uses `map▷reduce`, works best, while the version that performs redistribution does well on the GPU. On both platforms, the version selected by Nitro performs on par with reference implementations.

5.5.2.5 Tuning and Overheads

Comparing the performance of implementations tuned by Surge with that of those found via exhaustive search, we notice that the automatically constructed SVM models predict the right implementation for the given test inputs in almost all cases. This imples that the input features added by Surge are highly effective at predicting good implementations. Also, since the inferred features can be computed in constant time, and the number of variants is relatively small for all benchmarks, we observed that the overhead of feature evaluation and SVM model query was negligible (order of a few microseconds). Since applications may be drawn from various domains, we do not claim that the inferred features will always be

sufficient, or that the feature evaluation and SVM model query times will always be this low; instead, we believe that the inferred features, and the integration with Nitro in general, provide a good starting point for tuning the implementations generated by Surge.

5.5.2.6 Summary

Overall, the tuned implementations generated by Surge achieve excellent performance across the board, and often beat the performance of the reference implementations. On the GPU, the ability to vary the execution resource, logical warp sizes, and the number and size of blocks has the most effect on performance. On the CPU, tuning has a less pronounced effect. This is partly because most of the benchmarks operate on uniformly tiled sequences and hence perform well with the default OpenMP schedule. The notable exception is SpMV, which operates on irregularly nested sequences. However, as described by Ohshima et al. [58], the OpenMP scheduling policy seems to affect SpMV performance only when the number of nonzeros is extremely high.

5.5.3 Productivity Gains

We provide a rough measure of productivity by counting the source lines of code required in Surge to express the core computations of our benchmarks (memory management code is not included) and comparing it with the number of lines required to express the same computation in the reference CUDA and OpenMP implementations. In the absence of a superior metric, we believe this captures the conciseness of Surge programs, while still maintaining readability. The last three columns of Table 5.8 show this comparison.

5.6 Summary

This chapter has presented Surge, a nested data-parallel programming system targeted at application developers and nonexpert programmers. By using a two-level mechanism to decouple nested data-parallel computations from their implementations, it is able to systematically generate code variants for multiple platforms. Generated variants are then passed on to Nitro for execution context-aware autotuning. For five real-world benchmarks expressed in Surge, we demonstrate performance that is on par with or better than handcrafted reference implementations on both CPUs and GPUs.

CHAPTER 6

RELATED WORK

Adaptive programming has been extensively studied in the literature. This chapter surveys past and current work in this area and is split into two parts. First, we look at past research on the use of autotuning for supporting adaptivity: specifically, we describe research efforts in the areas of parameter and code variant tuning, input and architecture adaptivity, and multiobjective tuning. We then conclude by looking at past work on higher level programming systems, and comparing it to our work on Surge.

6.1 Autotuning for Adaptive Programming

A large body of research on adaptive programming has focused on using autotuning techniques; in this section, we review relevant prior work on parameter, domain-specific, input-adaptive, architecture-adaptive, and multiobjective tuning, and compare it to the code variant tuning strategies described in this dissertation.

6.1.1 Parameter and Domain-Specific Autotuning

A number of systems support the expression and tuning of optimization parameters. Examples of such systems include Active Harmony [59] (integrated with the CHiLL loop transformation framework [60] to generate variants), POET [61], Orio [62], and more recently, OpenTuner [4]. These can be adapted for code variant generation and tuning using parameterized templates which specify how to generate new variants based on the values of the parameters in the template (found through search). 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 [63], PhiPAC [64], and OSKI [18] for

linear algebra, [65], FFTW [66] and SPIRAL [67] for signal processing, [68]–[70] for stencil computations, and [71] for sorting.

6.1.2 Code Variant Tuning

Several programmer-directed autotuning frameworks support tuning of code variants. Petabricks [8] 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 [72]. Petabricks, however, implicitly tunes variants for the size of the input dataset. Our strategy for input-adaptive tuning, on the other hand, can tune based on any user-defined characteristic of the input data. Brewer [73] 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 runtime is then selected. Sequoia selects variants with user guidance for recursive algorithms that target the memory hierarchy [7].

The closely related problem of algorithm selection was first formally stated and studied by Rice in 1976 [14]. Vuduc [15] provides an evaluation of statistical learning techniques in the context of algorithm selection. Lagoudakis and Littman [16] 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 [17]. Petabricks uses a bottom-up evolutionary algorithm named INCREA [72] 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 [74]–[77]. Luo et al. [78] 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 subsystem, thus replacing or augmenting the SVM-based technique currently employed for input-adaptive tuning.

6.1.3 Architecture-Adaptive Tuning

Performance counters have been used to predict and guide code tuning and compiler optimizations. Cavazos et al. [79] use performance counters to determine good compiler optimization settings. Machine learning is used to learn relationships between performance counter and optimal code optimization settings. Another system introduced by Parello et al. [80] 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. In

comparison, our strategy for architecture-adaptive tuning uses machine learning to build a relationship between performance counters and best device feature subsets, which are subsequently used for cross-architectural tuning.

Machine learning has been extensively used in guiding performance optimizations, as heuristics and exhaustive search are often not practical. Supervised classification has been used to predict unroll factors to improve performance [81]. This problem can be seen as a variant selection problem where the selection depends on features extracted from the code itself. Magni et al. [82] 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 strategy 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.

In summary, to our knowledge we are the first to adapt code variant selection across architectures without retraining, formulating this as a multitask learning problem.

6.1.4 Energy and Power Efficiency Tuning on GPUs

Recent work on improving energy and power efficiency on GPUs has primarily focused on dynamic voltage and frequency scaling (DVFS) [83], [84]. DVFS is often coupled with other optimization techniques such as concurrent kernel execution [13] to improve energy/power efficiency. A survey of existing GPU DVFS techniques is provided by Mittal et al. [85]. To our knowledge, we are the first to demonstrate energy and power efficiency gains on the GPU using execution context-adaptive code variant and frequency selection.

6.2 High-Level Parallel Programming Systems

In this section, we compare the Surge programming system to relevant prior work on nested data parallelism, techniques to decouple computation from implementation, and programming models that support autotuning.

6.2.1 Nested Data-Parallelism

A majority of existing nested data-parallel programming models automatically employ the flattening transformation to convert nested data-parallel operations into flat data-parallel operations [46], [86], [87], which may not be always optimal (see, for example, [88]). This is especially true on architectures such as GPUs, which expose a hierarchical parallelism structure. The notable exceptions are Copperhead [89] and CuNesl [90], which support compiling nested data-parallel operations to match the hierarchical parallelism available in

GPUs. This mapping to hardware, however, is performed automatically by the CuNesl and Copperhead compilers and, unlike Surge (which exposes schedules and policies as part of the programming interface), no mechanism is exposed for experimentation with different mapping and implementation strategies.

6.2.2 Decoupling Computation and Implementation

Outside the realm of nested data-parallel programming models, the concept of decoupling the specification of a computation from its implementation has been explored in the literature. Some flat data-parallel models such as Haskell Parseq [91] and Thrust [50], and more recently, C++17 extensions for parallelism [92], support the use of constructs such as `par` and `seq` to guide the evaluation of data-parallel operators. The Haskell REPA library [93] supports operators such as `map` and `foldAllP` and supports lazy evaluation on numeric arrays. The Galois system [94] adopts a worklist-based approach to enable parallelization of irregular computations, and supports the use of various decoupled runtime schedulers to process work items in parallel. Declarative task-based programming models such as Concurrent Collections (CnC) [95] decouple the high-level program task-graph from its hardware implementation. Computations at the task level are then explicitly specified using a number of different parallel programming models. Charm++ [96] provides an asynchronous message-passing model to describe parallel programs. Halide [1] and Elixir [2] are domain-specific languages that enable users to decouple the specification of image processing pipelines and graph workloads, respectively, from their implementations using schedules. The Delite domain-specific language (DSL) compiler framework [97] uses Lightweight Modular Staging [98] to build an intermediate representation which can represent both parallel patterns and domain-specific constructs. The Delite compiler then compiles parts of the IR to Scala, C++, or CUDA. Similarly, the Lime [99] compiler generates Java code for the entire program, plus OpenCL for GPUs and Verilog for FPGAs; the Liquid Metal Runtime [100] then selects which compiled code to use. Systems such as OpenMP [101] and OpenACC [102], and loop transformation frameworks provide directive-based approaches to parallelize sequential code. Our work, in contrast, is specifically focused on nested data parallelism. Existing systems such as the ones described above, however, need not be mutually exclusive with Surge. For example, languages such as CnC define an entirely separate coordination language within which the programmer describes the data-parallel computation. A combined system can make use of Surge to provide the finer-grained data parallelism.

6.2.3 Programming Models Supporting Autotuning

Recent work has explored the integration of autotuning into parallel programming models. In Tangram [103], expert programmers specify a spectrum of codelets, and the Tangram compiler composes them to generate new ones; the best codelet is then chosen through autotuning. While both Surge and Tangram target performance portability, Surge exposes a higher level, functional interface for expressing computations that does not require expert knowledge to generate high-performance code. Steuwer et al. [104] describe a system that transforms high-level functional expressions into OpenCL code using a set of rewrite rules. By exploring the space of rewrite rules, multiple implementations are generated and autotuned. While this system specifically targets OpenCL code generation, Surge is not designed with any particular platform in mind. It provides separate mechanisms for generating platform-independent implementation strategies (based on schedules), and for customizing these strategies for any target platform (through policies).

6.3 Summary

This chapter has surveyed past and current work in the area of adaptive programming, and compared the strategies presented in this dissertation with relevant past research on autotuning and high-level parallel programming systems. This dissertation makes contributions to the area of adaptive programming by presenting novel research in the areas of input-adaptive code variant selection (based on metainformation other than input size), architecture-adaptive tuning based on multitask learning, and multiobjective code variant and core clock frequency selection. Additionally, our work on Surge introduces a high-level nested data-parallel programming system, tightly integrated with the aforementioned code variant tuning schemes, to target performance portability.

CHAPTER 7

CONCLUSIONS AND FUTURE RESEARCH

This dissertation has presented a framework for adaptive programming, including abstractions for both expert programmers and application developers, and techniques for input-adaptive, architecture-adaptive, and multiobjective tuning. This chapter summarizes the contributions of this dissertation and discusses potential directions for future research.

7.1 Contributions

The individual contributions we have presented in this dissertation are:

1. Nitro, a framework for code variant tuning targeted at expert programmers [57]. We demonstrated how it supports the convenient expression of code variants, together with metainformation for selecting among variants. Further, we showed how it can be used as a substrate for implementing a number of strategies for adaptivity, including support for input-adaptive, architecture-adaptive, and multiobjective tuning.

2. A strategy for input-adaptive code variant selection based on support vector machine (SVM) classification [57]. We additionally described an incremental tuning mode, based on the best-vs-second-best (BvSB) active learning heuristic, that achieves substantial reduction in the training set size. On five high-performance GPU applications, we demonstrated how variants tuned using our strategy achieve over 93% of the performance of variants selected through exhaustive search, averaged over the testing inputs.

3. A novel approach for architecture-adaptive code variant tuning based on multitask learning [105]. Additionally, we introduced profiling and cross-validation-based techniques for pruning device features and demonstrated their importance. Finally, we presented performance results on a set of six benchmark applications and a collection of six NVIDIA GPUs from three distinct architecture generations.

4. A strategy for multiobjective code variant and core clock frequency selection. In particular, we demonstrated how to build a sorting implementation for the NVIDIA Jetson TK1 and Tesla K80 GPUs that provides improved energy and power efficiency with less than a proportional drop in sorting throughput.

5. Surge, a nested data-parallel programming system that decouples the high-level specification of computations from their low-level hardware implementations using two first-class language constructs named *schedules* and *policies* [106]. For five benchmarks expressed in Surge, we demonstrated performance that is on par with or better than handcrafted reference implementations on both CPUs and GPUs.

7.2 Future Work

With the initial framework for adaptive programming in place, we envision a number of possible directions for future research. In this section, we discuss some of these ideas, including adding support for tunable parameters, tuning approximate computations, and operator transformations in Surge.

7.2.1 Support for Tunable Parameters

Optimization parameters are often used in autotuning systems to express large and continuous search spaces. While this dissertation has focused on code variants, we believe that a system capable of tuning both code variants and parameters can be extremely useful, as it would enable concise expression of much larger and richer search spaces. When given an input at runtime, such a system would be able to not only predict the optimal variant, but also values of the parameters. In this discussion, let V be the set of code variants, with each variant v having its own set of optimization parameters T_v.

A simple way to support tuning of parameters is to treat them as code variants; in this case, each unique value of a parameter would correspond to a different code variant. However, since code variant search spaces are assumed to be noncontinuous (for example, each variant may represent a fundamentally different algorithm), this approach ends up not utilizing any continuity that often exists in parameter search spaces; instead, the resulting space is searched exhaustively in the training phase for each input, leading to potentially high training overhead, especially when the search space is large. To overcome this issue, we plan to explore the following strategies that instead employ search heuristics to navigate parameter spaces:

- *Online parameter search:* A simple approach would be to not build any model for parameters; instead, build only a variant selection model, keeping each T_v fixed at user-specified default values. At runtime, the system performs a parameter search (using heuristics such as Nelder-Mead simplex [3]) to find optimal values of T_k, where k is the selected variant, and caches its value for use in subsequent calls to the same computation with the same input. While simple to implement, this approach would suffer from considerable runtime overhead; nevertheless, due to caching, it is expected to perform well when the same computation is called repeatedly with the same input.

- *Multilevel search:* This strategy involves building a model for variant selection, together with $|V|$ models for predicting parameter values for each variant. Techniques such as Kernel Canonical Correlation Analaysis (KCCA) [107] could be used to build the parameter models. The training phase would involve a full search of the parameter space corresponding to each variant for each training input, and could be implemented using existing parameter tuning frameworks such as Active Harmony [3] or OpenTuner [4]. Once optimal parameter values for each variant are known, the best variant can then be computed for the training input being considered. This strategy is expected to yield good variant and parameter models, but could be impractical due to the very high overhead of training data collection.

- *Restricted multilevel search:* To cut down its training time, we could make a simple modification to the above strategy: in the training phase, instead of first finding optimal parameter values and then variants, do it in the reverse order. If all T_v's are initially fixed at user-specified default values, it would be possible to first find the best variant, say k, and then perform a parameter search for variant k alone. This approach reduces training time, and has the additional advantage of not constructing unneeded parameter models (if certain variants are never selected on the given architecture). However, this approach may result in lower accuracy if parameter values strongly influence variant performance.

- *Active learning-based approaches:* Another promising approach involves augmenting the active learning-based incremental tuning mode described in Section 2.2.2 to also handle parameter models. The basic strategy will be similar to multilevel search described above, with the additional goal to reduce the number of inputs required to train the variant and parameter models. Existing active learning-based techniques for

logistic regression, such as the one described by Schein et al. [108], could be extended to achieve this.

7.2.2 Tuning Approximate Computations

Approximate computing trades off computation quality for gains in performance and energy/power efficiency. Systems that support approximate computing typically permit users to specify *error-tolerant* regions of code, together with acceptable error bounds. The system then tries to come up with an optimized implementation that produces less accurate results (subject to error bounds) and ideally better performance and energy/power efficiency. While recent work has demonstrated a number of successes in employing approximation [109], we are not aware of any systems that also take factors of execution context into account.

We believe that the techniques for adaptive programming described in this dissertation can be straightforwardly extended to accommodate approximate computing. For instance, we could add a construct for specifying acceptable error bounds for approximation in Surge. The code generation infrastructure would then be able to generate variants that satisfy these bounds, which would in turn be tuned for multiple optimization objectives by Nitro.

7.2.3 Extensions to Surge

Surge, in its current form, is capable of achieving on-par or better performance than manually optimized code running on CPUs and GPUs, as demonstrated in Chapter 5. However, we believe that there are still a number of opportunities to improve Surge and make it a more useful system. In this subsection, we discuss two such ideas: operator transformations and intercomputation optimizations.

7.2.3.1 Operator Transformations

While Surge schedules can be transformed using rewrite rules, there are a number of cases where transforming *operators* can also be useful. For example, on multi-GPU systems, it may be beneficial to tile a `map` operator into `map`▷`map`, with the outer `map`'s iterations assigned to multiple GPUs, and the inner one's to a single GPU. Since expression sequences are already captured in Surge, it would be straightforward to add new rewrite rules targeted at operators. The code generator could then be correspondingly extended to first rewrite operators, and then schedules, resulting in a richer search space of implementations for autotuning.

7.2.3.2 Intercomputation Optimizations

Certain operator transformations cross computation boundaries; for example, fusion of operators in adjacent computations (akin to loop fusion [110]). While Surge could be extended to collect expression trees for the entire program (as opposed to a single computation), such transformations will additionally require more powerful dependence checking and code generation facilities. We plan to explore the use of compiler frameworks such as CHiLL [111] to implement such transformations.

7.3 Summary

The increasing complexity and diversity of parallel architectures will place a tremendous burden on programmers, who will be forced to constantly rewrite and reoptimize code. The additional challenge of optimizing for multiple, possibly conflicting optimization objectives such as performance and energy/power efficiency compounds this problem. This dissertation has presented novel strategies and abstractions for adaptive programming which reduce the burden on the programmer considerably. While the problem of adaptive programming is far from solved, we believe that this dissertation also provides an important step towards more sophisticated techniques such as adapting code to unseen, currently nonexistent architectures.

REFERENCES

[1] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '13. ACM, 2013, pp. 519–530.

[2] D. Prountzos, R. Manevich, and K. Pingali, "Elixir: A system for synthesizing concurrent graph programs," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '12. ACM, 2012, pp. 375–394.

[3] 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.

[4] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe, "Opentuner: An extensible framework for program autotuning," in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, ser. PACT '14. ACM, 2014, pp. 303–316.

[5] 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. New York, NY, USA: ACM, 2009, pp. 38–49.

[6] 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*, ser. PLDI 2015, 2015.

[7] 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.

[8] 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.

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

[10] 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.

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

[12] A. Mishra and N. Khare, "Analysis of dvfs techniques for improving the gpu energy efficiency," *Open Journal of Energy Efficiency*, vol. 4, no. 04, p. 77, 2015.

[13] Q. Jiao, M. Lu, H. P. Huynh, and T. Mitra, "Improving gpgpu energy-efficiency through concurrent kernel execution and dvfs," in *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '15. IEEE Computer Society, 2015, pp. 1–11.

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

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

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

[24] 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.

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

[26] 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

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

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

[29] 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.

[30] 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.

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

[32] 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.

[33] 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.

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

[35] 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.

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

[37] R. Caruana, "Multitask learning," *Mach. Learn.*, vol. 28, no. 1, pp. 41–75, Jul. 1997.

[38] 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.

[39] S. Sanfilippo and P. Noordhuis, "Redis," http://redis.io.

[40] 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*, ser. PPoPP '14. New York, NY, USA: ACM, 2014, pp. 193–206.

[41] B. Catanzaro, "In-place matrix transposition," https://github.com/bryancatanzaro/inplace.

[42] H. Jordan, P. Thoman, J. Durillo, S. Pellegrini, P. Gschwandtner, T. Fahringer, and H. Moritsch, "A multi-objective auto-tuning framework for parallel codes," in *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, Nov 2012, pp. 1–12.

[43] P. Gschwandtner, J. J. Durillo, and T. Fahringer, *Euro-Par 2014 Parallel Processing: 20th International Conference, Porto, Portugal, August 25-29, 2014. Proceedings.* Springer International Publishing, 2014, ch. Multi-Objective Auto-Tuning with Insieme: Optimization and Trade-Off Analysis for Time, Energy and Resource Usage, pp. 87–98.

[44] R. S. Chen and J. K. Hollingsworth, "Angel: A hierarchical approach to multi-objective online auto-tuning," in *Proceedings of the 5th International Workshop on Runtime and Operating Systems for Supercomputers*, ser. ROSS '15. ACM, 2015, pp. 4:1–4:8.

[45] S. Muralidharan, "GPU Frequency Library," http://github.com/54ur4v/gpu_freqlib.

[46] G. E. Blelloch, "NESL: A nested data-parallel language," Tech. Rep. CMU-CS-95-170, 1992.

[47] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, ser. SC '09, 2009, pp. 18:1–18:11.

[48] P. Hudak, "Building domain-specific embedded languages," *ACM Comput. Surv.*, vol. 28, no. 4es, Dec. 1996.

[49] T. Veldhuizen, "Expression templates," *C++ Report*, vol. 7, no. 5, pp. 26–31, 1995.

[50] N. Bell and J. Hoberock, "Thrust: A productivity-oriented library for CUDA," *GPU Computing Gems*, vol. 7, 2011.

[51] "Intel Math Kernel Library," https://software.intel.com/en-us/intel-mkl.

[52] S. Lloyd, "Least squares quantization in PCM," *Information Theory, IEEE Transactions on*, vol. 28, no. 2, pp. 129–137, Mar 1982.

[53] "DoE Exascale Co-Design Center for Materials in Extreme Environments," http://www.exmatex.org.

[54] N. Sakharnykh, "CoMD-CUDA," https://github.com/NVIDIA/CoMD-CUDA, 2013.

[55] D. Merrill and A. Grimshaw, "High performance and scalable radix sorting: A case study of implementing dynamic parallelism for gpu computing," *Parallel Processing Letters*, vol. 21, no. 02, pp. 245–272, 2011.

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

[57] 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*, ser. IPDPS '14. IEEE Computer Society, 2014, pp. 501–512.

[58] S. Ohshima, T. Katagiri, and M. Matsumoto, "Performance optimization of SpMV using CRS format by considering OpenMP scheduling on CPUs and MIC," in *Embedded Multicore/Manycore SoCs (MCSoc), 2014 IEEE 8th International Symposium on*, Sept 2014, pp. 253–260.

[59] 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.

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

[61] 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.

[62] 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.

[63] 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.

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

[65] 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.

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

[67] 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.

[68] 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.

[69] 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.

[70] 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.

[71] 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.

[72] 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.

[73] 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.

[74] 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.

[75] 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.

[76] 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.

[77] 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.

[78] 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 (SMART'09), colocated with HiPEAC'09 conference*, 2009.

[79] 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*, ser. CGO '07. IEEE Computer Society, 2007, pp. 185–197.

[80] 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.

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

[82] 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*, ser. SC '13. New York, NY, USA: ACM, 2013, pp. 11:1–11:11.

[83] R. Ge, R. Vogt, J. Majumder, A. Alam, M. Burtscher, and Z. Zong, "Effects of dynamic voltage and frequency scaling on a k20 gpu," in *Parallel Processing (ICPP), 2013 42nd International Conference on*, Oct 2013, pp. 826–833.

[84] D. You and K.-S. Chung, "Dynamic voltage and frequency scaling framework for low-power embedded gpus," *Electronics Letters*, vol. 48, no. 21, pp. 1333–1334, 2012.

[85] S. Mittal and J. S. Vetter, "A survey of methods for analyzing and improving gpu energy efficiency," *ACM Comput. Surv.*, vol. 47, no. 2, pp. 19:1–19:23, Aug. 2014.

[86] M. M. T. Chakravarty, R. Leshchinskiy, S. Peyton Jones, G. Keller, and S. Marlow, "Data parallel Haskell: A status report," in *Proceedings of the 2007 Workshop on Declarative Aspects of Multicore Programming*, ser. DAMP '07. ACM, 2007, pp. 10–18.

[87] L. Bergstrom, M. Fluet, M. Rainey, J. Reppy, S. Rosen, and A. Shaw, "Data-only flattening for nested data parallelism," in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '13. ACM, 2013, pp. 81–92.

[88] G. Keller, M. M. Chakravarty, R. Leshchinskiy, B. Lippmeier, and S. Peyton Jones, "Vectorisation avoidance," in *Proceedings of the 2012 Haskell Symposium*, ser. Haskell '12. ACM, 2012, pp. 37–48.

[89] B. Catanzaro, M. Garland, and K. Keutzer, "Copperhead: Compiling an embedded data parallel language," in *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '11. ACM, 2011, pp. 47–56.

[90] Y. Zhang and F. Mueller, "CuNesl: Compiling nested data-parallel languages for SIMT architectures," in *Parallel Processing (ICPP), 2012 41st International Conference on*, Sept 2012, pp. 340–349.

[91] S. P. Jones and S. Singh, "A tutorial on parallel and concurrent programming in Haskell," in *Proceedings of the 6th International Conference on Advanced Functional Programming*, ser. AFP'08. Springer-Verlag, 2009, pp. 267–305.

[92] ISO/IEC, "Programming Languages — Technical Specification for C++ Extensions for Parallelism," http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4507.pdf, 2015.

[93] G. Keller, M. M. Chakravarty, R. Leshchinskiy, S. Peyton Jones, and B. Lippmeier, "Regular, shape-polymorphic, parallel arrays in Haskell," in *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '10. ACM, 2010, pp. 261–272.

[94] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui, "The Tao of parallelism in algorithms," in *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11. ACM, 2011, pp. 12–25.

[95] Z. Budimlić, M. Burke, V. Cavé, K. Knobe, G. Lowney, R. Newton, J. Palsberg, D. Peixotto, V. Sarkar, F. Schlimbach, and S. Taşirlar, "Concurrent collections," *Sci. Program.*, vol. 18, no. 3-4, pp. 203–217, Aug. 2010.

[96] L. V. Kale and S. Krishnan, "Charm++: A portable concurrent object oriented system based on C++," in *Proceedings of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications*, ser. OOPSLA '93. ACM, 1993, pp. 91–108.

[97] K. J. Brown, A. K. Sujeeth, H. J. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun, "A heterogeneous parallel framework for domain-specific languages," in *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, Oct 2011, pp. 89–100.

[98] T. Rompf and M. Odersky, "Lightweight modular staging: A pragmatic approach to runtime code generation and compiled dsls," in *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering*, ser. GPCE '10. ACM, 2010, pp. 127–136.

[99] J. Auerbach, D. F. Bacon, P. Cheng, and R. Rabbah, "Lime: A java-compatible and synthesizable language for heterogeneous architectures," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '10. ACM, 2010, pp. 89–108.

[100] J. Auerbach, D. F. Bacon, I. Burcea, P. Cheng, S. J. Fink, R. Rabbah, and S. Shukla, "A compiler and runtime for heterogeneous computing," in *Proceedings of the 49th Annual Design Automation Conference*, ser. DAC '12. ACM, 2012, pp. 271–276.

[101] L. Dagum and R. Menon, "OpenMP: an industry standard api for shared-memory programming," *Computational Science & Engineering, IEEE*, vol. 5, no. 1, pp. 46–55, 1998.

[102] "The OpenACC Specification version 2.0a," http://www.openacc.org/sites/default/files\discretionary{-}{}{}/OpenACC.2.0a_1.pdf, 2015.

[103] L.-W. Chang, A. Dakkak, C. I. Rodrigues, and W.-m. Hwu, "Tangram: a high-level language for performance portable code synthesis," in *Programmability Issues for Heterogeneous Multicores*, 2015.

[104] M. Steuwer, C. Fensch, S. Lindley, and C. Dubach, "Generating performance portable code using rewrite rules: From high-level functional expressions to high-performance opencl code," in *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP 2015. New York, NY, USA: ACM, 2015, pp. 205–217.

[105] S. Muralidharan, A. Roy, M. Hall, M. Garland, and P. Rai, "Architecture-adaptive code variant tuning," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '16. ACM, 2016, pp. 325–338.

[106] S. Muralidharan, M. Garland, B. Catanzaro, A. Sidelnik, and M. Hall, "A collection-oriented programming model for performance portability," in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP 2015. ACM, 2015, pp. 263–264.

[107] A. Ganapathi, K. Datta, A. Fox, and D. Patterson, "A case for machine learning to optimize multicore performance," in *Proceedings of the First USENIX conference on Hot topics in parallelism*, ser. HotPar'09. Berkeley, CA, USA: USENIX Association, 2009.

[108] A. I. Schein and L. H. Ungar, "Active learning for logistic regression: an evaluation," *Machine Learning*, vol. 68, no. 3, pp. 235–265, 2007.

[109] S. Mittal, "A survey of techniques for approximate computing," *ACM Comput. Surv.*, vol. 48, no. 4, pp. 62:1–62:33, Mar. 2016.

[110] M. J. Wolfe, *Optimizing Supercompilers for Supercomputers.* MIT Press, 1990.

[111] C. Chen, J. Chame, and M. W. Hall, "CHiLL: A framework for composing high-level loop transformations," University of Southern California, Technical Report 08-897, Jun 2008. [Online]. Available: http://www.cs.usc.edu/research/08-897.pdf