Cross-Architecture Performance Prediction and Analysis Using Machine Learning

by

Newsha Ardalani

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

October 2016

Date of final oral examination: 10/20/2016

Final Oral Committee:
Mark Hill, Professor, Computer Sciences
David Wood, Professor, Computer Sciences
Xiaojin Zhu, Professor, Computer Sciences
Dan Negrut, Associate Professor, Mechanical Engineering, Electrical and Computer Engineering
Karthikeyan Sankaralingam (Advisor), Associate Professor, Computer Sciences

For Parental Units.

Acknowledgments

Thanks everyone.

Table of Contents

List of Tables

List of Figures

Abstract

Over the last decade, in response to the slowing or end of Moore's law, the computer architecture community has turned towards heterogeneous systems, where specialized accelerators sit alongside the general purpose processor on chip. These accelerators are released usually with their own specialized programming languages to make the best use of their resources. Although this trend improves energy efficiency and performance, it comes at an increasing software engineering cost – many engineering hours need to be spent to find the best accelerator for each application, and there is no easy way to find the best candidate other than exploring all possible paths, i.e. porting each application into each programming language for each accelerator. To summarize, programmers' productivity is a rising challenge in this new environment.

This dissertation introduces the concept of cross-architecture performance modeling, particularly for CPU to GPU platforms. Cross-architecture performance modeling has the potential to enhance the programmers' productivity, as it is orders of magnitude faster than the existing norm of porting code from one platform to another, and can be used to improve programmers' productivity; Having one performance model per accelerator can help to find the best accelerator and the best algorithm before spending many engineering hours down each path.

1 Introduction

Contents

Although GPUs are becoming more general purpose, GPU programming is still challenging and time-consuming. For programmers, the difficulties of GPU programming include having to

think about which algorithm is suitable, how to structure the parallelism, how to explicitly manage the memory hierarchy, and various other intricate details of how program behavior and the GPU hardware interact. In many cases, only after spending much time does a programmer know the performance capability of a piece of code. These challenges span four broad code development scenarios: i) starting from scratch with no prior CPU or GPU code and complete algorithm freedom; ii) case-(i) with an algorithm provided; iii) working with a large code base of CPU code with the problem of determining what pieces (if any) are profitable to port to a GPU; and iv) determining whether or not a well-defined piece of CPU code can be ported over to a GPU *directly* without algorithm redesign/change. In many environments the above four scenarios get intermingled. This work is relevant for all four of these scenarios and develops a framework to estimate GPU performance before having to write the GPU code. We define this problem as *CPU-based GPU performance prediction*.

We discuss below how CPU-based GPU performance prediction helps in all four aforementioned scenarios. (i) and (ii) *Starting with a clean slate:* Since CPU programming is much easier than GPU programming, programmers can implement different algorithms for the CPU and use the CPU-based GPU performance prediction tool to get speedup estimations for different algorithms which can then guide them into porting the right algorithm. (iii) *Factoring a large code base* (either one large application or multiple applications): When programmers start with a huge CPU code with hundreds of thousands of lines, a CPU-based GPU performance prediction tool can help to identify the portions of code that are well-suited for GPUs, and prioritize porting of different regions in terms of speedup (iv) *Worthwhile to port a region of CPU code*: In some cases, algorithmic change (sometime radical) is required to get high performance and some GPU gurus assert that the CPU code is useless. In these cases, the CPU-based GPU prediction, at minimum, can inform the programmer whether or not algorithmic change is required when tasked with porting a CPU code.

In summary, CPU-based GPU performance prediction has value in many code development scenarios and with the growing adoption of GPUs will likely be an important problem. To the best of our knowledge, there is no known solution for the problem formulated as single-threaded

CPU-based GPU performance prediction, without the GPU code.

In the remainder of this Chapter, we further elaborate on the motivations and the use cases of cross-architecture performance modeling (Section 1.1). Next, we define the foundational terms that underlie the understanding of the rest of the document (Section 1.2). We then explain the principles of cross-architecture performance prediction (Section 1.3). Next, we provide a high-level overview of our model design and structure (Section 1.4 and Section 1.5). Next, we introduce a set of metrics for classification and comparison of related work in this domain (Section 1.6). We formally define our thesis problem in Section 1.7, and discuss our assumptions in Section 1.8. We conclude with the discussion of our contributions in Section 1.9.

1.1 Motivation

In this Section, we elaborate more on the motivations and use cases of cross-architecture performance modeling.

Insufficient Heuristics Advanced GPU programmers follow an intuitive approach to estimate the GPU speedup. There are a set of established heuristics about the influence of branch divergence or memory divergence on GPU speedup. For example, it is believed that branch divergence is detrimental to speedup, while memory coalescing is beneficial. There are three problems with these heuristics:

1. There is no intuition whatsoever on how these individual observations combine; in other words, it is not clear which factor is stronger when multiple positive and negative factors exist.

2. How these program properties interact depends on the underlying hardware.

3. The intuitions on the direction of impact are sometimes false. For instance, the common intuition that branch divergence has negative impact on speedup is not always true. Branch divergence on GPU and branch unpredictability on CPU are often times correlated. However,

there are cases where branch unpredictability hurts CPU performance more than the branch divergence hurts GPU performance, and the overall direction of impact is positive. Therefore, capturing speedup by human inspection of program properties is highly error-prone and likely possible only by GPU gurus.

Algorithmic exploration Often times, programmers start writing a GPU code from scratch, i.e. they are not beginning with an existing CPU code, and often times, there are a handful of different parallel algorithms for solving a single problem [1]. Assuming that the development of each algorithm implementation for the GPU takes one to two weeks, it can potentially take months to find the most suitable algorithm for a GPU. Therefore, it is highly desirable to have a technique to quickly explore different algorithms and find their potential speedup before actually implementing them in GPU. We argue that implementing a parallel algorithm in a serial fashion on CPU, where programmers do not deal with the intricate details of the GPU memory hierarchy is easier than implementing an optimized GPU program. Therefore, having a tool that predicts GPU speedup based on different CPU implementations can help programmers to explore the algorithm space more efficiently.

Region selection The common practice for porting existing CPU programs to GPUs is to first profile the code to find the set of program regions (functions) that account for the majority of the overall execution time, which we refer to as candidate regions. The next step is to find the potentially parallelizable regions among the candidates. This is usually done by eyeballing the candidate regions in order to find the nested regions of a program with no obvious data-dependency across its iterations. Finally, the last step is to port the parallelizable regions in order of their importance, specified by their profiling information.

This process is sufficient if a program is severely imbalanced – a program with N regions is balanced if each region accounts for approximately $\frac{1}{N}$ of the overall execution time and is imbalanced otherwise. Therefore, programmers can port the code in the order specified by the profiling

information. However, for a relatively balanced program, we need a new metric to prioritize porting of the regions. The potential speedup for each region can be used to sort the regions in the order of contributions to overall speedup. We make this topic clear using an example. Given a program with two parallel regions accounting for 60% and 40% of the total execution time and the potential speedup of 2 and 100, respectively, the right order to parallelize the regions is to first port the 40%-region which will bring us up to 50% of the maximum achievable speedup and parallelize the 60%-region next if we have time. Had we relied only on profiling information, we would have parallelized the 60%-region first – which would have get us up to 42% of the maximum achievable speedup – and the 40%-region next. The implications and usage of region selection are multifold. It can be embedded into modern interactive development environments like Eclipse, where programmers can simply highlight a region of code and be immediately presented with the prediction. Further visualization extensions are possible, like showing the prediction speedup of every function, loop, etc. Another use case would be to include this information as part of the future compilation techniques to target only the promising code regions to avoid unnecessary heavy-weight auto-compilation analysis.

Optimization insight One of the major challenges in GPU programming is the application of memory optimization, i.e. the restructuring of the code to fit within one of the many memories within the GPU memory hierarchies. Each memory is designed to optimize certain patterns of memory accesses, and therefore not all programs benefit from this non-trivial optimization. Except for advanced GPU programmers, answering to the two following questions is very hard: which memory optimization to apply and when to apply. Having a model that can quickly predict how the speedup changes as the memory pattern is transformed can significantly improve the GPU programming productivity.

1.2 Terms

This section defines terms and keywords that are frequently used in the rest of this document, and are foundational to the understanding of this work.

Program Properties (Program Features, Features) are conventionally defined as the projection of algorithmic properties on a given hardware. In other words, program properties are defined in terms of their interaction with the underlying *host* architecture/micro-architecture. For instance, cache miss ratio captures the application memory characteristics in interaction with the underlying cache hardware. As has been noted [2], program properties can be defined to be architecture/micro-architecture-independent. One architecture-independent way to capture memory access pattern is characterizing it with the probability distribution function (PDF) of memory access strides, which can be reduced into numeric values in many different architecture-independent ways. Hoste et. al. [2] suggests to break the PDF into different percentiles. In the rest of this dissertation, we use program properties, program characteristics or program features interchangeably to refer to architecture/microarchitecture-independent program properties.

Datapoint is a pair consisting of single-threaded CPU code and the corresponding GPU code. The CPU code is characterized in the form of a vector – where its program properties are the elements of the vector – and the GPU code is characterized by its execution time.

Dataset is a collection of datapoints.

Test set is a set of randomly drawn datapoints from the original dataset, held aside and not used in the model construction process, to evaluate the model accuracy in the end.

Training set is a collection of datapoints from dataset which is not included in the test set, and is used for model construction.

Discretization refers to the process of converting a continuous value to a nominal/categorical value.

Cutpoints or Cutoffs refer to the points that break a range of continuous values into multiple intervals.

1.3 Principles of Cross-Architecture Performance Prediction

We define **Cross-Architecture Performance Prediction** as *the ability to predict the relative performance of an algorithm on architecture B, based on its performance on architecture A, without writing a code/pseudo-code/skeleton for architecture B.* In this dissertation, we focus on a subclass of cross-architecture performance predictors, where $A \in CPU$ and $B \in GPU$. We develop our framework based on the three following principles::

1. **Architecture/microarchitecture-independent program properties are inherent to the algorithm**

   Given an algorithm P and its implementation on platform A, referred to as P_A and its implementation on platform B, referred to as P_B, the first principle states that the architecture/microarchitecture independent properties are inherent to the algorithm and therefore can be collected from any implementation, including P_A or P_B. This can be formulated as follows:

   $$I_{P_A} = I_{P_B} = I_P \tag{1.1}$$

   where I_{P_A} and I_{P_B} represent the architecture/microarchitecture-independent properties of P, obtained from its implementation on platform A and platform B, respectively. I_P represents the properties of the program P that are inherent to the algorithm.

2. **Algorithmic properties and microarchitecture properties interact to dictate performance**

   Given an algorithm P and its implementation on platform A and platform B, the second principle states that there exists a mathematical function that **maps** the architecture/microarchitecture-independent properties of program P, obtained from its implementation on platform A, **to** the performance of its implementation on platform B. Mathematically, this can be formulated as follows:

$$\exists F : \text{Perf}_{P_B} = F(I_{P_B}, H_B) \overset{\text{Principle 1}}{=} F(I_{P_A}, H_B) \tag{1.2}$$

   where H_B represents platform B's architecture/micro-architecture characteristics. The first part of the equation is intuitive, and the second part follows from Principle 1. Therefore, performance of P_B can be predicted from P_A. Note here that this is only possible because of the way we separate the architecture/microarchitecture properties from algorithmic properties.

3. **F is a complex function**

   The correlation between algorithmic properties, architecture/micro-architecture properties and performance is a complex non-linear relationship, with a large number of interacting features. Human-based intuitions or modelings usually fall short in capturing this complexity. Fortunately, machine learning provides powerful tools to discover the relationships where human beings are incapable.

1.4 Model Design

Cross-architecture performance modeling design space can be characterized along multiple orthogonal dimensions. Here we will present the most common ones.

1.4.1 Application scope

To the first order, classes of performance models can be categorized into application-specific, architecture-specific or universal, taking into account whether the models are applicable to any program/kernel, or specific to a particular application. We elaborate on this below.

Application-specific models , where the model is applicable to a particular application or application class. For example, Datta et. al. [3] propose a model that predicts the performance of stencil applications on CPUs, and Meng and Skadron [4], and propose a model that predicts the performance of stencil applications on GPUs. These types of models are usually used for auto-tuning the applications' parameters to minimize their execution time.

Architecture-specific models , where the model is specific to a particular architecture (pair), but is applicable to any/program or kernel. Our model falls under this category. We develop one model per GPU platform.

Universal models , is a model that predicts the performance of any application on any platform. Intuitively, discovering a universal model is very hard.

1.4.2 Feature Measurement

From the implementation perspective, performance models can be classified into three categories depending on the way the program properties are obtained. This simple decision can directly impact the quality of the captured properties, which controls the overall model accuracy, usability, and overhead.

Human-Based feature measurement techniques rely on users to estimate the features. Since human beings are involved, the number of features/inputs into the model are few and roughly estimated. These models are less intended for an accurate performance prediction, and more for

providing a high-level insight into potential performance bottlenecks. Roofline [5] and Boathull [6] models are two examples of human-based performance prediction models.

Dynamic-Based feature measurement techniques rely on having a functional program binary on the host architecture. Program properties will be collected, using one of the many dynamic binary instrumentation techniques, as the program runs. This technique can potentially provide the most accurate estimate of program properties, but it comes at the cost of 10 to 1000 times runtime overhead (as we will show later in Chapter 7).

Static-Based feature measurement techniques automatically analyze the written text of the source code or the intermediate representation of the program (IR) or the static binary to obtain program properties. Since static-based techniques do not rely on dynamic behavior of the program, their feature estimation is less accurate than dynamic-based techniques, but more accurate than human-based techniques. The main advantage of this is its very low overhead (almost instantaneous). In practice, this model is useful for users doing the algorithmic exploration, when no source code is available.

1.4.3 Output Precision

The output of a performance model can be either an exact-value or a range, dictated by its possible use cases. Generally, cross-architecture performance models are used as a programmer guideline on whether or not to port a program to a different architecture. Additionally, they are used to sort different candidate regions based on their potential speedup to prioritize porting.

Range-based models help users to gain insight into the potential benefits of porting a program into another architecture. As the precision of a range-based model increases, it becomes closer to the exact model.

Exact (Continuous) value are usually not required for many of the possible use cases of cross-architecture performance prediction. When a range-based model fails to sort different regions, i.e. it predicts the same performance range for all the candidate regions, an exact model would be preferred. Note that the exact models, in some sense are similar to range-based models, as their continuous output value is associated with some error bar.

1.4.4 Model Construction Technique

Performance models can be roughly classified into two categories based on their model construction approach.

Mechanistic Modeling builds a performance model which provides high-level insights about the system that is being modeled, based on the human understanding of the underlying mechanisms. Amdahl's law, Roofline [5] and Boathull [6] are all the examples of mechanistic modeling. Because of their reliance on human-beings, these models make many simplifying assumptions about the underlying system mechanism, which makes them perfect for understanding the underlying system, but not reliable for accurate performance prediction.

Empirical Modeling builds a performance model which predicts performance using statistical inference and machine learning techniques. Because of their reliance on machine, these models are very complex and hard to interpret, therefore conceived as a "black box" approach.

Machine learning techniques are classified into two major categories, based on their output nature:

- **Regression** models' outcome is a continuous value. Given a set of n observations as training data, the goal of the regression analysis is to find a relationship between input features and the output response, such that the sum of squared errors is minimum. Each observation consists of a vector of p features (also known as independent variables) $x_i = (x_{1i} \dots x_{pi})$ and a response (also known as dependent variable) y_i. $\hat{y}_i$ is formulated in terms of features and

coefficients (β) as follows:

$$\beta = (\beta_0, \beta_1, ..., \beta_p) : \hat{y}_i = \beta_0 + \sum_{j=1}^{p} \beta_j x_{ji} \tag{1.3}$$

We explain regression modeling in more detail in Appendix A. Many researchers [7, 8, 9, 10, 11] have used regression analysis for design space exploration either within a CPU processor or a GPU processor domain, but not across the domains.

- **Classification** model's outcome is a discrete value, known as class or label. Given a set of n observations as training data, the goal of classification is to find a function that maps each data object into one of several classes, such that the misclassification error is minimum. Baldini et. al. [12] have proposed a binary classifier that classifies applications into two classes of slowdown or speedup over a multithreaded CPU implementation.

  In addition to regression and classification models, there is a class of ensemble models which are a set of base learners (which can be either classification or regression or both) combined in a certain way to obtain better predictive performance. Bagging and boosting are two broad classes of ensemble models.

1.5 Model Structure

In this dissertation, we develop a cross-architecture performance modeling framework called XAPP, which uses architecture/micro-architecture-independent program properties to predict the performance of any CPU program on a GPU card. We develop two different variations of XAPP, using two different feature measurement techniques, static analysis-based and dynamic analysis-based, which we refer to as static XAPP and dynamic XAPP, respectively in the rest of this document. Below, we explain the key features of these models.

1. The models are structured such that the program properties and hardware characteristics are independent. The program properties are defined to be architecture/microarchitecture-

independent and are explicitly logged into the model using the input variables, while hardware characteristics are implicitly captured within the model's coefficients and/or the way the features interact.

2. We employ an adaptive two-level machine learning technique. At the first level, we use regression analysis for dynamic XAPP and use decision tree, which is a classification technique, for static XAPP. At the second level, we use ensemble modeling.

3. The model that maps program properties to GPU performance is non-linear. Non-linearity is captured through pairwise feature interactions for linear regression models or is inherent to the learning structure (i.e. the decision tree). In addition, the ensemble function that maps multiple outcomes into one single output is non-linear.

4. Our models are specific to a CPU-GPU pair, which makes them architecture-specific.

1.6 Implementation

An ideal GPU performance prediction framework should satisfy several key properties: *accuracy* – the degree to which the actual and predicted performance matches; *precision* – the granularity of the speedup prediction; *application-generality* – being able to model a wide variety of applications; *hardware generality* – being easily extendable for various GPU hardware platforms; *runtime overhead* – being able to predict performance quickly; and *programmer usability* – having low programmer involvement in the estimation process. We further elaborate on these metrics.

Precision indicates how fine-grain the speedup prediction is. We deem exact speedup prediction as overly precise, while binary speedup prediction (speedup or slowdown) as low precision. Predicting the exact value of speedup is not required for many of the use cases of cross-architecture performance modeling.

Programmer Usability indicates how much programmer involvement is required to make a CPU-based GPU speedup prediction. While some analytical techniques require GPU code to estimate program characteristics, others require extensive source code modification or GPU code sketches. We deem these techniques to have low and medium usability, respectively. Techniques that can work with just the single-threaded CPU implementation have high usability. In our methodology, a user only needs to tag her regions of interest. The entire process is automated, hence XAPP deemed to have high usability.

Application Generality indicates if the technique can target *any* application with *any* level of complexity. There is nothing inherent in our machine learning approach that makes it incapable of predicting certain application types. Kernels within our training and test sets span a large speedup range of $0.8\times$ to $2250\times$, including a wide range of application behaviors, from highly regular kernels to irregular kernels with many branches to even codes that are deemed to be non-amenable to GPUs. Hence, we claim XAPP has high application generality. Application generality can be improved further by adding more applications with different behavior to our training set.

Hardware Generality refers to whether the technique can easily adapt to various GPU hardware platforms. By definition, our models are architecture-specific. Because our program properties are defined to be architecture-independent, generic and exhaustive, we should be able to capture performance on any GPU platform.

Runtime overhead refers to the time needed by the tool to make a prediction. Our tool's runtime overhead can be categorized into two parts.

(1) One-time Overhead: Measuring platform-independent program features for the train set needs to be done only once (by us) and is provided with XAPP. Users must obtain the GPU execution time for all datapoints in the train set for each platform of interest. This requires about 30 minutes. Model construction, a one-time occurrence per GPU platform, takes about 30 minutes for dynamic XAPP and 30 seconds for static XAPP.

(2) Recurring Overhead: The user needs to gather features for the candidate program. For dynamic XAPP, this can take seconds to days — the instrumentation run introduces a $10\times$ to $1000\times$ slowdown to native execution — depending on the actual program execution time. For static XAPP, this always takes milliseconds. Speedup projection completes in milliseconds — it is a matter of computing the function obtained in the previous phase.

1.7 Thesis Statement

This dissertation introduces the concept of cross-architecture performance prediction (XAPP) across CPU-GPU – the capability to predict the GPU performance based on the single-threaded CPU implementation of a program, without writing a single line of GPU programs. We will show that GPU speedup can be formulated in terms of only architecture/micro-architecture-independent program properties as variables, obtainable dynamically from the CPU binary or statically from the CPU source code. We will show that the impact of underlying hardware on performance can be captured implicitly within the model structure in the form of coefficients and/or the way the features interact.

1.8 Assumptions

The following describes our approach in selecting/validating benchmarks. We organize these around the implications they have on the assumptions programmers should make when using our framework.

Assumption 1 Users should assume that the predicted GPU speedup is for a GPU implementation with similar algorithm as the CPU implementation. This is because we selected datapoints with similar CPU/GPU algorithms. We took the following approach: Two algorithms are similar if they produce the same output and not disqualified. We disqualify two algorithms if their computation complexity and/or their number of accesses to global memory are different. That said, common

GPU optimizations (such as loop reordering, loop blocking and overlapped tiling) that change the order of accesses to global memory, but do not change the number of global memory accesses, do not make two algorithms dissimilar, and therefore we allow them in our dataset.

Assumption 2 Programmers should assume that if they provide an optimized CPU code, they will get speedup prediction for an optimized GPU implementation. This is because all the applications within our training set are optimized. A GPU code is optimized if their data structures are mapped into shared memory, constant memory or texture memory on GPU, when applicable. Most of the GPU codes in our training set are written such that they parametrize to the GPU platform and as such do not require tuning. This assumption might not hold true for the recent generation of GPU cards that support dynamic parallelism.

Assumption 3 Programmers should know that our speedup prediction only accounts for the computation time and excludes the memory transfer time.

1.9 Dissertation Contribution

This dissertation makes the following contributions:

Contribution 1 We carefully define program properties, such that the effect of program properties to be independent from the effect of architecture properties on performance. We observe that this simple separation of program properties from architecture properties enables us to construct cross-architecture-specific models whose only inputs are the algorithmic properties – which can be collected from any implementation, including CPU implementation, GPU implementation, or pseudo-code.

Contribution 2 We observe and demonstrate that dynamic program properties are sufficient to predict exact performance value. We observe and demonstrate that statically-determinable program

properties are sufficient to predict performance range.

Contribution 3 We define an essential set of architecture/microarchitecture-independent program properties required for characterizing the performance on GPU.

Contribution 4 We develop a binary instrumentation-based tool that automatically collects dynamic program properties from CPU binaries. We also develop an LLVM-based tool that automates the feature collection from the intermediate representation of a CPU program.

Contribution 5 We present how to adapt existing machine learning techniques to predict performance attained by porting CPU code to GPU.

Contribution 6 Our performance results shows that both static and dynamic XAPP can sustain good accuracy across a wide range of application behaviors. Across a set of widely-known parallel benchmark suites, including Lonestar, Rodinia and NAS, and a set of ill-suited microbenchmarks for GPU, the cross-validation accuracy for static XAPP is 90%, and the average error for dynamic XAPP is 26%. We also test the performance of our tool in practice, through collaboration with researchers in the Statistics department.

1.10 Relation to Author's Prior Work

Table 1.1 highlights the correlation between the chapters of this dissertation and the author's published and under-submission work. A publication appeared in MICRO 2015 [13], and also the US patent 20150261536 had influence on Chapter 3 and Chapter 4. Another related work was submitted to MICRO top-picks 2015, which received honorable mention (top 20 papers that year in computer architecture conferences). This had an influence on Chapter 8. A work under preparation for IPDPS 2017 and another patent under submission have influenced Chapter 4 , Chapter 5 and Chapter 7.

Work	Topic	Chapters
MICRO 2015 [13], US Patent 20150261536	Dynamic XAPP	3, 5
IPDPS 2017 (under preparation)	Static XAPP	4, 6, 7
MICRO top-picks 2015†	Future implications of XAPP	8

† Received honorable mention (top 20 papers that year in computer architecture conferences)

Table 1.1: Author's prior work and its correlation with the dissertation material.

1.11 Dissertation Organization

This dissertation is organized around two pieces: dynamic XAPP and static XAPP. Each piece is organized into two Chapters: the concept and design, and the evaluation. Chapter 2 discusses the related work and places this dissertation in the context of the previous work. Dynamic XAPP will be discussed in Chapter 3 and 5, and static XAPP will be discussed in Chapter 4 and 6. Chapter 7 compares these two concepts/frameworks, and Chapter 8 concludes.

Dynamic XAPP Chapter 3 introduces the dynamic XAPP concept, discusses the set of program properties collected dynamically, and develops a dynamic-binary instrumentation tool to obtain program properties. Chapter 5 analyzes and evaluates the performance of dynamic XAPP on real hardware.

Static XAPP Chapter 4 introduces the concept of static XAPP and develops a static analysis framework for analyzing the intermediate representation (IR) of a CPU code to obtain program properties. Chapter 4 identifies the set of program properties statically collectible. Chapter 6 analyzes and evaluates the performance of static XAPP on real hardware, and studies the impact of dynamic input.

2 | Related Work

Contents

The idea of leveraging **inherent** program properties to **predict** performance **across different accelerators** has never been discussed before. This dissertation is the first that defines and discovers this area. Below, in Section 2.1, we discuss a few related works to this area.

In addition, there has been some broader work on using CPU program properties to predict performance across different CPU microprocessors with different microarchitectural design parameters and/or ISAs, and using GPU program properties to predict performance across different GPU designs. Section 2.2 and Section 2.3, respectively, expand on related work in these areas.

Also, in the context of compilers, a rich body of work exists that explores the automatic code generation across different architectures, which can be re-purposed for our problem statement.

Section 2.4 explains the work in this area. Finally, Section 2.5 provides a comparison against the state-of-the-art techniques.

2.1 CPU->GPU

Although descriptive cross-architecture performance modeling techniques – a technique that provides high-level insights about the underlying system without providing an exact number – such as Amdahl's Law or Roofline model are well-studied and understood, only a few recent works have studied predictive modeling – a model that is capable of predicting performance. Below, we discuss the descriptive and predictive performance models in the cross-architecture performance modeling domain.

2.1.1 Descriptive Modeling

The Roofline model is a simple analytical model that can provide upper-bound projections given platform-specific source code [5]. Roofline is not intended for accurate speedup prediction, but to help programmers detect bottlenecks and improve performance.

2.1.2 Predictive Modeling

Grophecy [14] is a GPU performance prediction framework that begins with a CPU code skeleton. CPU code skeletonization is a manual process where users convert CPU code into an abstract parallel code, where parallel loops are marked and augmented with information about array sizes, the number of loop iterations, and loops being streaming or nested, and data accesses are expressed explicitly as sets of loads and store operations, where addresses are expressed in terms of loop indices, array sizes and other constants. Given the code skeleton, Grophecy generates an abstract GPU code layout, which they characterize into a set of parameters. Given the parametrization of the abstract GPU code layout, they use the GPU analytical model, introduced by Hong and Kim [15], to predict GPU execution time. They have evaluated their technique for three benchmarks and

they have shown their projected performance deviates from the actual performance by 17% in average and 31% at maximum. While accurate, this approach requires an extensive programmer effort to generate the CPU code skeleton. The boat-hull model [6, 16] is another GPU performance prediction framework that approaches the problem from an algorithmic template standpoint. Each CPU program is split into sections, where each section belongs to a certain class of algorithms. They will then use modified Roofline model to predict execution time for each class. They have evaluated their techniques on two image processing applications and has shown 8% deviation in performance projection. While accurate, their approach seems to be applicable to "structured" algorithms like convolution and FFT, and cannot handle arbitrary codes. Meswani et. al. [17] have proposed an idiom-based approach to predict execution time for memory-bound applications. Their model can support only scatter/gather and streaming behavior, and ignores the computation cost and branch divergence impact on overall execution time. Baldini et. al. [12] have proposed a binary predictor for GPU speedup over a multi-threaded CPU implementation. Their goal is not to predict the numerical speedup value and they need an openMP implementation to begin with.

2.2 CPU->CPU

Leveraging program properties to predict performance within the CPU domain has been previously explored, primarily for design space exploration and platform ranking.

Design space exploration A large body of work explores the use of performance modeling for the purpose of design space exploration [10, 18, 19, 20]. Lee and Brooks [10] predict CPU performance over different microprocessor configurations in a large microarchitectural design space, using regression modeling.

Platform ranking Another body of work attempts to find the best CPU platform amongst many CPU microprocessors with different ISAs based on program similarity [21, 22, 23, 24]. Ozisikyilmaz et. al. [24] use neural network and regression modeling to predict SPECint2000 rates – which is

an important metric that determines a system performance, price and marketability – for different microarchitectural configurations.

2.3 GPU->GPU

A related problem is to predict GPU performance from the *existing* GPU code, for GPU design space exploration or auto-performance tuning [15, 9, 25, 26, 27, 28]. Hong and Kim [15] propose an analytical model that predict GPU execution time based on the user input and statically collected values from PTX code. They introduce two metrics of memory warp parallelism (MWP) – the maximum number of warps that can access memory within one memory access period – and computation warp parallelism (CWP) – the maximum number of warps that can execute during one memory access period – to decide if a program is memory-bound or computation-bound, and they provide equations that explain execution-time for each scenario in terms of parameters collected from GPU source code and PTX. Eiger [27], Stargazer6189201 and Starchart [25] are automated performance modeling frameworks that evaluate performance sensitivity to GPU microarchitectural configurations, using machine learning. Huang et. al. [28] propose GPUMech, an interval analysis technique that profiles the instruction trace of every warp. Their goal is to study different architecture design options. They report 14% error when study different warp-scheduling policies. There are also some studies [26, 29, 30] on using modeling to help programmers find the performance bottlenecks in their code and/or to assist auto-tuning compilers. Baghsorkhi et. al. [26] described a method which statically analyzes the GPU source code and generates a program dependence graph (PDG). They analyze the PDG to break GPU execution time into several stages, such as synchronization or branch divergence.

2.4 Automatic GPU Code Generation

Auto-compilation from C/C++ to CUDA or GPU binaries is orthogonal to this dissertation [31, 32, 33, 34]. Thus far, these efforts have not produced high quality code, and no compiler exists

	CPU→GPU Prediction [5, 16, 6]	GPU→GPU Prediction [15]	Auto-Compile [14, 32, 31] [33, 34]	**Dynamic XAPP**	**Static XAPP**
Accuracy	Low	Medium	High	High	High
Precision	Low	Medium	High	Very High	High
Usability	Medium	Low	Medium	High	High
App Generality	High	Low	Low	High	Medium
HW Generality	High	Low	High	High	High
Runtime Overhead	Medium	Low	Low	Medium	Low

Table 2.1: Comparison against the state-of-the-art techniques.

that supports auto-compilation of arbitrary CPU programs. Their scope of applicability remains limited to affine programs. Zhang and Mueller [35] propose a compiler framework for 3D stencil computations, using DSL specifications fed by the user. CUDA-lite [32] and hiCUDA [36] use compiler-directives to translate some specific program patterns in sequential C code into CUDA programs with the help from programmers. KernelGen [33] covers a wider scope of program patterns and does not require directives from programmers. Similarly, OpenACC [37] is an open specification/API for parallel programming which uses compiler directives to specify the program regions within C/C++ programs to offload to GPU and how to offload them. We examined OpenACC in particular, and tried to GPUize benchmarks from our test set applying its pragmas. On irregular kernels the generated code always performed poorly and sometimes had slowdowns, even when the CUDA version had $> 10\times$ speedup (e.g. nn1 and srad1_3). We concluded that OpenACC is not yet effective and lacks application generality. Hoshin et. al. [38] run similar experiments and observed that OpenACC performance is approximately 50% lower than CUDA. We acknowledge that, if such compilers do succeed, tools like XAPP become irrelevant for programmers. That said, tools like XAPP could be used to guide the development of such compilers.

2.5 Comparison

We develop a framework to estimate GPU performance before having to write the GPU code. We define this problem as *CPU-based GPU performance prediction.* Recall that an ideal GPU performance prediction framework should have several key properties: *accuracy* – the degree to which the actual and predicted performance matches; *precision* – the granularity of the prediction provided; *application-generality* – being able to model a wide variety of applications; *hardware generality* – being easily extendable for various GPU hardware platforms; *runtime overhead* – being able to predict performance quickly; and *programmer usability* – having low programmer involvement in the estimation process. *To the best of our knowledge, there is no known solution for the problem formulated as single-threaded CPU-based GPU performance prediction, without the GPU code.* The literature on GPU performance prediction from GPU code, sketches, and other algorithmically specialized models can be re-purposed for our problem statement and evaluated using our six metrics [5, 16, 6, 14, 32, 31, 33]. Table 2.1 categorizes them according to these six metrics. As shown in the Table, no existing work can achieve all six properties.

3 | Dynamic XAPP: Concept and Design

Contents

Dynamic XAPP is an instance of cross-architecture performance prediction framework introduced in Chapter 1. It predicts the exact continuous value of speedup and uses dynamic binary instrumentation to obtain program properties with high accuracy. We anticipate programmers will use this tool early in the software development process to save time.

3.1 Overview

Problem Statement GPU programming is challenging and time-consuming. We propose dynamic XAPP (Cross-Architecture Performance Predictor), an automated performance prediction tool that predicts GPU performance with high accuracy and precision, when provided a piece of CPU code *prior to developing the GPU code*. Considering some GPU platform x, we will show that

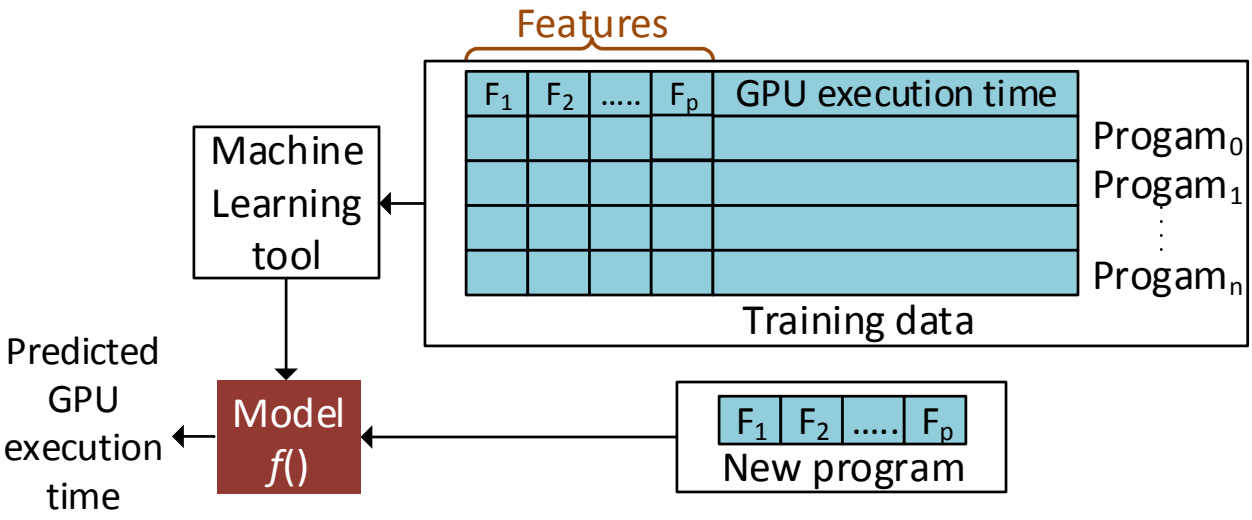


Figure 3.1: Dynamic XAPP overall flow.

some mathematical function exists that map program properties (features) to GPU execution time. In other words, considering features $f_0, f_1, f_2, ...$, we will show that there exists an F_x such that: GPU Execution Time $= F_x(f_0, f_1, f_2, ...)$, where the only inputs to the function are program properties and all the other platform-dependent properties are embedded in the function as constants. This formulation is the key novel contribution of this work. In mathematical terms, this formulation is indeed simple. However, it enables us to collect program properties from any implementation, including the CPU implementation, GPU implementation or algorithm.

Note that dynamic XAPP's goal is not to predict how to port a C/C++ code to GPU, but to predict its performance on GPU, if time was invested to develop an optimized CUDA code.

Insight Dynamic XAPP is built on the two following insights: i) GPU performance varies between different programs and different GPU platforms. Each program can be characterized by a set of micro-architecture-independent and architecture-independent properties that are inherent to the algorithm, such as the mix of arithmetic operations. These algorithmic properties can be collected from CPU implementation to gain insight into GPU performance. ii) By examining a vast array of previously implemented GPU codes along with their CPU counterparts, we can use machine

learning (ML) to learn the non-linear relationship between quantified program features collected from the CPU implementation and GPU execution time measured from the GPU implementation. Figure 3.1 shows the overall flow of dynamic XAPP.

Practical Implementation We take the following steps:

1. **Feature definition (Section 3.2)** The first step toward learning this function is defining the set (ideally the exhaustive set) of features that are inputs to this function.

2. **Function discovery (Section 3.3)** With the above step completed, mechanistic models, machine learning, simulated annealing, deep neural networks, or various other modeling, optimization, or learning techniques, can be used to learn this function. It is presumed that learning the exact function is practically impossible and hence some analysis is required on the learned function.

3. **Analysis (Section 5.2, 5.3)** Once this function is learned, one can analyze the function to test if it is meaningful given human understanding of programs and how they actually interact with hardware, measure the accuracy on some real meaningful test cases, and consider other metrics.

Given the main observation, performing the above steps is quite straight-forward engineering. These steps are, however, necessary to demonstrate that the problem, as formulated, is solvable (the function can be discovered) in a meaningful manner, which is the focus of the rest of this Chapter.

We conclude with a comment on the role of GPU x. Observe that we defined that a unique function exists for each GPU platform. Implicitly, this captures the role of the hardware. We could have defined a broader problem that characterizes GPU x with its own features $x_0, x_1, ..$ to discover a single universal function G, which can predict execution time for any GPU platform, and any application: GPU Execution Time $= G(x_0, x_1, x_2, ..., f_0, f_1, f_2, ...)$. Undoubtedly discovering G is significantly more useful than having to discover F_x for each GPU platform. Intuitively discovering G seems very hard, and we instead seek to discover $F_x()$.

3.2 Program Features

Feature	Range	Description	Relevance for GPU speedup
ilp.$(2^5, 2^8, 2^{11}, 2^{16})$	1-Window Size	Avg num. of independent operations in a window IW, where IW of sizes $(2^5, 2^8, 2^{11}, 2^{16})$ are examined.	Captures the potential for parallelism.
mem/ctrl/int	0 -1	Fraction of the number of operations that are memory/control/integer arithmetic operations.	
coldRef	0 - 1	Fraction of memory references that are cold misses, assuming a block size of 128 B.	Captures cache effectiveness.
reuseDist2	0 - 1	Fraction of memory references that have a reuse distance of less than 2.	Captures cache effectiveness.
ninst	0 - inf	Total number of instructions	
fp/dp	0 -1	Fraction of single-/double-precision floating-point arithmetic operations.	
stride_0	0 - 1	Group every 32 consecutive instances of a static load/-store into a window, calculate the fraction of windows in which all instances access the same memory address.	Captures suitability for constant memory.
noConflict	0 - 1	See Section 3.2	Captures bank conflicts in shared memory.
coalesced	0 - 1	See Section 3.2	Captures global memory coalescing.
shMemBW	0 - 1	See Section 3.2	Captures shared memory bank-effectiveness.
gMemBW	0 - 1	See Section 3.2	Captures memory throughput.
blocks/pages	0 - #blocks	Avg. number of memory accesses into a block of 128 B/4KB granularity.	Captures locality & cache effectiveness.
ilpRate	1 - 16384	ILP growth rate when window size changes from 32 to 16384.	Captures amenability to GPU's many-threaded model.
mulf/divf	0 - 1	Fraction of single-precision floating-point operations that are multiplication/division operations.	Captures the effect of a GPU's abundant mul/div units.
sqrtf/expf/sincosf	0 - 1	Fraction of single-precision floating-point operations that are square root/ exponential or logarithmic/ sine or cosine functions.	Captures the effect of SFU.
Lbdiv.$(2^4 - 2^{10})$	0 - 1	See Section 3.2	Captures the branching pattern.

Table 3.1: List of program properties used as input features.

Determining the set of features that are required for defining $F_x(f_0, f_1, f_2, ...)$ involves two difficult challenges: discovering the explanatory features and formulating them in quantifiable ways. There is also a subtle connection between feature definition and function discovery. If a function discovery technique can automatically learn what are the important features, then one can be aggressive and include features that may ultimately not be necessary. There is no algorithmic way, that we know of, to define a list of features. We started with a list of features that have been

used in previous workload characterizations, and defined several new features that seem plausibly related to GPU performance. GPU execution time is dictated strongly by the memory access pattern and how well it is coalescable, the branching behavior and if it causes warp divergence, how well shared memory can be used to conserve bandwidth, and even somewhat esoteric phenomenon like bank conflicts. This intuition on GPU hardware serves as the guide to determining a set of good explanatory features.

Table 3.1 lists the set of all program properties we have used in our model construction, and how each feature is correlated with performance on GPU hardware. In Chapter 5, we describe the dynamic binary instrumentation tool we developed to measure these properties. Below, we explain each of these features in more detail.

Global Memory Bandwidth Utilization ($gMemBW$ and $coalesced$) Non-coalesced memory accesses are known to hurt GPU performance. At every load/store operation, if a warp can coalesce its memory accesses into one single memory transaction, it achieves 100% memory transaction utilization. If a warp coalesces all of its memory accesses into two memory transactions, it achieves 50% memory transaction utilization. If a warp coalesces all of its memory accesses into three memory transactions, it achieves 33% memory transaction utilization. In the worst case scenario, a warp generates 32 different memory transactions for every load/store operation. This reduces the utilization by 1/32. We estimate effective memory transaction utilization ($gMemBW$) by $\sum_{i=1}^{32} \frac{MW_t[i]}{i}$, where $MW_t[i]$ is the number of MW windows which coalesce into i memory transactions, normalized to the number of windows across the application runtime. Another feature, $coalesced$ specifically captures the case where i is one.

Shared Memory Effectiveness (shMemBW and noConflict) Bank conflicts in shared memory are also known to have negative impact on GPU performance and it is more likely to happen for certain memory access patterns. For example, applications with regular memory access patterns, where the strides of accesses are either 2-word, 4-word, 8-word, 16-word or 32-word will get a

2-way, 4-way, 8-way, 16-way or 32-way bank conflict, respectively. In absence of any bank conflict, 32 (16) words can be read from 32 (16) banks every clock cycle for GPU platforms with compute capability>3.X (compute capability<3.X), but an X-way bank conflict reduces the bank effectiveness by 1/X. Therefore, we will estimate bank-effectiveness ($shMemBW$) by $\sum_{i=0}^{5} \frac{MW_r[2^i]}{2^i}$, where MW is a window of 32 consecutive memory operations generated by the same PC, and MW_r is a MW window with constant stride. $MW_r[2^i]$ represents the number of MW_r windows with $stride = 2^i$, normalized to the number of windows across the application runtime. Another feature, $noConflict$, specifically captures the case where the $stride$ is an odd number.

Constant Memory Effectiveness ($stride_0$) Constant memory on GPU is suitable when all the threads within a warp are accessing the same memory location. Otherwise, memory accesses would get serialized. $stride_0$ characterizes the suitability of each static load/store memory operation for constant memory, by considering every 32 consecutive dynamic instances of each memory operation grouped into a window, and calculating the fraction of windows where all instances access the same memory address.

Temporal Locality ($coldRef$, $reuseDist_2$) To quantify the temporal locality of memory accesses (and therefore cache effectiveness) in a micro-architecture-independent way, we adapt $reuseDist$ and $coldRef$ feature from MICA [2]. MICA characterizes temporal locality by looking at the distribution of least-recently-used (LRU) stack distances. The LRU stack distance of a memory operation is defined as the number of unique memory operations between two accesses of the same memory address. $reuseDist_2$ shows the probability of the distances to be ≤ 2. Another feature, $coldRef$ measures the portion of memory operations that are cold. A memory access is considered cold if it is not accessed before. Hoste et. al. provide an elaborate explanation about these features [2].

Memory Footprint ($blocks$, $pages$) The amount of memory the application uses during its execution time, known as memory footprint, can affect GPU execution time. If the memory footprint is smaller than the GPU's scratchpad memory the GPU code would benefit from shared memory.

We characterize this feature as the number of unique 128-byte blocks ($blocks$) and the number of unique 4KB pages ($pages$) that were touched during the program execution time.

Branch Divergence (Lbdiv) Another program feature that could degrade performance is branch divergence. Consider the local branch history per branch instruction, divided into consecutive windows of X decisions, W_X. We estimate branch divergence ($Lbdiv_X$) as the fraction of the number of W_X windows where the branches within them are not going in the same direction.

Instruction-level Parallelism (ILP) Modern GPUs capture instruction-level parallelism (ILP) through pipelining [39]. ILP represents the average number of independent instructions that can execute in parallel within one cycle. We estimate ILP_X as the average number of independent instructions over a window of X subsequent dynamic instructions. We measure ILP for different window sizes (2^5, 2^8, 2^{11} and 2^{16}). Hoste et. al. explain this feature in more detail [2].

Parallelism Potential ($ILPRate$) GPUs follow the SIMT execution model that requires the program (algorithm) to be partitionable into somewhat coarse-grained regions that can execute concurrently. We define $ILPRate$ as the ratio of the ILP in a large window (16384) to the ILP in a small window (32) to capture the potential for coarse-grain parallelism.

Single-Precision Floating Point Transcendentals ($sqrtf$, $expf$, $sinf$) Floating-point transcendental operations have dedicated hardware support on GPUs [40] and the programs containing any of these operations show significant performance improvement on GPU. This property can be characterized by determining the percentage of the floating-point transcendental operations in the program. We define $sinf$ as the ratio of the single-precision floating-point sine and cosine operations to the total number of single-precision floating-point instructions. We define $expf$ as the ratio of the single-precision floating-point exponential and logarithm operations to the total number of single-precision floating-point instructions. We define $sqrtf$ as the ratio of the floating-point square root operations and the reciprocal square root operations to the total number of single-precision

floating-point instructions. Note that we classify the transcendental operations into three separate features, as each feature has a different impact on speedup. Through micro-benchmarking, we discovered that trigonometric operations have the highest benefit on GPU, while the square root operations have the lowest. Trigonometric operations are usually computed using math libraries on the CPU, using iterative algorithms requiring many instructions. The square root operation has ISA support, meaning less CPU overhead and less available speedup for the GPU.

Single-Precision Floating Point Multiplication/Division Codes containing single-precision floating-point division or multiplication tend to show higher performance on GPU, since they map to a few native instructions (when compiled with $-use\_fast\_math$ flag). We define $mulf$ and $divf$ as the ratio of single-precision floating-point multiplication and division operations, respectively, to the total number of single-precision floating point instructions.

Instruction Mix (mem**,** $ctrl$**,** int**,** fp**,** dp**)** The ratio of memory operations to arithmetic operations directly impact the GPU capability to hide long latency memory operations or computations. mem captures the fraction of instructions that are memory operations. int captures the fraction of instructions that are integer arithmetic operations. fp captures the fraction of instructions that are single-precision floating-point arithmetic operations. dp captures the fraction of instructions that are double-precision floating-point arithmetic operations. $ctrl$ captures the fraction of instructions that are controlled by conditional branches. Note that diverging branches that control a small portion of the program have low impact on GPU speedup. $ctrl$ and the previously defined $bdiv$ features together capture this property.

Number of Instructions ($ninst$**)** Finally, GPU execution time directly correlates with the dynamic number of instructions running on GPU, which itself correlates with the dynamic number of instructions running on CPU.

3.3 Machine Learning Approach

After defining program properties (input features in machine learning terminology), the next step is to discover the function that captures the correlation between GPU execution time and CPU program properties. We use machine-learning, specifically the ensemble of forward step-wise regression learners. This section explains our machine learning (ML) technique choice and its construction in detail. We emphasize there is nothing novel in our ML technique, and it is a straight-forward application of established ML techniques. We define the term data point first. To aid explanation, recall that a "data point" is a pair consisting of single-threaded CPU code and the corresponding GPU code. The CPU code is characterized in the form of a vector – where its program properties are the elements of the vector – and the GPU code is characterized by its execution time.

3.3.1 Overview

We employ an adaptive, two-level machine learning technique. We begin with regression as our base learner for the following reasons: (1) Regression is a mature, widely-used ML technique, that can capture non-linearity using *derived features*, such as pairwise interaction and higher-order polynomials. (2) It is a natural fit for problems with real-valued features and real-valued outputs.

We then combine the predictions of multiple learners to make the final prediction. This second level is critical in our technique as different applications require different sets of features to explain their execution time, and we do not have enough training data to allow all features appear in one single model without the risk of overfitting. Instead, we construct smaller models and automatically decide which models are likely to explain the execution time better. The decision on which models to pick is simple - we select 60% of the most similar models in terms of the output. This technique is known as ensemble prediction, and theoretical and empirical results show that it improves the base learner accuracy [41, 42, 43]. In Subsection 5.2.3, we discuss the analysis we made that ultimately lead us to use an ensemble solution.

Model	ninst*(	Lbdiv32	+mem	+gMemBW	+gMemBW:Lbdiv32	+pages	+pages:Lbdiv32
Coefficient		0.0290038	0.0126532	-0.0228180	-0.0070995	-0.0380114	0.1076867
Adjusted R^2			0.1781	0.2704	0.3173	0.3685	0.4697

Model	+stride0	+stride0:pages	+dp	+blocks	+blocks:pages	+shMemBW	+shMemBW:mem
Coefficient	-0.0027127	-0.0196313	-0.0045341	0.0968233	0.0010343	-0.0036923	-0.0072603
Adjusted R^2	0.5061	0.5412	0.5649	0.5776	0.596	0.6024	0.6634

Model	+ilp256	+ilp256:stride0	+arith	+div	+div:mem	+coalesced	+coldRef)
Coefficient	-0.0014900	-0.0036693	-0.0016565	-0.0057103	-0.0064187	0.0130530	0.0017546
Adjusted R^2	0.6678	0.6915	0.6969	0.699	0.7152	0.7162	0.717

Table 3.2: An instance of a regression model generated at first level.

3.3.2 Implementation Details

Here, we explain the details of our adaptive two-level machine learning technique. We first explain our model construction procedure for the regression model at the first level and then the ensemble algorithm at the second level. In the end, we briefly discuss the other machine learning techniques we explored, but failed to generate accurate result.

Level 1: Regression We use forward feature selection stepwise regression [44] as our base learner. What this means is that every model starts with with zero features, then we evaluate all the models with one feature, and add the feature that yields the best performance (the highest adjusted R^2) to the model. Next, we evaluate all models with an extra feature and add the one that yields the best performance to the previous model. As we add new features, we also consider the interaction terms with existing features in the model, and add them only if the performance improvement is above a threshold, θ_1. We repeat this process until the improvement from the new feature is less than a threshold, θ_2. Empirically, we found $\theta_1 = 0.0095$ and $\theta_2 = 0$ generates good accuracy models. Table 3.2 shows an example byproduct of this stage. To reduce the space search, we use our expert knowledge and enforce the number of instructions ($ninst$) as a multiplicative term before the model construction starts. Jia et. al. elaborate this technique in detail [9].

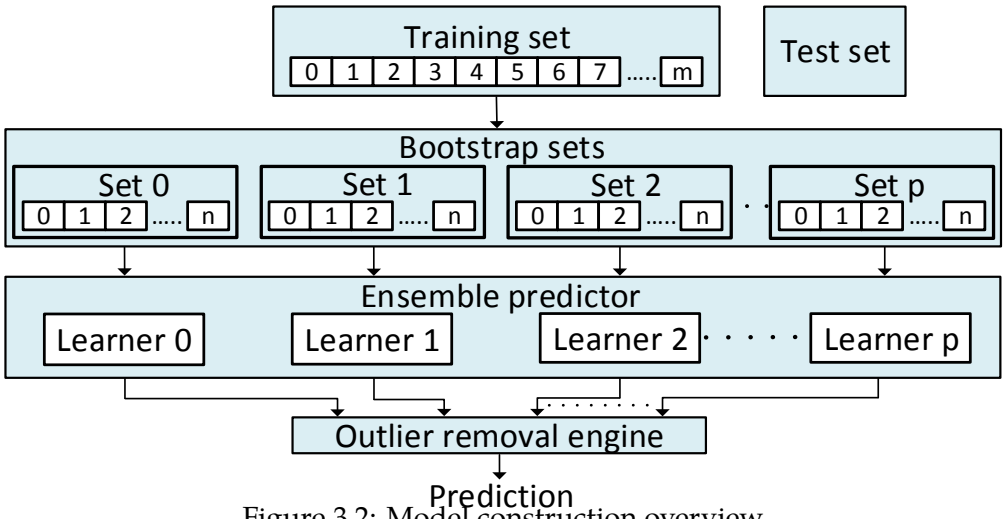


Figure 3.2: Model construction overview.

Level 2: Ensemble Prediction Figure 3.2 gives an overview of our ensemble prediction technique. We begin by randomly partitioning our dataset into two mutually-exclusive sets of train and test, where the test set is put aside for the final evaluation. We then generate p new training sets, each of which is generated by sampling m training examples drawn randomly *with replacement* from the original training set of m items. Such a sample is called a *bootsrtap sets*, and the technique is called ***b**ootstrap **agg**regat**ing** (or **bagging**)*. By sampling with replacement, we mean that examples can appear multiple times in each bootstrap set. On average, each set contains 63.2% unique examples of the original training set, that is $n \approx 0.63m$ [43]. We then construct p individual models[1] for p bootstrap replicates. For any new program, we would have p execution time predictions, from which we filter out the outliers, and then get the arithmetic mean of the result. Our outlier removal technique is simple - sort all the predictions in numerical order, find the median point and only pick the 30% of prediction instances above and below that point. Finally, we turn GPU execution time prediction into speedup prediction by dividing by the measured CPU time.

[1]In this Chapter, we use the terms 'base learner' and 'individual model' interchangeably, the latter being an informal but more intuitive term.

Approach	Description	Pros and cons
Simple linear regression	Consider all features and minimize for residual error	+ Simple − Too many features, too little training data − Residual error too high, poor accuracy
LASSO-1	Standard LASSO with all features	+ provides list of features to consider − By itself poor accuracy
LASSO-2	Use subset of features from LASSO-1 and use higher order powers for these features	+ Good for some test data − Too aggressive in eliminating features − Some applications end up having no useful representative feature
Exhaustive feature selection	Exhaustive feature selection, higher-order powers, all interactions, and minimize residual error	+ Excellent model for training data − Overfitting and poor accuracy for test data
Exhaustive feature selection and repeated random sub-sampling validation	Exhaustive feature selection, higher-order powers, all interactions, minimize cross-validation error	+ Good accuracy − Unstable − Long run-time (about 30 minutes)
Ensemble of step-wise regression learners	An ensemble of 100 individual models for different data subsets, allow all features and their pairwise interactions, report the average across the 60% of predictions around the median value	+ Good accuracy + Stable − Long run-time (about 30 minutes)

Table 3.3: Summary of approaches we explored.

3.3.3 Other Machine Learning Approaches

In this subsection, we describe in an incremental fashion all the approaches we explored but failed to generate an accurate results, until we found the ensemble of regression learners. At each step, we summarize the deficiencies we discovered and attempted to address in the next approach. Table 3.3 summarizes the various models we considered.

Simple linear regression model: In this model, we simply used standard linear regression based on all the features. The model was simple and ran very quickly (matter of seconds). The main drawback was its extremely poor accuracy. In general there was too little training data to fit a high quality model of 27 features.

Regularized regression using LASSO: We then used regularized regression in two steps: (LASSO-1) First we ran LASSO with all the features, which provided an initial model with better accuracy than simple linear regression. This model gave us the most important features. We then built a second model (LASSO-2) also using LASSO but including up to the 4th-order powers and interactions between these variables. We only included higher-order candidate features if their constituents appear in the first-order LASSO model. This is a common practice in machine learning to control the exponential growth of higher-order candidate features. We found that for some applications the representative features selected proved insufficient and resulted in overall poor accuracy. We concluded that standard regularized regression package eliminates too many features and the automatic selection of λ did not help.

Exhaustive feature selection: So in this next approach, we adapt our exhaustive feature selection technique, in order to control the number of features eliminated. We created all models with 4 features ($_{27}C_4$ = 17550 models) and included all their interacting features and up to 4th order powers[2]. We kept the number of features at 4 to avoid the over-fitting problem. We exhaustively search the space of all four models to find the model with low residual error. This approach had a over-fitting problem, resulted in poor accuracy on test data.

Exhaustive feature selection and repeated random sub-sampling: To avoid over-fitting, we used repeated random sub-sampling validation, where we randomly split our training set into mutually exclusive training subset and validation subset, in 100 different ways. For each such split, we fit a regression model to the training subset, and assessed its accuracy using the validation subset. Finally, we selected the model which has the lowest cross validation error. Although this model showed very good accuracy for our dataset, the accuracy was not very stable, i.e. a small modification in our dataset hurt the accuracy.

Eventually, we figured out that we need to construct different models for different parts of

[2]Machine learning practitioners advise us that up to 4th order is in general sufficient.

program space, find the most representative models for each program and get the average of their predictions, to ensure stability. This leads us to the ensemble prediction idea. We will discuss this insight in more detail in Chapter 5.

3.4 Summary

In this Chapter, we developed an automated performance prediction tool that can provide accurate estimates of GPU execution time for any CPU code *prior to developing the GPU code*. This work is built on two insights: i) Hardware characteristics and program properties dictate the execution time. ii) By examining a vast array of previously implemented GPU codes, along-with their CPU counterpart, we can use machine learning to discover the correlation between CPU program properties and GPU execution time.

The key contribution of this work is the observation that for any GPU platform, GPU execution time can be formulated as a mathematical function where fundamental microarchitecture-independent and architecture-independent program properties are *variables* and GPU hardware characteristics are *fixed coefficients*. Variables alone change from one application to another, and coefficients are fixed for all applications and change from one GPU hardware implementation to another. We are the first to observe this platform independent correlation.

4 | Static XAPP: Concept and Design

Contents

The dynamic XAPP framework, developed in Chapter 3, analyzes the CPU binaries using dynamic binary instrumentation to obtain their program properties. In this chapter, we introduce static XAPP, which is a novel sub-branch to the cross-architecture performance prediction paradigm: *Intermediate-representation (IR)-based* feature estimation. This approach avoids the slowdowns of binary instrumentation. However, it requires fundamental changes to our dynamic XAPP approach and infrastructure. This is because: (1) Statically determined features are different than dynamic

features. (2) It is hard to get the exact values statically for most/all of these features, but getting their binary values are easier. (3) The best machine learning techniques for discrete inputs are different than for continuous inputs. As we will show in Chapter 7, the consequence of these modifications is that static XAPP is better at predicting the speedup range.

This chapter makes the insightful observation that program properties obtainable with simple static analysis—without execution—are sufficiently explanatory to predict performance *range* across architectures. Regardless of whether programmers consider the execution-time and human-time overheads of existing tools a barrier for their usage, from an aesthetic and intellectual standpoint, the limit of static analysis for this problem is an open question. Furthermore, a source-code-based technique opens up the possibility of embedding such fast estimates into integrated development environments (IDEs), to interactively provide developers with feedback on potential performance improvements gained by porting fragments of their single-threaded code to a GPU. We anticipate that programmers will use static XAPP at the early stage of development to explore GPU speedup for different algorithms.

4.1 Overview

Problem statement GPU programming is challenging and time consuming. Dynamic XAPP incurs binary instrumentation overhead. We will show that statically-collected features from the intermediate representation (IR) of the program are sufficiently explanatory to predict the GPU speedup range.

Insight We investigate and propose program feature extraction via static analysis of CPU code, with the goal of predicting GPU performance using machine learning. In particular, we statically extract numerical features from programs. The numerical feature values are usually very inaccurate. We then use a machine-learning-based discretizer to turn inaccurate numerical features into accurate binary ones. Finally, we employ the *random forest classification* technique to predict the *speedup range*.

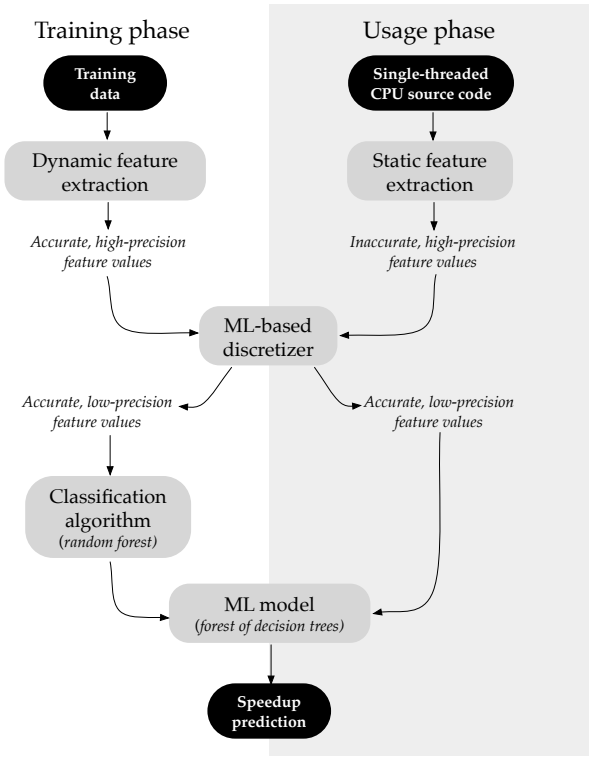


Figure 4.1: Overall flow

Two inter-related key insights enable us to use source-code analysis to obtain meaningful program properties. Figure 4.1 shows the overall flow of static XAPP.

While IR analysis can only provide approximate values of program features—since it lacks information on the influence of dynamic effects—we observe that the approximate values of features can be made accurate by reducing precision. In particular, we reduce the precision to just one bit—*high* or *low*. In practice, we observe that static analysis is 100% accurate in determining the binary values of the program properties we define. We have discovered a meaningful set of ten features that are sufficiently explanatory. We use machine learning here to determine where to place cut-points on any given feature to discretize it as high or low.

Second, we observe that low-precision inputs comprising *multiple* binary features can be combined to produce *higher-precision* output (speedup). In particular, by using a classification-based

machine learning technique, we can classify speedup into as many as 5 intervals (e.g. $[0, 3)$, $[3, 20)$, $[20, 50)$, $[50, 100)$, $[100, \infty)$). We can have users provide the speedup intervals, or can use machine-learning to determine the best output cutpoints to maximize accuracy.

Practical implementation We use a set of 147 single-threaded CPU programs (written in C or C++) that are already ported into GPU code (written in CUDA) to construct and evaluate our model. In an offline and one-time training phase (per GPU platform), we collect dynamic program properties from CPU code and GPU speedup to learn the correlation between features and speedup. To collect program properties for the training set, we re-purpose an existing dynamic binary instrumentation tool to measure accurate values of program properties. Using machine learning, we then discretize these values into accurate, low-precision binary values. We then compare these to our static-analysis estimated low-precision values. Our first quantitative result is that static analysis is 100% accurate. We feed this binary feature vector of the train-data corpus into a regression-tree classifier to predict the speedup range for different numbers of intervals (3, 4, 5, and 6 bins) with varying interval widths. For speedup, we measure CPU execution time and GPU execution time on the host CPU and target GPU platform. We also discretize speedup based on user preferences. Our prediction accuracy is 94% for the interesting speedup intervals of (1) $\textit{speedup} \leq 3$ and (2) $3 < \textit{speedup}$. Figure 6.2 shows that our model can accurately predict speedup for any 2, 3, or 4 arbitrary speedup intervals, while average accuracy remains above 70%

Contributions This chapter makes one broad intellectual contribution: We observe and demonstrate that statically determinable program properties are sufficient to predict GPU performance. Specifically, (1) we identify and precisely define 10 program properties that are accurately statically determinable and are sufficiently explanatory (from a machine learning perspective); (2) we determine a method for converting the approximate static estimates into an accurate low-precision value; and (3) we implement a machine-learning classification method that uses random forests to predict speedup range.

Statement type	Description
MEM	Memory loads and stores
ARITH	All arithmetic operations
MDIV $\subseteq$ ARITH	FP multiplication and division
SCOS $\subseteq$ ARITH	FP sine and cosine
ELOGF $\subseteq$ ARITH	FP logarithm and exponential
SQRT $\subseteq$ ARITH	FP square root
CTRL	Conditional control statements

Table 4.1: Program statements

#	Feature	Formal definition	Relevance of feature for GPU performance
1	Memory coalescence	$\sum_{s \in \text{MEM}, coalesced(s)} f_I(s) / \sum_{s \in \text{MEM}} f_I(s)$	Captures whether memory accesses are coalesced
2	Branch divergence	$\sum_{s \in \text{CTRL}, diverge(s)} f_I(s) / \sum_{s \in \text{CTRL}} f_I(s)$	Captures vulnerability to branch divergence
3	Kernel size (*ksize*)	$\sum_{s \in P} f_I(s)$	Captures whether kernel is embarrassingly-parallel
4	Available parallelism	$f_I(s)$ s.t. s is inner-most loop in PBAND	Captures GPU resource utilization
5	Arithmetic intensity	$\sum_{s \in \text{ARITH}} f_I(s) / \sum_{s \in \text{MEM}} f_I(s)$	Captures ability to hide memory latency
6	Control intensity	$\sum_{s \in \text{CTRL}} f_I(s) / ksize$	Captures whether kernel is control-intensive
7	Mul/div intensity	$\sum_{s \in \text{MDIV}} f_I(s) / ksize$	Exploits GPU's abundant mul/div units
8	Sin/cos intensity	$\sum_{s \in \text{SCOS}} f_I(s) / ksize$	Exploits GPU hardware support for SFU
9	Log/exp intensity	$\sum_{s \in \text{ELOGF}} f_I(s) / ksize$	Exploits GPU hardware support for SFU
10	Square root intensity	$\sum_{s \in \text{SQRT}} f_I(s) / ksize$	Exploits GPU hardware support for SFU

Table 4.2: Program features, their formal definition, and how they impact GPU speedup

4.2 Program Features

As mentioned before, the first step in formulating a machine learning problem is to capture an essential set of features required for characterizing a desired output—in our case, the speedup attained by porting CPU code to a GPU. In this section, we describe a generic CPU program model and an associated static analysis that computes a number of important program features for GPU speedup prediction.

4.2.1 Program Model

We will assume that we are given a sequential CPU program P in a standard representation (e.g., LLVM's intermediate representation). Program instructions are categorized as shown in Table 4.1.

We will use MEM to denote the set of all memory load and store instructions that appear in P. Similarly, we will use ARITH to denote arithmetic operations in P, and CTRL to denote conditional branches.

Given an expression e over program variables in P—e.g., an address—we will say that e is *loop invariant* with respect to a given loop if it does not depend on the loop's induction variable. Detecting loop-invariant expressions is performed with a standard static analysis. Loop-invariant expressions will be particularly important for the analysis of the branch divergence and non-coalesced memory accesses.

4.2.2 Program Features and Static Extraction

Given a program P, as defined above, we will assume that the developer has annotated the region of the code—the loop or loops—they wish to parallelize. We call this region the *parallel band* (PBAND). We refer to the rest of the code enclosed within the PBAND, as *kernel body* (KBODY), as it corresponds to the body of the GPU kernel that maps to a GPU thread. Consider, for example, the sample CPU code in Figure 4.2. Here, the developer has indicated that they would like to parallelize across the outermost loop only. Therefore, the region inside (line 3-14) maps to the thread body.

While our features are statically determinable, for the purposes of illustration, we will assume that we are given an input I of the program P. Using I, we can characterize the number of times an instruction s is executed as a function of I, which we call the *expected occurrence frequency* of s and denote by $f_I(s)$. Note that this function, f_I, can only be discovered dynamically. However, as we shall see, our approach is robust to the values of f_I and we can elide f_I computation, thus maintaining static extraction of all features.

The set of (numerical) features computed from P is formally defined and described in Table 4.2. In what follows, we provide a thorough exposition of these features and the rationale behind choosing them. We note that, while these features are numerical, they will be later *discretized* automatically by our machine learning algorithms.

```
1  #pragma parallel XAPP (1048576) //parallel band
2  for (i=0; i < num_elements; i++){
3    key=i; j=0;
4    if (key == tree[0]){
5      found++;
6      continue;
7    }
8          #pragma XAPP(16)
9    for(j=0; j<depth-1; j++){
10     j = (j*2) + 1 + (key > tree[j]);
11     if (key == tree[j]) {
12       found++;
13       break;
14     }
15   }
16 }
```

Figure 4.2: Example CPU code

Note here that the set of program features measured for static XAPP are different from the set of features obtained for dynamic XAPP, explained in Chapter 3. Dynamic XAPP features were collected using dynamic binary instrumentation and were capturing the dynamic behavior of the program. We will use a subset of dynamic XAPP features that are statically collectible. Although, we have to modify the feature definitions to make them statically collectible.

1. Memory coalescence The first feature, *memory coalescence*, is a high-impact feature on GPU speedup; it captures the possibility of global memory accesses to be coalesced. A non-coalesced memory access can reduce the bandwidth efficiency to as low as 1/32, which negatively affects the speedup [39]. Specifically, this feature characterizes the percentage of memory instructions in the KBODY that are considered *coalesced*. We weight each operation $s \in$ MEM by its occurrence frequency $f_I(s)$. Given a memory operation $s \in$ MEM, we consider *coalesced*(s) to be true *iff* one of the following holds: (1) Its memory index expression is loop-invariant with respect to all the loops within the PBAND. Intuitively, this means that all threads access the same memory location. (2) Its memory-index expression is loop-invariant with respect to all the loops within the PBAND, except

the innermost one. The innermost-loop induction variable should appear with a $multiplier \leq 1$ in the memory-index expression. Intuitively, this means that consecutive threads are accessing consecutive or the same memory locations.

In our running example in Figure 4.2, there are two memory operations: `tree[0]` is memory coalesced, as the memory index is loop-invariant; `tree[j]` is considered not memory coalesced, as the memory index, j, depends on i, the induction variable of the loop in PBAND.

2. Branch divergence Branch divergence is a measure of how effectively the parallel resources on the GPU are being utilized. Specifically, we characterize branch divergence as the percentage of conditional statements in the program that are considered *diverging*. We weigh each operation $s \in$ CTRL by its occurrence frequency $f_I(s)$. For a branch $s \in$ CTRL, we consider *diverge*(s) to be true *iff* at least one of the conditional expressions in s is not loop-invariant with respect to the parallel band loops. Intuitively, this means that the branch condition may differ in different threads, therefore potentially causing divergence.

3. Kernel size The *kernel size* (*ksize*) feature is the number of instructions in the KBODY of the given program, where each instruction is weighted by its occurrence frequency. This is used as an indication of the dynamic number of instructions to appear in the GPU kernel, and to enable computation of the *intensity* features described below. Generally, when the kernel size is very large, it suggests that there is a loop with data dependency across its iterations inside the KBODY, otherwise the loop should have moved into the PBAND. Therefore, the large kernel size indicates that the kernel is not embarrassingly-parallel.

4. Available parallelism The *available parallelism* feature is an approximation of the number of GPU threads. Specifically, available parallelism is approximated as the occurrence frequency of the inner-most loop in the parallel-band. In our running example, the parallel band is comprised of a single loop (the outer-most one), and therefore occurrence frequency of that loop provides an

indication of the number of GPU threads. Available parallelism indicates whether GPU resources are fully utilized.

5-10. Instruction intensities The lower part of Table 4.2 contains features that measure whether the CPU code, when ported to GPU, will exploit the strengths of GPUs. For instance, the *arithmetic intensity* feature is a measure of how well the arithmetic operations can hide memory latency, and is defined as the ratio of the number of arithmetic operations to the number of memory operations. To estimate the number of memory operations/arithmetic operations statically, we weigh each operation s by its occurrence frequency $f_I(s)$.

Similarly, other features in this category, measure of how effectively special function units on GPU are utilized. For instance, the ratio of the number of single-precision floating-point sin/cos (log/exp, sqrt) operations to the total number of instructions.

4.2.3 Expected Occurrence Frequency

The above feature extraction assumed the existence of a function f_I that specifies the expected occurrence frequency of each program instruction. While f_I is not statically determinable, we have empirically validated that our model is robust to changes in f_I. Specifically, the expected occurrence frequency of an instruction is a function of (1) loop-trip counts of loops enclosing the instruction, and (2) the probability of taking branches that lead execution to the instruction. In Chapter 6, we will show that speedup range prediction remains almost unchanged with varying the values of loop-trip counts and branch probabilities. We will show that majority vote on predictions gives an accurate estimate of speedup range.

4.3 Machine Learning Approach

In this Section, we discuss the details of our machine learning approach, including the preparation phase, model construction phase, the details of the training and test sets, and the software/hardware

platforms used for evaluation of our technique.

4.3.1 Preliminaries

A *feature vector* is the set of 10 program properties, outlined in Section 4.2, estimated per CPU code and presented in the form of a vector. The elements within the vectors are all binary values. Thus, the entire program space is characterized by $2^{10} = 1024$ feature vectors.

4.3.2 Preprocessing Steps

Recall that we favored static analysis over dynamic analysis to expedite the estimation process. However, this comes at the cost of reduced precision in feature estimation. Machine learning can learn from low-precision feature values as long as they are accurate. We discretize feature values, measured through static analysis, into two or three levels to increase their accuracy, and feed the discretized values of the features into our model. In what follows, we describe how we discretize feature values and speedup.

Discretize the input The input features or program properties are naturally continuous values. To improve the accuracy of the low-precision feature values, we discretize them into two levels (low or high) or three levels (low, medium, or high). We use the *equal frequency binning* algorithm to find the cutoffs that discretize the features [45] into two or three levels. This algorithm finds k cutpoints that divide the feature range into $k + 1$ intervals, such that each interval contains approximately the same number of values.

Discretize the output The output variable (speedup) is a continuous value that can span a wide range, from 0 to 1000s. From the developer's perspective, the decision to port a code to GPU rests on the achievable speedup range, and not the actual speedup value. However, different developers or development scenarios find different speedup ranges worthwhile for porting. This difference comes from variations in their CPU/GPU programming expertise, time availability, the capability

of their host CPU and target GPU, and their application domain. To this end, we allow the user to feed their speedup range of interest as an input into our model. For example, if a user is interested in dividing the speedup range into $[0, 3]$, $(3, 10]$, $(10, \inf)$, they would feed 3 and 10 into the model.

4.3.3 Model Construction

We employ a random forest classifier to construct our model. A random forest is an ensemble of many decision trees. We selected random forests as they do not overfit by design, so their accuracy can be generalized to unseen data [46]. Moreover, accuracy of random forests is competitive with other classification techniques, including support vector machines (SVM) and neural networks [47, 48]. Note here that the machine learning technique used for dynamic XAPP, which is designed for *exact* speedup prediction, is unsuitable for speedup *range* prediction. Dynamic XAPP uses regression as a base learner, whose objective is to minimize the distance in the continuous space, while static XAPP objective is to minimize the classification error in discretized space. We will analyze this in more detail in Chapter 7. We explain our model construction procedure which entails the details about the construction of a single binary decision tree and an overview of the random forest technique.

Binary Decision Tree This is a classification technique that predicts the class label (output) based on the input program properties (features). The outcome of the technique is a tree-like structure, where internal nodes represent binary tests on the input features (e.g, the feature value is low or high), the edges between the nodes represent the outcome of the test, and the leaf nodes represent the class labels. The paths from the root to leaves represent classification rules.

We use the *ID3 algorithm* to construct a decision tree. ID3 starts with zero features. It then finds the feature that splits the data into two most distinctive sets (referred to as *best split*), and adds that feature as a root node into the model. The root node divides the dataset into two partitions. ID3 recursively splits each partition into two partitions and finds their best splits, until a stopping criterion is met. A best split is a feature that maximizes the information gain, or most reduces the conditional entropy of the output for the datapoints in the given partition. Information gain is simply

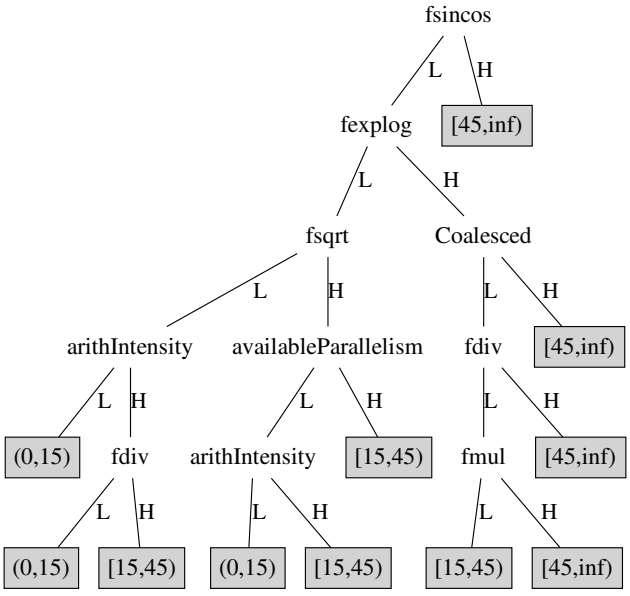


Figure 4.3: An instance of a learned tree

the difference between the entropy of the data before and after the split happens. Mathematically, entropy is defined as follows:

$$H_D(Y) = -\sum_{y \in values(Y)} P(y) \times log(P(y)) \tag{4.1}$$

where D is the dataset, Y is the set of all class labels, and $P(y)$ is the probability that a datapoint in D has the label y [49]. The minimum size of the leaf node (*nodesize*) is a parameter into our model. Empirically, we found $nodesize = 1$ generates models with good accuracy. Figure 4.3 shows an example binary decision tree.

Random Forest This is an ensemble of many decision trees, where each tree is constructed with a random subset of the features and a random subset of the training set (shown in Figure 4.4). Each tree is constructed as follows: First, sample N datapoints with replacement from the original training set of size N. Drawing with replacement means that a datapoint can be drawn multiple times. This is going to be the training subset for the growing tree. Second, randomly select s feature

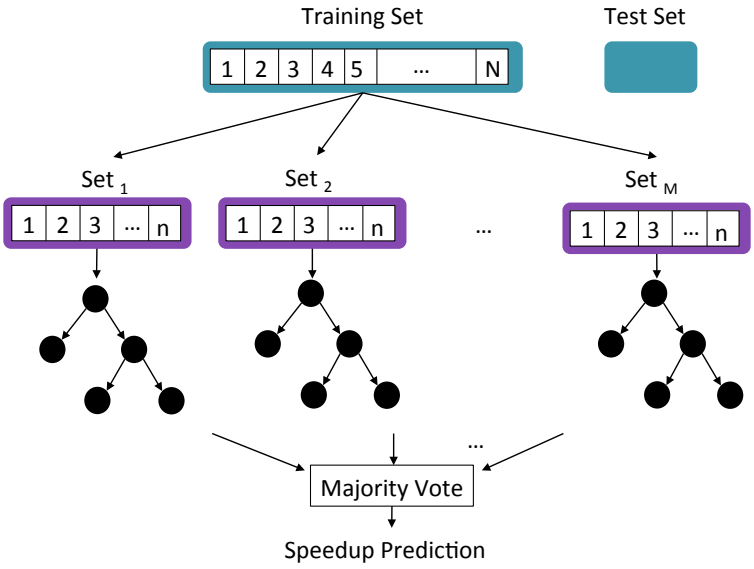


Figure 4.4: Model construction overview

split candidates from the feature set. The best split among these s features is used to split the node. The number of decision trees (M) and the number of features that are randomly sampled as candidates at each node (s) are parameters. Empirically, We found $M = 1000$ and $s = 3$ generates models with good accuracy. To classify a new program, we feed its feature vector to each of the trees in the forest. Each tree gives a classification, or votes for a speedup interval. The forest chooses the speedup interval having the most votes over all the trees in the forest [50].

4.3.4 Other Machine Learning Approaches

In this subsection, we describe in an incremental fashion all the approaches we explored but failed to generate an accurate results, until we found we need to discretize input features and output speedup and use the random forest technique. We first tried to apply the same machine learning approach we used for dynamic XAPP. Dynamic XAPP uses continuous input, generates continuous output and uses step-wise linear regression as a base learner. We explored all possible types of inputs and outputs, being continuous or discretized, alongside using linear regression. Here, we

briefly explain all scenarios and observed results.

Continuous Input, Continuous Output We used dynamic XAPP methodology to predict the exact speedup value, using the subset of features used for static XAPP. We ran this study using the accurate feature values (dynamically-collected) and observed 120% relative error, in average. Since the model accuracy was so low with even perfect information about the features, we concluded that accuracy would be even worse with inaccurate feature values. We will discuss this scenario in more detail in Chapter 7.

Continuous input, Discretized Output We also used dynamic XAPP methodology to predict speedup range. We used the same model as before but converted the final continuous speedup predictions into ranges and observed 88% accuracy for speedup cutpoint at 3. We will discuss this scenario in more detail in Chapter 7.

Discretized Input, Continuous Output We also slightly modified dynamic XAPP, to use discretized input and predict continuous speedup value, and observed 94% relative error, in average. Much like the previous scenario, we had to replace the step-wise linear regression base learners with regression tree, since all input features were categorical.

Discretized Input, Discretized Output We explored different classification techniques, particularly neural network and support vector machine (SVM) for speedup range prediction. They showed 59% and 60% accuracy, respectively, when predicting three speedup intervals (cutoffs at 4 and 20), while random forest achieved 82% for the same set of cutoffs.

4.4 Summary

In this Chapter, we developed a new speedup prediction technique that relies only on source code to advance the state-of-art. It has been believed that program properties needed for predicting GPU

prediction must necessarily be obtained from the dynamic execution of the program.

Static XAPP makes a fundamental intellectual contribution in demonstrating that statically determinable program properties are sufficiently explanatory for developing a machine-learning based speedup predictor. Based on train-data from commonly used GPU benchmarks, we have developed an effective predictor with high accuracy. The implications and usage of our tool are multifold. It can be embedded into modern interactive development environments like Eclipse, where programmers can simply highlight a region of code and be immediately presented with the prediction. Further visualization extensions are possible, like showing the prediction speedup of every function, loop, etc. Another use case would be to include it as part of the future compilation techniques to target only the promising code regions for heavy-weight auto-compilation analysis.

5 | Performance Evaluation: Dynamic XAPP

Contents

This chapter evaluates the performance of the dynamic XAPP model presented in Chapter 3. Section 5.1 presents the experimental methodology. Section 5.2 presents the results for model accuracy and robustness. It also discusses why we need an ensemble solution, and provides an insight into our ensemble results. Section 5.3 presents an end-to-end case study, which explains how our tool performs in the wild.

5.1 Methodology

In this section, we explain the infrastructure which we use to implement and evaluate our model. We first describe our dataset selection process. Then, we explain our evaluation methodology, where we describe how we characterize error and how we divide our dataset into a train and test set. Finally, we describe our measurement strategy on real hardware, and how the models are implemented.

5.1.1 Training Data and Test Data

We examined many prevalent benchmarks suites, namely Lonestar [51], Parsec subset [52], Parboil [53], Rodinia [54], NAS subset [55, 56] and some in-house benchmarks based on the throughput kernels [57]. We also looked at various source code repositories like `https://hpcforge.org/`. Across benchmarks, we consider each kernel as a piece of training data, since kernels within a benchmark could have very different behavior. The criteria for something to serve as train data was the following: i) It must contain corresponding CPU source code written in C or C++; ii) The algorithm used in the GPU and CPU case should be similar. Recall that we consider two algorithms dissimilar, if their computational complexity mismatches. That said, common GPU optimizations (such as loop reordering, loop blocking and overlapped tiling) that change the order of accesses to global memory, but do not change the number of global memory accesses, do not make two algorithms dissimilar, and therefore we allow them in our dataset. For instance, matrix multiplica-

tion implementation on GPU requires data layout transformation to make the best use of shared memory. We consider these modifications non-algorithmic and can serve as a valid candidate in our training set. iii) The CPU source code should have well defined regions that map to kernels on the GPU - to avoid human error we discarded programs where this was not obvious or clear.

We also developed our own low-speedup microbenchmarks to include some obviously ill-suited code for GPU in our dataset. At this stage, we obtained a total of 42 kernels (original kernels).

We managed to increase the number of datapoints by 80 using a combination of input modification and code modification (derived kernels). (1) Modifying the input parameters of a program generally changes its speedup and feature vector, and hence it can be considered as a new data point. (2) Based on reading the description of the Lonestar kernels, we defined related problems. We then manually developed alternate CPU and GPU implementations, ensuring the above three criteria were adhered to. The point of interest here is that this provides data points with different speedup and different features. For example, XORing an existing conditional statement with random values can change the code's branching behavior.

Our final dataset contains 122 datapoints, which are well spread across the speedup range as follows: 24 in (1-4], 22 in (4-10], 25 in (10,25], 36 in (25-100], and 16 in (100-∞). This shows we have representative data points for the interesting parts of the speedup space.

To evaluate the accuracy of our model in the end, we need a *test set* which is not used during model construction. The criteria for something to serve as a test set was the following; i) It should not appear in the train set. ii) It should be a real-world benchmark from the publicly-available benchmark suite. We made an exception on low-speedup benchmarks and allow our microbenchmarks appear in the test set, since there are few low-speedup kernels in available benchmark suites. iii) It should be unique in that it does not have a modified input or modified kernel counterpart in the train set. Only 24 kernels satisfy these conditions. We refer to this subset as QFT (Qualified for Test). The speedup span for QFT is 0.8 to 109. We always select our test set (10 datapoints) randomly from QFT.

	Platform 1	**Platform 2**
Microarchitecture	Maxwell	Kepler
GPU model	GTX 750	GTX 660 Ti
# SMs	7	14
# cores per SM	192	192
Core freq.	1.32 GHz	0.98 GHz
Memory freq.	2.5 GHz	3 GHz
CPU model:	Intel Xeon Processor E3-1241 v3 (8M Cache, 3.50 GHz)	

Table 5.1: Hardware platforms pecifications.

5.1.2 Hardware Platforms and Software Infrastructure

To demonstrate robustness, we considered two different hardware platforms (summarized in Table 5.1) for which we automatically predict speedups. All the explanations in this chapter are for Platform-1, and we use Platform-2 to test the *Hardware generality* property. We used MICA [58] and Pin [59] to obtain program properties. The tools that we wrote for *Lbdiv* and others (highlighted in grey in Table 3.1) are fairly straightforward and are hence not described in further detail. We manually examined each benchmark, identified the CPU code that corresponds to each GPU kernel in an application and added instrumentation hooks to collect data only for those regions. For implementing the regression model itself, we used the R package [60]. To obtain the program properties i.e. features, we executed MICA or PIN on the training and test data. To obtain the execution time or speedup, i.e. the output response, we measured execution time using performance counters.

5.2 Results and Analysis

Recall that we are using an ensemble technique. Our ensemble is a set of 100 individual models trained on 100 different subsets of the training set, whose individual predictions are aggregated into one prediction. The aggregation process selects 60 of the most similar predictions and get their average.

Kernel	Suite	Actual Speedup	Predicted Speedup	Relative Error%
$\mu 3$	μbench	1.30	1.29	0.5
srad1_4	rodinia	3.70	3.63	1.9
ftv6	nas	6.20	6.63	7.0
bkprp2	rodinia	10.0	9.43	5.7
ftv2	nas	10.4	14.7	41
cfd2	rodinia	23.3	28.4	21.9
srad1_3	rodinia	34.4	15.6	54.6
nn1	rodinia	39.7	23.3	41.4
srad1_1	rodinia	108	76.4	29.8
srad1_5	rodinia	109	87.3	20.0
Average				22.4
Gmean				11.6

Table 5.2: Accuracy of a representative ensemble model.

In the next subsections, we first show the accuracy of our model on *one* test set. We then show our model is robust by presenting its accuracy on *100* different test sets. This is done by going back to our 122-datapoint original dataset, randomly selecting 10 datapoints from QFT, and using the rest for training, and repeat this process 100 times. We also present the accuracy of our technique for low-speedup applications. Next, we discuss why we need to use an ensemble solution, and provide insights into our ensemble result. We then compare our solution against existing solutions, in terms of the metrics introduced in Chapter 1. Finally, we list the limitations and extensions of our tool.

5.2.1 Accuracy

Summary: Table 5.2 shows the accuracy of our tool for 10 kernels randomly selected from QFT. The average and geometric mean of the absolute values of the relative errors is 22.4% and 11.6%, respectively; Overall, dynamic XAPP is accurate.

To study the accuracy of a model, the common practice is to use the model to predict the output for a set of diverse datapoints that never appeared in the training process, measure the prediction

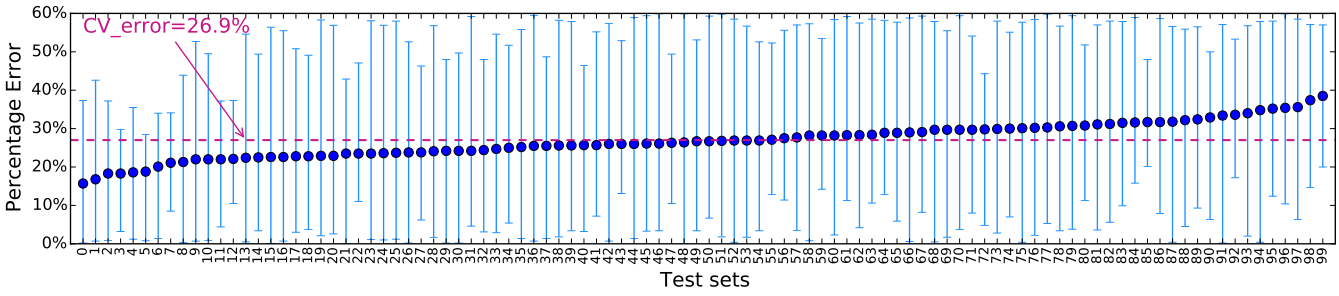


Figure 5.1: Ensemble technique accuracy across 100 different sets of test and train.

accuracy for each datapoint and report the average accuracy across these datapoints as the model accuracy. We use the absolute value of the relative error $((abs(Measured - Predicted)/Measured) * 100)$ to evaluate the accuracy for each datapoint.

Table 5.2 shows the accuracy of our model on a set of 10 datapoints (programs), randomly selected from QFT. We can see that our model is able to predict speedup accurately for a diverse set of applications (speedup span of 1.3 to 109), and achieve an average accuracy (relative error) of 22.4% and geometric mean of 11.6%, with minimum error of 0.5% and maximum error of 54.6%.

5.2.2 Robustness

Summary: Figure 5.1 shows the accuracy of our ensemble technique for 100 different test sets. Across these test sets, it maintains an average error (CV_{error}) of 26.9%. Overall, dynamic XAPP is very robust.

A machine learning technique is robust, if any given set of test or train generates similar models with similar accuracy [61]. To evaluate the robustness of our technique, we draw 100 different pairs of test and train, and construct 100 different ensemble models[1] , as outlined in Subsection 3.3.2. Figure 5.1 shows the average, minimum and maximum accuracy (relative error) of each ensemble model evaluated on its test set (of 10 datapoints each). The models are sorted along the X-axis based on their average error. Across 100 different test sets, the minimum, average and maximum relative error is 15%, 26.9%, and 46%. If we allow non-QFT datapoints to appear in the test set,

[1]It is a coincidence that the number of ensemble models is the same as the number of individual models within each ensemble.

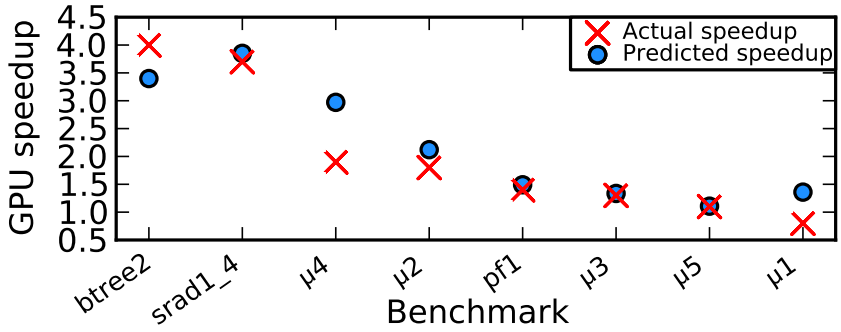


Figure 5.2: Low-speedup prediction accuracy.

accuracy would be even higher - minimum, average and maximum error of 10%, 23.5%, and 36%. The average error across 100 different test sets is referred to as cross-validation error (CV_{error}). This figure also shows that in the majority of test sets, there is at least one datapoint which has a prediction error of above 50%. These are usually low-speedup applications with high sensitivity to small error. An example would be predicting $\mu 1$ as 1.25, when the actual speedup was 0.8, as this will be regarded as 56% relative error, but is still a reasonable estimation of low-speedup. Next, we show our accuracy for all low-speedup applications that appeared across different test sets.

Low-Speedup Applications Users are often interested in knowing whether it is worthwhile to port their code into another architecture, given the time cost and performance benefits. We consider applications with a GPU speedup of less than 4 against a **single-threaded** CPU implementation to be *low-speedup*, that is, not worth the porting effort. Being able to classify applications into low and high speedups is perhaps the most important facet of the model. Figure 5.2 shows the actual and predicted speedup for all of the low-speedup kernels in QFT. As shown, we always predict the range correctly, and are often close to the actual value.

QFT Applications Figure 5.3 shows the average accuracy results for each QFT datapoint that appeared across the test sets of all 100 ensemble models. As shown, we always predict the range

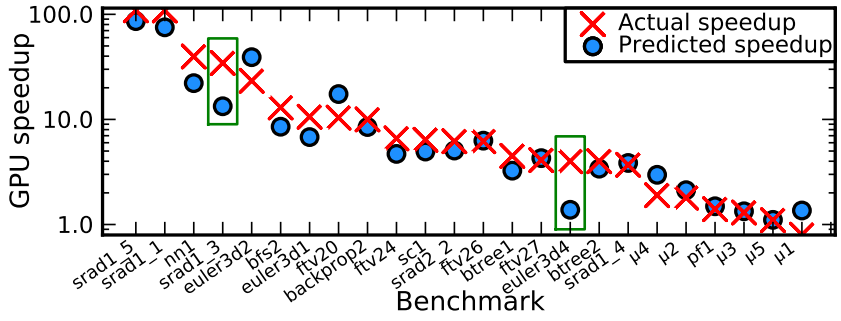


Figure 5.3: Accuracy results across all QFT kernels.

correctly, and are often close to the actual value.

5.2.3 Why Adaptive Ensemble Solution?

Summary: There are many good individual models as well as many bad individual models. Half of the models that are considered good for one application are usually bad for another application. We need an adaptive solution to automatically separate good individual models from bad individual models, per application.

Recall that one ensemble model is a collection of 100 individual models trained on 100 different subsets of the training set. For a given program, we observed that there are good individual models and bad individual models, which provide a wide range of predictions, with varying degrees of accuracy. Figure 5.4(a) shows speedup predictions for $bkprop2$, varying from 1 to 40, depending on which individual model is picked. We also observed that what we identify as a good or bad individual model depends on the application. Figure 5.4(b) shows speedup predictions for another program, $cfd2$. If we identify the 60 individual models around the model with median prediction (between the two vertical dashed lines) as good individual models and the ones outside this range as bad individual models, we can see that 24 models that are bad for $bkprop2$ are actually good for $cfd2$. We call this phenomenon model disagreement, and we show that this is prevalent between any two programs;

Table 5.3 shows model disagreement across the 10 different programs that appeared in Table 5.2,

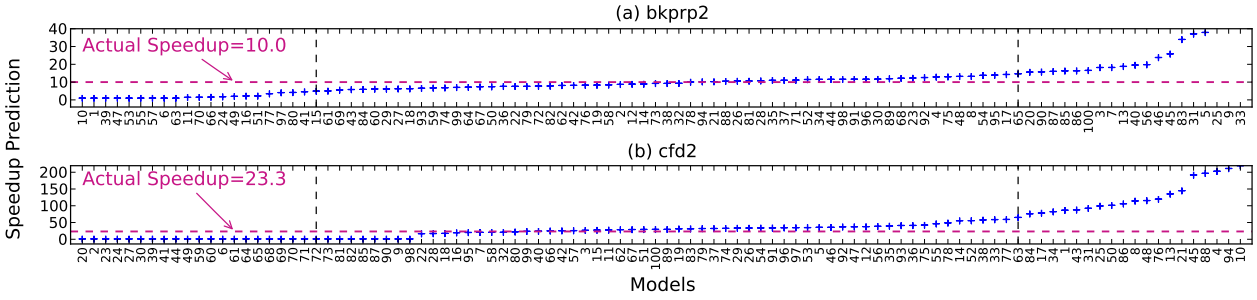


Figure 5.4: High sensitivity of single learners to the choice of train and how ensemble and outlier removal can fix it.

	a	b	c	d	e	f	g	h	i	j
a	0									
b	24	0								
c	25	28	0							
d	17	28	23	0						
e	23	22	27	25	0					
f	26	24	24	20	25	0				
g	28	19	28	23	18	24	0			
h	26	24	28	22	23	20	21	0		
i	25	24	28	25	24	26	20	23	0	
j	28	18	27	24	19	24	5	20	21	0

Table 5.3: Model disagreement Matrix.

now labeled a through j. The value at row **m** and column **n** shows the number of models that are good for program m but bad for program n. By definition, this Table is symmetric. We can see that almost half of the models that are good for one program are actually bad for another program. To quantify this in terms of error, we can consider a simple example. If we use j's 60 best models to predict performance for programs a through j, the accuracy would have been 23%, 127%, 237%, 38%, 31%, 14%, 28%, 37%, and 45%, in order. These error numbers show that good models for j are among the very bad models for b, c and i. Therefore, we need an adaptive ensemble technique that selects different models for each test point.

Another observation that we can make from Figure 5.4 is that the number of good models is significantly more than the number of bad models. Therefore, if we can automatically detect the good models and drop the bad models, then the average prediction across the good models would

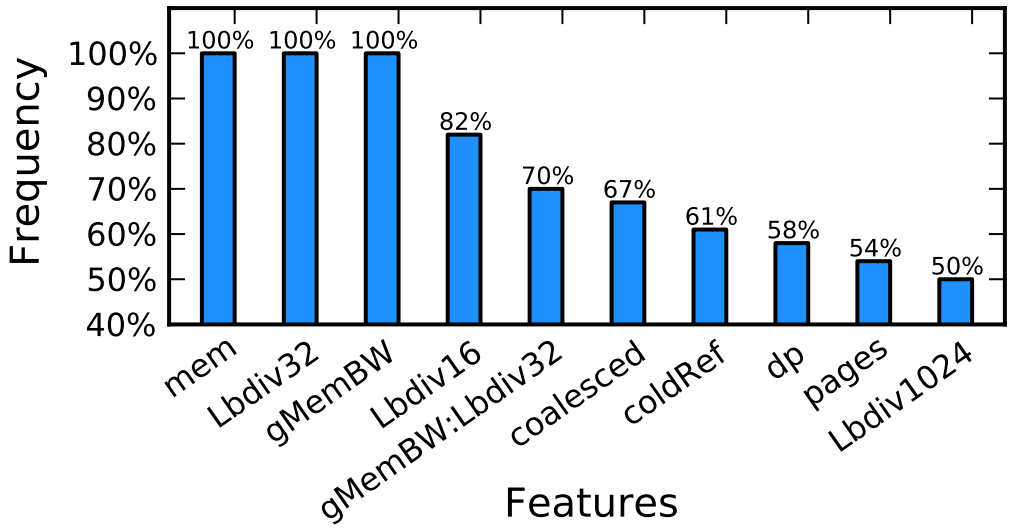


Figure 5.5: Highly correlated features with GPU execution time.

be a good indicator of the actual speedup. We refer to this step as outlier removal analysis, and we use a simple heuristic - for each application, we simply select 60% of the most similar models in terms of predictions. We refer to this percentage as the inclusion ratio, and we study how the inclusion ratio affects the overall accuracy. By sweeping the inclusion ration from 0% (using the median value as speedup) to 100% (including all predictions) by steps of 20, the accuracy (CV_{error}) changes as follows: 26.0% at 0%, 25.6% at 20%, 24.2% at 40%, 22.4% at 60%, 24.0% at 80% and 49.3% at 100%.

5.2.4 Model Interpretation

Summary: Figure 5.5 shows the top 10 most frequent features that appeared across 100 individual models, used in the construction of an ensemble. All features are intuitively correlated with GPU execution time.

Ensemble methods are popular ML techniques that boost the accuracy and stability of the base learner at the cost of comprehensibility [62, 63, 64, 65]. The result of our ensemble technique is 100 individual models of 17 features each, each of which explains part of the feature space, whose predictions are aggregated into one prediction after outlier removal analysis. The large number

of individual models and the adaptive outlier removal analysis that changes which models are selected from one benchmark to another, makes the ensemble outcome hard to interpret. Increasing the comprehensibility of the ensemble model comes at the cost of reduced accuracy ($\sim 40\%$ drop) [63, 64, 65]. Moreover, our main goal is to provide a tool with high predictive accuracy, capturing correlation and not necessarily causation.

To gain insight into the final complex model, we looked into the set of all features that appear across all individual models and measured the frequency of their occurrence across all models. Figure 5.5 shows the top 10 most frequent feature combinations. Of all the possible feature combinations (27 single features + $C(27, 2)$ pairwise features = 378), 143 unique features appeared across all models. The correlation of GPU execution time with memory ratio (mem), branch divergence ratio ($Lbdiv$), effective memory throughput ($gMemBW$ and $coallesced$), streaming/non-streaming behavior ($coldRef$ and $pages$) and dominance of double-precision floating-point arithmetic vs. integer or single-precision floating-point arithmetic (dp) is intuitive. The only unintuitive feature was **gMemBW** $\times$ **Lbdiv32**, which appeared across 70 models (with negative sign). This captures how the perfect memory coalescing (high $gMemBW$) can cancel out the increasing impact of high branch divergence (high $Lbdiv32$) on execution time.

5.2.5 Other Metrics

We now evaluate our tool in terms of the other four properties introduced in Section 1.6.

Programmer Usability indicates how much programmer involvement is required to make a CPU-based GPU speedup prediction. While some analytical techniques require GPU code to estimate program characteristics, others require extensive source code modification or GPU code sketches. We deem these techniques to have low and medium usability, respectively. Ones that can work with just the single-threaded CPU implementation have high usability. In our methodology, a user only needs to tag her regions of interest. The entire process is automated, hence dynamic XAPP has high usability.

Application Generality indicates if the technique can target *any* application with *any* level of complexity. There is nothing inherent in our machine learning approach that makes it incapable of predicting certain application types. We have a wide range of application in our dataset, from non-amenable to GPU, to irregular, to highly regular (speedup span of 0.8 to 321). Hence, we claim dynamic XAPP has high application generality.

Hardware Generality refers to whether the technique can easily adapt to various GPU hardware platforms. We use two different GPU cards with different micro-architectures as outlined in Table 5.1. The CV_{error} is 27% and 36% on platform 1 and 2, respectively.

Speed refers to the time needed by the tool to make a prediction. Our tool's runtime overhead can be categorized into two parts. (1) One-time Overhead: Measuring platform-independent program features for the train set needs to be done only once (by us) and is provided with dynamic XAPP. Users must obtain the GPU execution time for all datapoints in the train set for each platform of interest. This requires about 30 minutes. Model construction, a one-time occurrence per GPU platform, takes about 30 minutes.

(2) Recurring Ovehread: The user needs to gather features for the candidate program. This takes seconds to minutes — the instrumentation run introduces a $10\times$ to $20\times$ slowdown to native execution. Speedup projection completes in milliseconds — it is a matter of computing the function obtained in the previous phase.

5.2.6 Limitations and Extensions

Our current model cannot capture the impact of texture memory and constant memory. However, this is not a fundamental limitation of the technique, and is more a limitation of the small dataset. Our original dataset had only 5 kernels which use texture memory and/or constant memory. This issue can be resolved by adding more kernels with texture memory or constant memory to our training set.

Good for GPU	Easy for Human	Dynamic XAPP prediction	Prediction space	
No	Yes	No	TN	CS1
No	No	No	TN	CS2,K2
No	Yes	Yes	FP	-
No	No	Yes	FP	-
Yes	Yes	Yes	TP	CS2,K3
Yes	No	Yes	TP	CS3
Yes	Yes	No	FN	-
Yes	No	No	FN	CS2,K1

Table 5.4: Dynamic XAPP prediction space. The last column shows example code in case-studies in Figure 5.6.

5.3 End to End Case Studies

We now describe some end-to-end case studies that explain how our tool could perform in the wild.

5.3.1 Is Dynamic XAPP's Speedup Recommendation Always Correct?

Our test data shows impressive accuracy and range match on all test cases. But a natural question is whether dynamic XAPP is always correct. We consider this both from software development terms and from machine learning terms. From a software development perspective, we consider whether a piece of code is easy for a human to predict correctly or not (this is subjective of course and depends on programmer expertise etc.). The two other variables are whether or not the code is good for GPU (provides appreciable speedup), and then whether it is true positive (TP), true negative (TN), false positive (FP) or false negative (FN) from machine learning terms. If the prediction is in the right level, we deem it true positive/negative, else deem it false positive/negative. Table 5.4 shows the entire space.

We took three CPU applications for which optimized GPU code already exist and compared the measured speedup to the predicted speedup. Figure 5.6 shows the interesting regions of these CPU codes. The boxes indicate percentage execution based on profiling information for that region of

```
for(j = 0; j<Nparticles; j++){
   int index = -1;
   for(int x = 0; x<Nparticles; x++){
      if(CDF[x] >= u[j]){
         index = x; break;
      }
   }
   if(index == -1)  i = Nparticles - 1;
   else  i = index;
   if(i == -1)  i = Nparticles-1;
   xj[j] = arrayX[i]; yj[j] = arrayY[i];
}
```

Predicted	Measured	% time
1.93	1.96	73.9%

(a) Case study I

K0

```
for (i=0; i<Ne; i++)
    image2[i] = expf(image[i]/255);
```

Predicted	Measured	% time
76.4	108	0.3%

....

K1

```
    for (j=0; j<Nc; j++){
        for (i=0; i<Nr; i++){
            k = i + Nr*j; Jc = image[k];
            dN[k] = image[iN[i] + Nr*j] - Jc;
            dS[k] = image[iS[i] + Nr*j] - Jc;
            dW[k] = image[i + Nr*jW[j]] - Jc;
            dE[k] = image[i + Nr*jE[j]] - Jc;
            G2 = (dN[k]*dN[k] + dS[k]*dS[k]
                 + dW[k]*dW[k] + dE[k]*dE[k]) / (Jc*Jc);
            L = (dN[k] + dS[k] + dW[k] + dE[k]) / Jc;
            num = (0.5*G2) - ((1.0/16.0)*(L*L));
            den = 1 + (.25*L);  qsqr = num/(den*den);
            den = (qsqr-q0sqr) / (q0sqr * (1+q0sqr));
            c[k] = 1.0 / (1.0+den);
            if (c[k] < 0) c[k] = 0;
            else if (c[k] > 1) c[k] = 1;
        }
    }
```

Predicted	Measured	% time
15.6	34.4	88%

....

K2

```
    for (j=0; j<Nc; j++){
        for (i=0; i<Nr; i++){
            k = i + Nr*j;
            cN = c[k];  cS = c[iS[i] + Nr*j];
            cW = c[k];  cE = c[i + Nr*jE[j]];
            D = cN*dN[k] + cS*dS[k] + cW*dW[k] + cE*dE[k];
            image[k] = image[k] + 0.25*lambda*D;
        }
    }
}
```

Predicted	Measured	% time
3.63	3.7	8.8%

....

K3

```
for (i=0; i<Ne; i++)
    image[i] = logf(image[i])*255;
```

Predicted	Measured	% time
87.34	109	0.4%

(b) Case study II

```
for(int x = 0; x < w; x++){
    cameraX = 2 * x / float(w) - 1;
    rayDirX = dirX + planeX * cameraX;
    rayDirY = dirY + planeY * cameraX;
    float deltaDistX = sqrtf(1+(rayDirY*rayDirY)/(rayDirX*rayDirX));
    float deltaDistY = sqrtf(1+(rayDirX*rayDirX)/(rayDirY*rayDirY));
    hit = 0;
    if (rayDirX < 0){
         stepX = -1;
         sideDistX = (rayPosX - mapX) * deltaDistX;
    }else{
        stepX = 1;
        sideDistX = (mapX + 1.0 - rayPosX) * deltaDistX;
    }
    if (rayDirY < 0){
        stepY = -1;  sideDistY = (rayPosY - mapY) * deltaDistY;
    }else{
        stepY = 1;  sideDistY = (mapY + 1.0 - rayPosY) * deltaDistY;
    }
    while (hit == 0){
        if (sideDistX < sideDistY){
            sideDistX += deltaDistX;  mapX += stepX;  side = 0;
        }else{
            sideDistY += deltaDistY;  mapY += stepY;  side = 1;
        }
        if (worldMap[mapX][mapY] > 0) hit = 1;
    }
    if (side == 0)
        perpWallDist = fabs((mapX-rayPosX+(1-stepX)/2)/rayDirX);
    else perpWallDist = fabs((mapY-rayPosY+(1-stepY)/2)/rayDirY);
    int lineHeight = abs(int(h / perpWallDist));
    int drawStart = -lineHeight / 2 + h / 2;
    if(drawStart < 0) drawStart = 0;
    int drawEnd = lineHeight / 2 + h / 2;
    if(drawEnd >= h) drawEnd = h - 1;
}
```

Predicted	Measured	% time
21.29	21.1	99%

(c) Case study III

Figure 5.6: Case study kernel regions

the CPU code. These applications were picked intentionally to specifically highlight that our tool is *not* perfect. Dynamic XAPP is not meant to be used as a black box, nor is its output to be treated as definitive. We note here that we went out of our way to obtain these examples - some of these are from the train data set. We resorted to train data, since we did not find these combinations with easy to explain code in our test data. In the Table, an empty cell indicates that we could not get an easily explainable example. We emphasize this spread and behavior is the **uncommon** case for dynamic XAPP, but is possible.

In summary, our tool can have false positives and false negatives and cases where it could be easy for a human to estimate. We recommend using dynamic XAPP as an adviser to get an estimate, but ultimately users should pay attention. As we get more training data, dynamic XAPP's accuracy should improve even further. We discuss each of the case studies in detail below.

CS1: Bad for GPU, Easy for Human, True Nagative In this example, the code consists of a number of conditional operations, data dependent *for loops* and conditional break statements. The structure of this kernel makes it easy for a human to predict that it is a bad fit for GPU, and dynamic XAPP corroborates this.

CS2,K1: Good for GPU, Hard for Human, False Negative In this case study, the code contains a number of regular memory accesses, computations that have hardware support on the GPU and memory accesses that are heavily data dependent, making for an awkward combination of features. This is deemed hard for a human since it would be difficult to gain an understanding of the memory access pattern and consequently GPU performance through visual inspection. Dynamic XAPP predicts a speedup of 15.6 while the measured speedup is 34.4. Even though dynamic XAPP predicts correctly, we treat the under-prediction in this case as an example of a false negative to contrast with the next case (CS2,K2).

CS2,K2: Bad for GPU, Hard for Human, True Negative Similar to CS2,K1, this code contains a mixture of features that could be detrimental or beneficial to GPU execution time. However, unlike

CS2,K1, the measured speedup is quite small and dynamic XAPP predicts correctly. This case study shows that dynamic XAPP can predict the speedup of programs that might appear ambiguous to humans.

CS2,K3: Good for GPU, Easy for Human, True Positive This kernel is extremely simple and uses the `logf` function and hence should have a good speedup. Dynamic XAPP predicts correctly and this is easy for humans as well.

CS3: Good for GPU, Hard for Human, True Positive This program contains one dominant kernel region that contains a number of control flow statements, as well as a data dependent loop, creating opportunities for divergence on a GPU. Dynamic XAPP predicts a relatively high value of speedup (21.29) that almost seems counter-intuitive. However, the measured value of speedup (21.1) goes against our intuition and is closer to the predicted value. This case study shows that our tool can predict the speedup of programs that are hard for humans and might even appear to be a poor fit for GPUs. Here, the two `sqrt` and heavy use of division and multiplication make this code favorable for GPU.

5.3.2 Using Dynamic XAPP

Programmers are often times tasked with porting a CPU code to a GPU platform. In this scenario, dynamic XAPP combined with gprof (which is part of our packaged tool) can serve as a push button tool for determining what pieces of code to target. To demonstrate this, we took a CPU code with many kernels. We ran gprof on it and determined the top functions (this gives % breakdown when running on CPU). We then demarcated them as regions and obtained dynamic XAPP's predictions for those kernels. The results are shown in Figure 5.6, case study 2. Dynamic XAPP correctly predicted the speedup for all kernels, and it was also close for the dominant kernel. A programmer can use this information to then focus her efforts on that function first. If Kernel-2 had ended up being the dominant kernel according to gprof, it indicates the programmer should develop a

new algorithm. These speedups can be combined with Amdhal's law to project full application speedups, and complement tools like Kremlin [66].

5.4 Summary

In this Chapter, we evaluated the accuracy and robustness of our dynamic XAPP framework by presenting its accuracy over 100 different test sets. Overall, dynamic XAPP is accurate and robust at predicting GPU speedup from CPU code. Our tool showed 26.9% cross-validation error, where error is quantified as the average of the absolute value of the relative errors. We also analyzed the accuracy of our technique across the hard-to-predict applications (low-speedup applications), and provided a detailed analysis of why ensemble prediction is required and how our heuristic of simple adaptive selection of majority prediction works. We also showed that our ensemble model is insightful.

6 | Performance Evaluation: Static XAPP

Contents

6.1 Methodology

In this section, we explain the infrastructure which we use to implement and evaluate our model. We first describe our dataset selection process. Then, we explain our evaluation methodology. Finally,

we describe our measurement strategy on real hardware, and how the models are implemented.

6.1.1 Training and Testing

Training We collected our training set from widely-known benchmark suites, including Lonestar [51], Rodinia [54], and NAS [55, 56]. We followed the same approach we used for dynamic XAPP to construct our training set. We also developed extra microbenchmarks with different types of transcendental operations to capture the impact of transcendental operations. Table 6.1 shows our entire dataset. The naming convention for our benchmarks are as follows: *name-kernelNum-inputNum-suite*, where *name* is the benchmark name, *kernelNum* shows the kernel number inside the code, *inputNum* refers to the input number being used, and *suite* shows the benchmark suite that the kernel belongs to. *rd*, *ls*, μ refer to *Rodinia, Lonestar, microbenchmarks*; *others* refers to the NAS and in-house benchmarks. Some of the benchmarks have mvN in their name. This shows that the benchmark is a "derived" kernel and N shows the version number.

Testing We use *leave-one-out cross-validation* (LOOCV), a variation of cross-validation, to evaluate the accuracy of our technique. LOOCV is a widely-used evaluation technique, typically used in the analysis of small datasets. LOOCV works as follows: (1) Partition the dataset with N kernels in N different ways into a test set of one kernel and a training set of $N - 1$ kernels. (2) For each partition, construct the model on the training set and evaluate the accuracy on the test set, i.e check if the test kernel is correctly classified. (4) Use the ratio of the correctly classified kernels to the total number of kernels, N, as the measure of the overall accuracy [67].

This methodology is slightly different than what we used in Chapter 5. For dynamic XAPP, we use leave-10-out cross validation and only among QFT kernels, because it is very expensive and time-consuming to perform LOOCV for dynamic XAPP: It takes about 30 minutes to construct one ensemble model for dynamic XAPP, while it takes 30 seconds to construct a random forest model for static XAPP.

6.1.2 Hardware Platforms and Software Infrastructure

To study the robustness of our technique across different platforms, we constructed our model across two different GPU platforms with different microarchitectures (specifications in Chapter 5, Table 5.1). We use MICA [58] and Pin [59] to obtain the program properties for datapoints in our training set. We manually examined each benchmark, identified the CPU code that corresponds to each GPU kernel in an application and added instrumentation hooks to collect data only for those regions. For implementing the random forest model itself, we used the freely available R package [60].

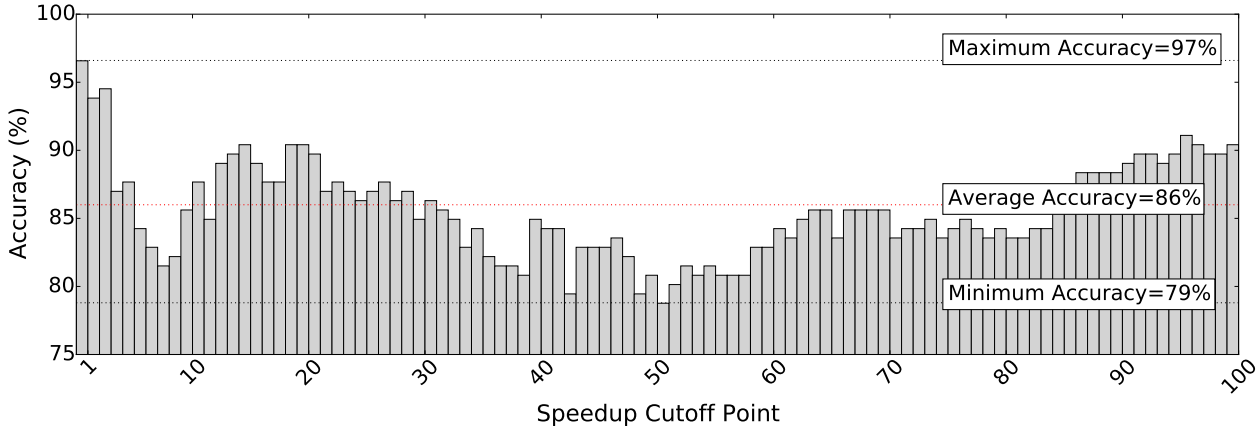
6.2 Results and Analysis

Recall that our model predicts speedup intervals. Speedup intervals of interest are provided by the user and are fed into the tool-chain as an input to the model construction phase. We use the user-provided speedup intervals to discretize measured speedup into a set of class labels, for all the datapoints in our dataset, prior to model construction. We then construct a random-forest model, and use LOOCV to evaluate our model. Specifically, we use all training datapoints to construct one random forest model, then use the model to predict speedup range for the set aside datapoint. This process gets repeated for all the datapoints in the dataset. Our random forest technique is an ensemble of 1000 decision trees, where each tree is constructed using a random subset of features and training datapoints.

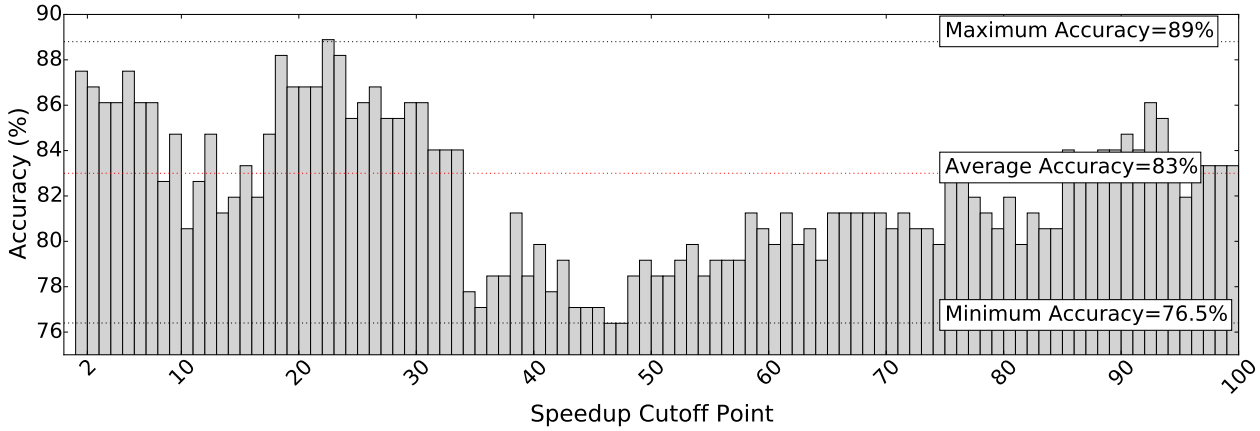
In what follows, we first show the accuracy of our model for *one* set of speedup intervals that we intuitively found useful. Second, we show our model is stable by presenting its accuracy for an *arbitrary* set of speedup intervals. Third, we show our model's accuracy for finer granularity speedup intervals. Fourth, we show our static analysis accuracy, and present the overall accuracy when static analysis results are fed into the machine learning model. Finally, we discuss the useful insights the models provide to users on how to optimize their code.

Benchmarks	Meas.	Pred.	Benchmarks	Meas.	Pred.	Benchmarks	Meas.	Pred.
backprop2-1-rd	L	L	srad_v22-1-rd	L	L	one5-3-μ	H	H
bfs0-mv0-1-ls	L	L	sssp0-mv0-1-ls	L	L	one7-1-μ	H	H
bfs0-mv0-2-ls	L	L	sssp0-mv0-2-ls	L	L	one8-1-μ	H	H
bfs0-mv5-2-ls	L	L	sssp0-mv1-2-ls	L	H	raycasting1-1-capp	H	H
bfs0-mv6-2-ls	L	L	sssp0-mv3-2-ls	L	H	sp0-mv2-2-ls	H	H
bfs2-1-rd	L	L	sssp0-mv7-2-ls	L	L	sp0-mv3-2-ls	H	H
bfs2-mv0-1-ls	L	L	sssp0-mv8-2-ls	L	L	sp0-mv4-2-ls	H	H
bfs2-mv0-2-ls	L	L	sssp0-mv9-2-ls	L	L	sp0-mv6-2-ls	H	H
bfs2-mv1-2-ls	L	L	sssp2-mv0-1-ls	L	L	sp0-mv8-2-ls	H	H
bfs2-mv3-2-ls	L	L	sssp2-mv0-2-ls	L	L	sp0-mv9-2-ls	H	H
bfs2-mv4-2-ls	L	L	sssp2-mv1-2-ls	L	L	sp1-mv2-2-ls	H	H
bfs2-mv5-2-ls	L	L	sssp2-mv7-2-ls	L	L	sp1-mv3-2-ls	H	H
bfs2-mv6-2-ls	L	L	sssp2-mv8-2-ls	L	L	sp1-mv4-2-ls	H	H
bfs2-mv8-2-ls	L	L	sssp2-mv9-2-ls	L	L	sp1-mv5-2-ls	H	H
bfs2-mv9-2-ls	L	L	two3-1-μ	L	L	sp1-mv6-2-ls	H	H
b+tree1-1-rd	L	L	two4-1-μ	L	L	sp1-mv7-2-ls	H	H
b+tree2-1-rd	L	L	ftv1-1-other	L	L	sp1-mv8-2-ls	H	H
euler3d1-1-rd	L	L	bfs0-mv1-2-ls	H	H	sp1-mv9-2-ls	H	H
euler3d4-1-rd	L	L	bfs0-mv2-2-ls	H	H	sp2-mv1-2-ls	H	H
fft1-1-capp	L	L	bfs0-mv3-2-ls	H	H	sp2-mv2-2-ls	H	H
fft2-1-capp	L	L	bfs0-mv4-2-ls	H	H	sp2-mv3-2-ls	H	H
ftv0-1-other	L	L	bfs0-mv7-2-ls	H	H	sp2-mv4-2-ls	H	H
ftv4-1-other	L	L	bfs0-mv9-2-ls	H	H	sp2-mv7-2-ls	H	H
ftv6-1-other	L	L	bfs2-mv2-2-ls	H	L	sp2-mv8-2-ls	H	H
ftv7-1-other	L	L	bfs2-mv7-2-ls	H	H	sp2-mv9-2-ls	H	H
one6-1-μ	L	L	convolution1-1-capp	H	H	sp3-mv2-2-ls	H	H
one6-2-μ	L	L	euler3d2-1-rd	H	H	sp3-mv3-2-ls	H	H
one6-3-μ	L	L	nn0-1-rd	H	H	sp3-mv4-2-ls	H	H
one9-1-μ	L	L	nn1-1-rd	H	H	sp3-mv5-2-ls	H	H
particle_filter1-1-rd	L	H	nn2-1-rd	H	H	sp3-mv6-2-ls	H	H
four6-1-μ	L	L	montecarlo1-1-capp	H	H	sp3-mv7-2-ls	H	H
four2-1-μ	L	H	montecarlo2-1-capp	H	H	sp3-mv8-2-ls	H	H
four8-1-μ	L	L	nn3-1-rd	H	H	sp3-mv9-2-ls	H	H
four9-1-μ	L	L	one10-1-μ	H	H	srad_v11-1-rd	H	H
four10-1-μ	L	L	one11-1-μ	H	H	srad_v15-1-rd	H	H
four3-1-μ	L	L	one11-2-μ	H	H	sssp0-mv2-2-ls	H	H
four4-1-μ	L	L	one11-3-μ	H	H	sssp0-mv4-2-ls	H	H
four5-1-μ	L	L	one1-1-μ	H	H	sssp0-mv5-2-ls	H	H
four7-1-μ	L	L	one12-1-μ	H	H	sssp2-mv2-2-ls	H	H
sp0-mv1-2-ls	L	L	one1-2-μ	H	H	sssp2-mv3-2-ls	H	H
sp1-mv0-1-ls	L	H	one13-1-μ	H	H	sssp2-mv4-2-ls	H	H
sp1-mv0-2-ls	L	L	one1-3-μ	H	H	sssp2-mv5-2-ls	H	H
sp2-mv0-1-ls	L	L	one2-1-μ	H	H	sssp2-mv6-2-ls	H	H
sp2-mv0-2-ls	L	H	one2-2-μ	H	H	three1-1-μ	H	H
sp2-mv6-2-ls	L	L	one2-3-μ	H	H	tsearch1-1-capp	H	H
sp3-mv0-1-ls	L	L	one3-1-μ	H	H	two1-1-μ	H	H
sp3-mv0-2-ls	L	H	one4-1-μ	H	H	two2-1-μ	H	H
srad_v13-1-rd	L	L	one5-1-μ	H	H	two5-1-μ	H	H
srad_v14-1-rd	L	L	one5-2-μ	H	H	two6-1-μ	H	H
Accuracy					**%90**			

Table 6.1: Speedup range prediction accuracy for following ranges: low speedup (speedup ≤ 3), and high speedup (speedup > 3).



(a) Platform 1 (GTX750)



(b) Platform 2 (GTX660)

Figure 6.1: Prediction accuracy for different binary classifiers. The x-axis represents the cutoff point that divides the speedup range into low and high. Kernels with speedup $\leq$ x will be labeled as low (L) and kernels with speedup $>$ x will be labeled as high (H). The y-axis shows the cross-validation accuracy for a model that is constructed with a dataset labeled as such.

6.2.1 Model Accuracy

Summary: *Table 6.1 shows the leave-one-out cross-validation accuracy of our technique. The model predicts low (speedup $\leq$ 3), and high speedup (speedup $>$ 3) ranges with 94% accuracy.*

Recall that speedup intervals are selected prior to model construction. Different users find different cutoffs interesting, depending on the capabilities of their host CPU machine and target GPU machine, as well as their application domain. An interesting cutoff set is the one that can clearly identify the speedup and slowdown regions. A code belongs to a speedup region if the speedup achievable is high that it is clear that the time spent to port the code is worthwhile. A code belongs to a slowdown region if the speedup achievable is very low. We define $(0, 3]$ as the low-speedup range where it is unlikely to get high speedup, independent of the amount of time and effort invested. We define $(3, \infty]$ as high-speedup range, where getting high speedup is guaranteed.

Table 6.1 shows the measured and predicted speedup range across 147 kernels. Speedup values are measured on platform 1 (shown in Table 5.1). The speedup interval for each kernel is predicted using the model constructed on the remaining 146 kernels. Across 147 kernels there are only 9 kernels misclassified (highlighted in gray). This indicates that the cross-validation accuracy is 94%, demonstrating the high accuracy of our approach. We also measured the ratio of the high-speedup kernels that are correctly identified as high speedup, which is $\frac{87}{88} = 99\%$, and is known as sensitivity in statistical terms. Likewise, we measured the ratio of the low-speedup kernels that are correctly identified as low speedup, which is $\frac{61}{69} = 88\%$, and is known as specificity in statistical terms. For any test, there is usually a trade-off between sensitivity and specificity. The very high sensitivity value indicates that our model outcome is highly reliable when predicts a kernel having high speedup.

Analysis of the underpredicted kernels As shown, our model underpredicts only one kernel, bfs2-mv2-2-ls. The kernel high speedup (217) is due to the presence of expensive floating-point sine operations, however our feature cutoffs classifies its sinf ratio as low, since it is lower than the threshold. Recall that feature thresholds are automatically found using equal frequency binning,

	mul	div	coalesced	Lbdiv32	sincos	exp	sqrt	compInt	ksize	par
particle_filter1-1-rd	L	L	L	H	L	L	L	H	H	L
sp3-mv0-2-ls	L	L	H	L	L	L	L	H	L	L
sssp0-mv3-2-ls	H	L	L	L	L	L	L	L	H	L
sssp0-mv1-2-ls	H	L	L	L	L	L	L	L	H	L
four4-1-μ	H	L	L	H	L	L	L	L	H	L
four2-1-μ	H	L	L	H	L	L	L	H	H	L
sp2-mv0-2-ls	H	H	L	H	L	L	L	L	L	L
sp1-mv0-1-ls	H	H	L	H	L	L	L	L	H	L

Table 6.2: Feature vector of mispredicted kernels.

which means more than half of the kernels had higher sinf ratio than bfs2-mv2-2-ls's. Adding more kernels with low sinf ratio can fix this problem.

Analysis of the overpredicted kernels Our model overpredicts the speedup of eight kernels depicted in Table 6.2. For four of these kernels (particle_filter1-1-rd, four4-1-μ, sp2-mv0-1-ls and sp1-mv0-1-ls) there are one or more datapoints with similar feature vectors in the training set whose speedups are High. This implies that our set of features are not explanatory enough for these kernels. Three kernels (sp3-mv0-2-ls, sssp0-mv3-2-ls, and sssp0-mv1-2-ls) have unique feature vectors, and one kernel (four2-1-μ) has two datapoints with similar feature vectors in the training set whose speedups are also low. The fact that our machine learning technique mispredicts these cases indicates that there are many datapoints in the training set with High speedup which are close to these mispredicted datapoints in feature space. Adding more datapoints with low speedup would help to solve this problem.

6.2.2 Model Stability

Summary: *Figure 6.1(a) shows the accuracy of our technique across different sets of speedup cutoff on GPU platform 1. As shown, the prediction accuracy is always above 79%, irrespective of the choice of cutoff. Figure 6.1(b) shows the same accuracy results on GPU platform 2, and the prediction accuracy is always above 76%. Therefore, our technique is robust across different platforms and/or different speedup intervals.*

Different users find different cutoff interesting. The availability of time and human resources,

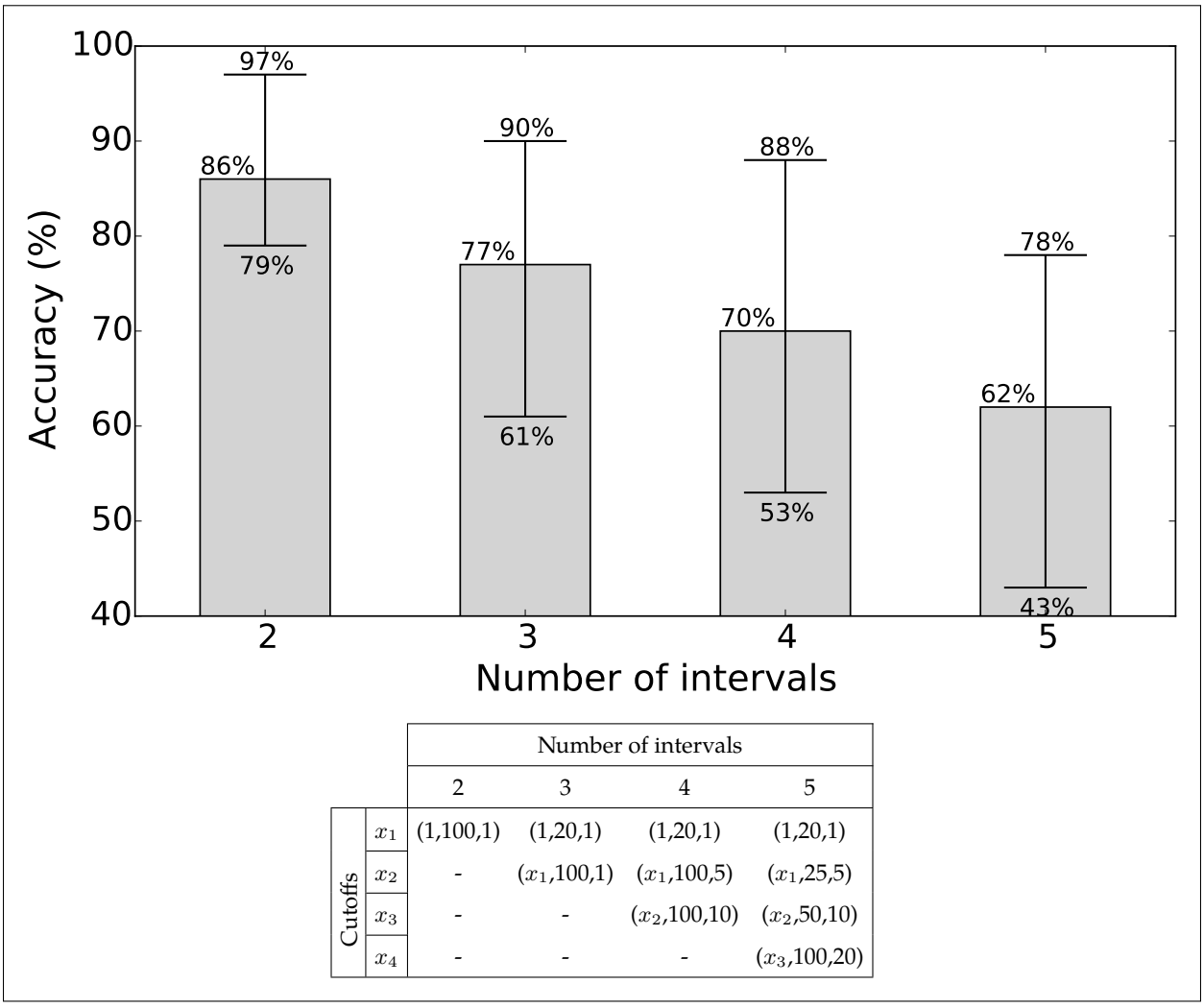
the importance of the problem at hand, the capability of the host CPU machine and the target GPU machine, are among the factors that make different users to find different cutpoints interesting. To this end, we allow the cutpoints, which divide the speedup range into different intervals, to be defined by users and fed into our model as an input. In this subsection, we study the impact of the cutpoint choice on the overall model accuracy.

To partition the speedup range into low and high, one cutoff point, x, needs to be defined, where $(0, x]$ and (x, ∞) capture the low and high speedup intervals, respectively. To study the impact of the cutpoint on accuracy, we vary the cutoff,x, from 0 to 100 in steps of 1. For each cutoff, we relabel our dataset and construct a new model, and measure its LOOCV accuracy. Figure 6.1(a) shows the prediction accuracy for different cutoffs on GPU platform 1. As shown, our technique maintains minimum, average and maximum accuracy of 79%, 86% and 97%, respectively. Note here that the slight differences in accuracy across different cutoffs is partly due to changes in the number of datapoints within each interval. Too many or too little datapoints in a bin can bias the model and hurt the generalization accuracy. A fair study of the relationship between cutpoint and accuracy requires a very large dataset, where we can maintain a balanced distribution of datapoints across different intervals, for any cutpoint. Figure 6.1(b) shows the accuracy results on GPU platform 2, which achieves the minimum, average and maximum accuracy of 76%, 82.5%, and 89%, respectively. Different platforms show different accuracy for the same cutoff, as speedup distribution is different across different platforms. In conclusion, our technique is robust to changes in platforms and speedup cutoffs.

6.2.3 Model Precision

Summary: *Figure 6.2 represents the minimum, maximum, and average speedup range prediction accuracy, with the number of speedup intervals varying from 2 to 5. As shown, the prediction accuracy drops as the number of intervals increases.*

We can achieve higher speedup range prediction precision by decreasing the speedup interval's width, or increasing the number of intervals in a fixed speedup span. Figure 6.2 represents the



		Number of intervals			
		2	3	4	5
Cutoffs	x_1	(1,100,1)	(1,20,1)	(1,20,1)	(1,20,1)
	x_2	-	(x_1,100,1)	(x_1,100,5)	(x_1,25,5)
	x_3	-	-	(x_2,100,10)	(x_2,50,10)
	x_4	-	-	-	(x_3,100,20)

Figure 6.2: The figure shows the accuracy-precision tradeoff. The Table below shows the space range that each bar represents. (l, u, s) at row i and column j shows that cutoff x_i for j intervals sweeps between l and u in steps of s.

minimum, maximum and average prediction accuracy, as the number of intervals increases from 2 to 5, within the speedup span of (1,100). The minimum, maximum and average accuracy are measured across different models constructed with different speedup cutoffs. For example, the second bar represents the average, minimum and maximum accuracy across all models constructed with two speedup cutoffs, (x, y), defined by $speedup \leq x$, $speedup \geq y$, and $x < speedup < y$, where x varies from 1 to 20 in steps of 1 and y varies from x_1 to 100 in steps of 1, as clarified in the Table below the Figure. For example, (3,20) is a cutoff set we visited through our search which achieved 85% accuracy.

As shown in Figure 6.2, the model accuracy goes down as the number of intervals (classes) increases. This is because, with each additional cutpoint, each interval gets smaller and there is more probability that a predicted datapoint to end up in the wrong interval.

6.2.4 Dynamic Effect Role on Speedup Range Prediction

Summary: *There are two dynamic variables that control the dynamic value of each feature. These dynamic variables are loop trip counts and branch probabilities. We show that the discretized value of these features are generally robust to variations in branch probability and loop trip count. Figure 6.3 shows that for 31 out of 34 kernels, variations in branch probability do not affect the speedup range prediction accuracy. Figure 6.4 shows that for 21 out of 23 kernels, variations in loop trip count do not affect the speedup range prediction accuracy. Therefore, static analysis is sufficient to estimate discretized feature values accurately.*

We study the impact of branch probability on the overall speedup prediction accuracy, by varying the branch probability for each branch from 0% to 100% in steps of 25%, for all the branches in the kernel, and all the datapoints in our dataset. We then use our static-analysis tool to estimate the feature vector for each branch probability combination, and feed it into our speedup prediction model to get one speedup range prediction for each combination. For instance, for a program with 2 branches inside the region of interest, we get 25 feature vectors[1] and therefore speedup range

[1]Each branch will be assigned 5 different probability values (0, 0.25, 0.5, 0.75 and 1). There are two branches, therefore 25 combinations.

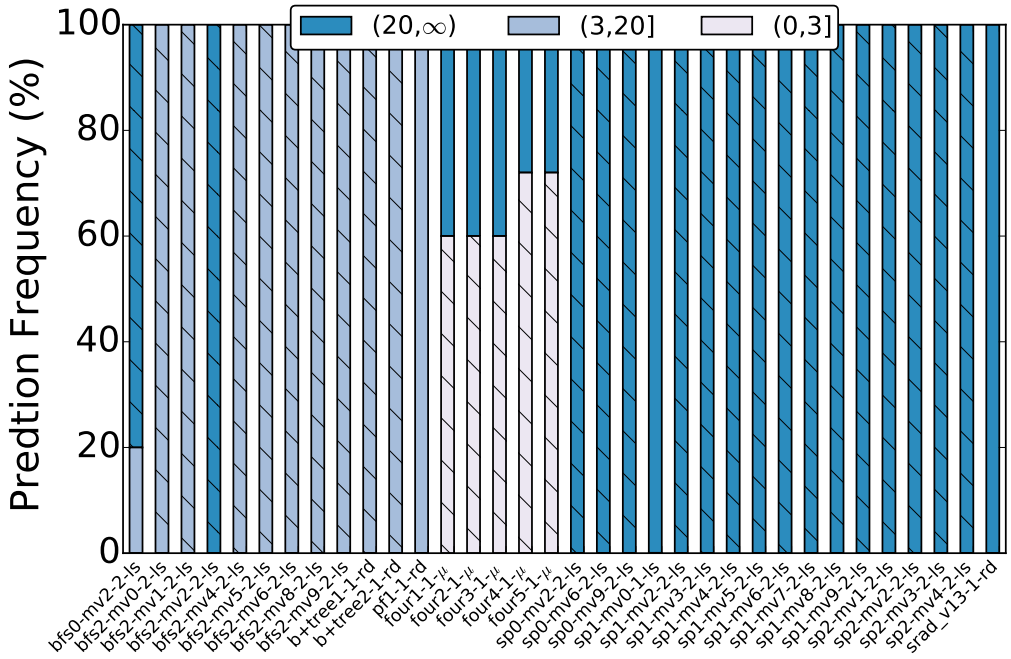


Figure 6.3: Speedup prediction sensitivity to branch ratio.

predictions. Figure 6.3 shows the per-benchmark histogram of speedup predictions for different branch probabilities, for all the datapoints in our dataset with one or more if-statements in their body. Each stack bar shows the percentage of speedup predictions belonging to each stack (speedup interval), across all different branch probability combinations. We hatched the interval in each bar where the actual speedup belongs to. As shown, in all except 3 cases, the actual speedup range matches with the majority prediction. This indicates that majority vote amongst predictions is an effective heuristic to predict the speedup.

Next, we study the impact of trip counts on the overall accuracy, by varying the trip-count values for each loop from 1 to 1000 in logarithmic steps. This includes only the loops which are part of the KBODY. We then use our static-analysis tool to estimate a feature vector for each trip-count combination, and feed it into our speedup prediction model to get one speedup range prediction for each combination. For instance, for a program with two loops, we generate 16 feature vectors[2] and therefore make 16 different speedup range predictions. Figure 6.4 is the per-benchmark histogram

[2]Each loop will be assigned 4 different trip counts (1, 10, 100 and 1000). There are two loops, therefore 16 combinations.

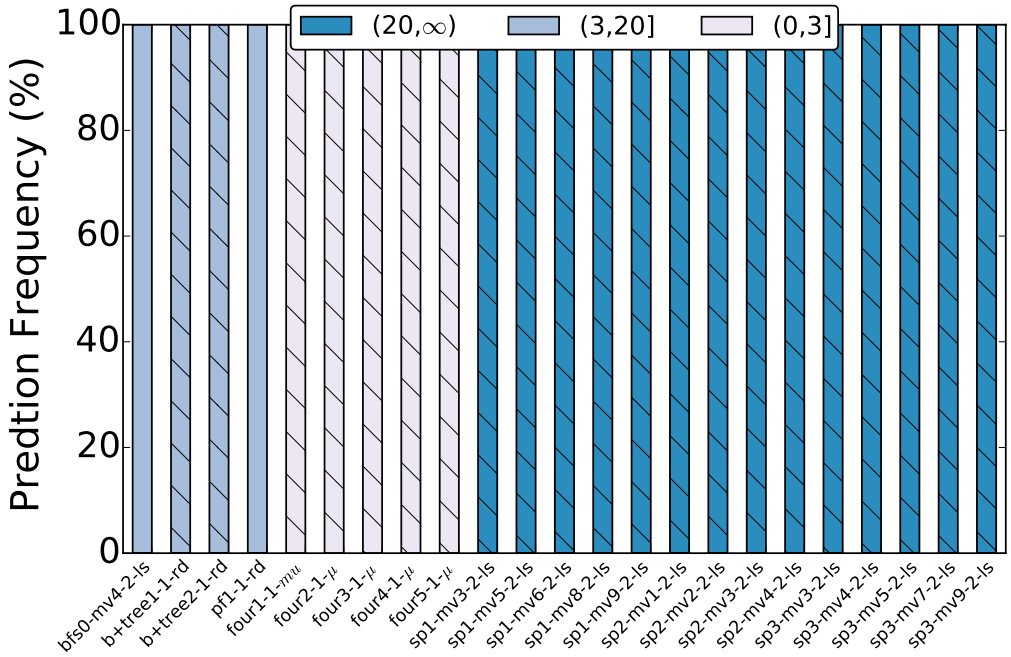


Figure 6.4: Speedup prediction sensitivity to loop trip count.

of speedup predictions for different loop trip counts, for all the datapoints in our dataset with one or more for-loops in their KBODY. Each stack bar shows the percentage of speedup predictions belonging to each stack (speedup interval), across all different trip-count combinations. We hatched the interval in each bar where the actual speedup belongs to. As shown, in all except 2 cases, the actual speedup range matches with the majority prediction. This indicates that majority vote amongst predictions is an effective heuristic to predict the speedup.

The reason the feature vectors, and consequently speedup range is robust to variations in dynamic variables is multifold: (1) The features are defined as ratios of two dynamic events, and usually the numerator and denominator scale similarly when the dynamic variable changes. (2) They are discretized, which makes the discretized feature values robust to small changes in their actual value. (3) Cutpoints are close to the extreme ends, therefore the change in dynamic value of the feature keep it within the same region as long as it falls somewhere in in the wider side. (4) Variation in dynamic variables affect CPU and GPU execution time in the same direction, and therefore speedup range remains unchanged.

Next, we explain (1) in more detail:

Trip count The feature value over a perfectly nested-loop region can be approximated by the feature value of the innermost loop. Mathematically, this can be represented as follows:

$$f = \frac{\sum_{i=1}^{N}(a_i \prod_{j=i}^{N} T_j)}{\sum_{i=1}^{N}(b_i \prod_{j=i}^{N} T_j)} \approx \frac{a_1 \prod_{j=1}^{N} T_j}{b_1 \prod_{j=1}^{N} T_j} \approx \frac{a_1}{b_1} \tag{6.1}$$

where N represents the number of loops within the nested region, T_i represents the trip count values, where $1 \leq T_i$, and a_i and b_i represent the number of static events for the feature under study (for example the number of arithmetic operations and memory operations for arithmetic intensity) within the $loop_i$, but not the $loop_{i-1}$. Loops are indexed from the innermost loop to the outermost one.

For program regions with one perfectly nested-loop region dominating the entire region, the overall feature value can be approximated by the feature value of the innermost loop, which does not depend on the trip count values of the enclosing loops. For program regions which are not perfectly nested, there is at least a level within the loop nest where the loops are appearing sequentially. If the loop depths are different across sequential loops, the overall feature value can be approximated by the feature value of the innermost loop of the deepest loop, therefore independent from the trip count. If the loop depths are the same across sequential nested loop regions, and trip counts are likely to be similar (for example, when consecutive loops are operating on the same data structure), the feature value can be approximated as follows:

$$f = \sum_{\oplus,i}^{S} f_i(Innermostloops) \tag{6.2}$$

where S is the number of sequential nested-loop regions at a given level and $\oplus$ is a special sum

operator, defined as follows:

$$\frac{a_1}{b_1} \oplus \frac{a_2}{b_2} = \frac{a_1 + a_2}{b_1 + b_2} \tag{6.3}$$

If the loop depths are the same across sequential nested-loop regions, and the feature value of their innermost loops are similar, then that feature value can be approximated as the feature value of the innermost loop of one of the loops. The only scenario that trip count can actually matter is when the code has sequential nested-loop regions with the same depth, but different feature values in their innermost loops. We found this pattern rare within the KBODY. A good GPU programming practice is to map sequential loops into separate kernels.

Branch probability The overall feature value over a perfectly nested-branch region can be approximated by the feature value of the innermost branch, therefore it is independent from the branch probability of the enclosing branches. Recall that our feature values are ratios of two dynamic events. Mathematically, this can be represented as follows:

$$f = \frac{\sum_{i=1}^{N}(a_i \prod_{j=i}^{N} P_j)}{\sum_{i=1}^{N}(b_i \prod_{j=i}^{N} P_j)} \approx \frac{a_N P_N}{b_N P_N} \approx \frac{a_N}{b_N} \tag{6.4}$$

where N represents the number of branches within the nested-branch region, P_i represents the branch probabilities, where $0 \leq P \leq 1$, and a_i and b_i represent the number of the events under study (for example the number of arithmetic operations and memory operations for arithmetic intensity) within the $branch_i$, but not the $branch_{i-1}$.

For program regions with one perfectly nested-branch region dominating the entire kernel, the overall feature value can be approximated by the feature value of the outermost branch. For program regions where the nested-branch region is not covering the entire program, the overall feature value can be approximated as the feature value of the rest of the code, if the controlled region is small, and therefore branch probability does not matter. For program regions which are not perfectly nested, there is at least a level within the nested region where the branches are appearing

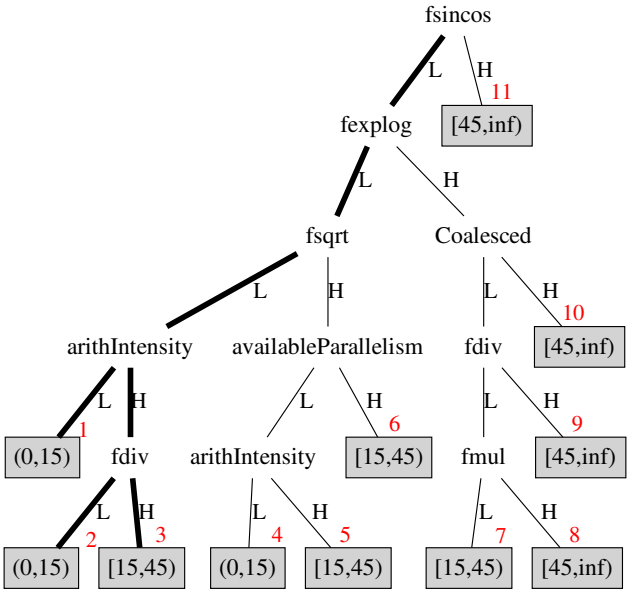


Figure 6.5: The single tree that approximates the random forest model which predicts speedup range with (15,45) cutoffs.

sequentially. If the branch depths are different across sequential branches, the overall feature value can be approximated by the feature value of the outermost branch of the shallowest branch, therefore independent from the trip count. If the branch depths are the same across sequential nested regions, and the feature value of the outermost branches are similar, then the feature value can be approximated as the feature value of the outermost branch of one of the branches.

6.2.5 Model Interpretation

Summary: *Figure 6.5 shows the decision tree that approximates the random forest model that predicts if speedup < 15, 15 ≤ speedup < 45, or speedup ≥ 45. The paths from root to leaves capture classification rules.*

A random forest model is notoriously hard to interpret as it is composed of 1000 decision trees,

and each tree has approximately 30 leaves (classification rules). To explain our model, we adapt the ideas from machine learning community to *approximate* the forest with a single tree [64]. The idea is simple, but comes at the expense of accuracy: We use the random forest model to predict the speedup for the entire program space, characterized by 1024 (2^{10}) feature vectors, and use the generated synthetic data to construct a single tree, using CART algorithm [68]. Figure 6.5 shows a tree that approximates the random forest model depicted in Table 6.1. Note that we have not grown this tree to its fullest extent in order to keep it easy for explanation. This also explains why some of our features (computation Intensity, kernel size and amount of parallelism) have not appeared in the approximation tree. The paths from root to leaves capture classification rules. We first explain the rules that classifies applications into Very High, High and Low/Medium speedup. We then explain how this tree can be used to guide programmers through GPU optimization.

According to Figure 6.5, GPU applications belong to one of the three following categories if the following conditions are met:

- **Very High Speedup (Speedup $\geq$ 45** An application has very high speedup if:

    1. It contains floating-point sine/ cosine operations (leaf 11). Trigonometric operations are computed using math libraries on CPU and will be replaced by a series of instructions, while GPU has a special hardware support for them. Therefore, CPU codes with trigonometric operations usually get very high speedup on GPU.
    2. It contains floating-point exponential operation and all its memory accesses are coalescable (leaf 10), or it contains floating-point exponential operation and also contains floating point multiplication and division operations (leaves 8 and 9). Exponential operations are slightly slower than sine/cosine operations on GPU [69]. Therefore, we would get high speedup (as opposed to very high speedup) when porting codes containing them. A code achieves very high speedup (speedup > 45), if other conditions are met: (1) All memory operations are coalescable (leaf 10). (2) Or the code contains floating-point division or multiplication operations (leaves 8 and 9). Codes benefit from having floating-point

multiplication/division operations on GPU, as there are many multiplication/division units on GPU.

3. The code has high computation intensity and high division ratio (leaf 2).

- **High Speedup (15 $\leq$ Speedup $<$ 45)** According to Figure 6.5, it is unlikely to achieve very high speedup (speedup > 45) on GPU, if the code does not contain floating-point sine/cosine or exponential/logarithmic operations. However high speedup (15 $\leq$ Speedup $<$ 45) is still possible if the code contains floating-point sqrt (fsqrt) operations and has abundant parallelism (leaf 6) or contains fsqrt and has high computation intensity (leaf 5) or has high computation-intensity but low division ratio (leaf 2).

- **Low Speedup (Speedup $<$ 15)** As shown in Figure 6.5, it is unlikely to achieve high speedup (speedup > 15) on GPU if there are no transcendental operations in the codes and (1) it has low computation intensity (leaf 1), or (2) it has high computation intensity but does not contain any floating-point division operation (leaf 2).

Optimization Insight Figure 6.5 can also be used to provide optimization hints to users. For instance, if a code has low speedup (falling under leaf 1), improving its arithmetic intensity can potentially improve its potential speedup on GPU to high (leaf 2) or very high (leaf 2), depending on whether or not it contains division operations. The paths representing this scenario are shown in bold. One way to improve arithmetic intensity is the use of shared memory, as it reduces the number of memory accesses to global memory. This tree is simplified, but in the full version of the tree, there are paths where improving arithmetic intensity, in other words or using shared memory is not useful. Considering the complexities of mapping data structures into shared memory, this is a valuable hint and time-saver to the user.

6.3 End to End Case Studies

In order to study the performance of our tool in the wild, we looked for collaborations with researchers from other departments who were seeking for help to port a project to GPU. We helped our participants to identify the promising code regions in their program, and their potential speedup on GPU. We then helped them to port their code to GPU and compared the speedup of our implementation with the predicted speedup. In the next subsections, we discuss our study and results.

6.3.1 Experimental Framework

- **Participants** Our participants were researchers from bio-statistics department at UW-Madison. As part of their research, our participants had developed a statistical inference tool for large-scale graph-structured data, which leverages the information about the graph structure to perform a more knowledgeable hypothesis testing. In nutshell, their goal is to combine clustering and hypothesis testing, in order to improve hypothesis testing accuracy. They refer to their new technique as 'block testing'. Their technique is potentially useful in many fields of science, including brain scan imaging and genomic. For example, it can be applied to find if there is a statistical difference between the magnetic resonance images (MRI) of the brains of a healthy population and a population subject to Alzheimer's disease. Brain images can be represented as a graph, where each node maps to a voxel, and each edge represent a neighboring relationship between two voxels.

  From the execution time perspective, their source code takes about a month to generate the final results. From the complexity perspective, their source code repository contains 2000 lines of C++ codes with heavy use of Eigen library [70] and Armadillo library [71]. From the programming skill perspective, our participants had no GPU programming experience.

- **Subjects** After profiling their code, we identified that two function calls together account for

more than 84% of the total execution time. LUDecomposition and FindConnectedComponent take 70% and 14% of the total execution time, respectively.

1. **Cholesky Decomposition** As part of their algorithm, they require to find the number of spanning trees in their intermediate graphs. Based on the Kirchhoff's theorem, the number of spanning trees in a graph is determined by the determinant of the matrix representing the graph. The determinant of a symmetric positive definite matrix can be computed efficiently using cholesky decomposition. Cholesky decomposition of the matrix A, factors it into the product of a lower triangular matrix and its transpose. Given the cholesky decomposition of the matrix A, the determinant can be computed as the square of the product of the diagonal entries on the lower (or upper) triangular matrix.
2. **FindConnectedComponent** One step of the algorithm is to find the connected components of their graph. A connected component of a graph is a sub-graph where there is a path between any two vertices, and there is no path to any additional vertices in the original graph.

- **Procedure** For each region of the program identified as bottleneck in previous step, we developed a new parallelizable CPU program and a corresponding GPU program. Note that we had to develop CPU programs besides GPU programs, as their CPU codes relied entirely on library function calls (LUDecomposition) and/or it was not parallelizable in its original form (FindConnectedComponent). We then use each CPU program to predict the speedup for the corresponding GPU program and therefore predict the overall speedup. We use the developed GPU program to measure the actual speedup for each region for final comparison.

6.3.2 Results

Table 6.3 shows the speedup prediction accuracy for static XAPP for the two kernels, LUDecomposition and FindConnectedComponent. The first column shows the kernel name, the second column

Kernel	Execution time (%)	Actual Speedup Range	Predicted Speedup Range
LUDecomposition	70	[0.8 , 4]	(0 , 7)
FindConnectedComponent	14	[0.1 , 135]	(0 , 7)

Table 6.3: Prediction accuracy of our case study.

shows the overall contribution of each kernel to the total execution time, the third column shows the measured speedup range, and the forth column shows the predicted speedup range. The model we use for prediction, predicts speedup into five intervals: (0,7), [7,20), [20,50), [50, 100) and 100 and above. Note here that we have reported the actual speedup as a range and not a value. This is because we measured the speedup for different set of input parameters. For LU decomposition, the exogenous value is the number of nodes, which varies from 100 to 100000 nodes. For FindConnectedComponent, there are two exogenous values: the number of nodes and the number of edges. The number of nodes varies from 100 to 10000, while the number of edges vary from the number of nodes to 10 times the number of nodes. As can be seen, LU decomposition's speedup varies within a small range, and therefore not very sensitive to input size. Our model accurately predicts the range the speedup belongs to for any input size. On the other hand, FindConnectedComponent's speedup sweeps a wide range, and is very sensitive to input size. FindConnectedComponent is a graph traversal algorithm with irregular memory access patterns and unpredictable branches. The algorithm is very memory intensive. As shown, the actual speedup varies from 0.1 to 135, as we increase the input size from 100 to 10000 nodes. This is because the CPU execution time dramatically increases as we increase the graph size, while GPU execution time almost stays unchanged. GPU has high throughput access to global memory and is capable of hiding memory latency, while CPU has a limited bandwidth. We speculate that adding a feature that measures the average number of memory transactions across static memory operations in the code could improve accuracy, as it could capture if CPU bandwidth or GPU bandwidth has reached to its maximum. We found the current accuracy reasonable for many applications and so we did not look further.

Although we mispredict the speedup range for FindConnectedComponent, the speedup projec-

tion for the entire program remains accurate – the projected overall speedup is 2.5 ($= \frac{1}{\frac{0.70}{3.5}+\frac{0.14}{3.5}+0.16}$) [3], while the actual overall speedup varies between 1 to 2.97 ($= \frac{1}{\frac{0.70}{4}+\frac{0.14}{135}+0.16}$) – since FindConnectedComponent accounts for a small portion of the program.

6.4 Summary

In this Chapter, we evaluated static XAPP accuracy for different set of cutoffs and different number of intervals. We showed that static XAPP can accurately predict if speedup is lower than or greater than 3 (with 94% accuracy). We performed a stability analysis to study how accuracy changes with the choice of cutoff. We showed that average accuracy is 86% for one GPU platform and 83% for another GPU platform. We performed a sensitivity analysis to study the impact of the number of intervals on overall accuracy, and showed that accuracy drops with the number of intervals however it remains at least a factor of 2 better than the baseline predictor. We showed our model is insightful. We concluded the chapter by presenting the accuracy results for a case study in the wild.

[3] We use the midpoint of the predicted interval for each kernel to estimate the overall speedup.

7 | Static XAPP vs. Dynamic XAPP

Contents

Unlike static XAPP, which rely on static analysis of the source code or IR, dynamic XAPP collect properties as programs run and therefore collects a rich set of highly-accurate memory-related and branch-related properties. This difference in feature measurement technique results in differences in runtime overhead and accuracy, which we will discuss in this Chapter.

7.1 Accuracy

Dynamic XAPP and static XAPP accuracy are not directly comparable as they are using different metrics for accuracy evaluation, dictated by the nature of their output. Dynamic XAPP predicts continuous speedup value and therefore evaluates accuracy as the relative **distance** between the actual and predicted speedup within the continuous space, while static XAPP predicts discretized speedup value and uses the **similarity** between the actual and predicted speedup value to evaluate accuracy. Since static XAPP cannot make continuous speedup prediction, we compare their accuracy

Benchmarks	Meas.	Pred
raycasting1-1-capp	21.1	1.0
tsearch1-1-capp	38.0	1.4
euler3d4-1-rodinia	4.0	1.4
fft1-1-capp	8.1	1.1
sp2-mv0-2-lonestar	2.5	16.3
sp3-mv0-2-lonestar	2.2	7.6
bfs0-mv1-2-lonestar	43.2	1.6
sssp0-mv1-2-lonestar	2.5	7.7
sssp0-mv3-2-lonestar	1.9	7.1
sssp0-mv6-2-lonestar	2.3	5.4
bfs0-mv8-2-lonestar	2.0	14.3
sssp0-mv9-2-lonestar	5.1	2.3
sp1-mv0-1-lonestar	2.6	3.4
sp1-mv0-2-lonestar	4.2	2.3

Table 7.1: Dynamic XAPP's misprecited kernels within the discretized speedup space, with speedup cutoff at 3.

within the discretized speedup domain, i.e. we discretize dynamic XAPP predictions as well and use misclassification ratio to evaluate both dynamic and static XAPP accuracy.

For static XAPP, we use the same model presented in Table 6.1. As shown, speedup domain is discretized into two ranges (speedup ≤ 3 or speedup > 3), and static XAPP mispredicts 8 kernels marked in gray. For dynamic XAPP, we use the same model presented in Subsection 5.2.1 [1], and discretized the continuous speedup prediction into speedup ≤ 3 or speedup > 3. Dynamic XAPP mispredicts 14 kernels listed in Table 7.1. Dynamic XAPP's lower performance compared to static XAPP within the discretized speedup space is due to the differences between their objective functions. Dynamic XAPP uses regression algorithm with the objective function that minimizes the distance between the actual and predicted speedup values, within the continuous speedup space. Static XAPP uses a classification algorithm with the objective to minimize the dissimilarity between the actual and predicted class. Using regression model to minimize dissimilarity in discrete space results in worse accuracy. For example, as shown in Table 7.1, dynamic XAPP has predicted that

[1]We slightly modified our evaluation methodology to be consistent with static XAPP. We predict speedup for all kernels and not only QFTs, and we use LOOCV to evaluate accuracy for all the kernels, rather than using leave-10-out cross-validation limited to only QFT kernels.

		Input Dataset	
		Static (147)	Dynamic (121)
Features	Static (10)	4428%	120%
	Dynamic (27)	4654%	32%

Table 7.2: Dynamic XAPP error.

sp1-mv0-1-lonestar's speedup is 2.6 when it is actually 3.4. While this error is considered small in the continuous speedup space, it is a costly error within the discretized space with cutoff 3 – a low speedup kernel is misclassified as high. This study implies that the machine learning approach that used for exact speedup prediction, although is more precise and uses perfect information about the features, is not suitable for speedup range prediction.

7.2 Features and Dataset Unification

Recall that dynamic XAPP and static XAPP use different sets of features and dataset for training. In terms of features, static XAPP uses the maximal subset of dynamic XAPP's feature set that is statically collectible. In terms of training set, static XAPP training set is the superset of dynamic XAPP's training set. Static XAPP has extra microbenchmarks to capture the impact of transcendental operations. Table 7.2 shows the dynamic XAPP model accuracy when the features and/or dataset are similar to static XAPP features and/or dataset. As shown, dynamic XAPP error quadruples (32% to 120%) when we eliminate the dynamically-collectible features. This drop in accuracy is expected as dynamic properties are critical to fine granularity speedup prediction. This study implies that if we had used dynamic XAPP to predict exact speedup, using static feature set, we would have got poor performance, even with perfect information about the features. Therefore, dynamic XAPP methodology is not appropriate for static cross-architecture performance prediction. The table also shows that dynamic XAPP error increases by orders of magnitude (32% to 4654%) when we add extra micro-kernels with transcendental operations to our dataset. This drop is also expected as more kernels with transcendental operations makes transcendental features more dominant in more models which masks the effect of memory-related and branch-related features that control

	% of datapoints with overhead in:				
	mSec.	Sec.	Min.	Hrs.	Days
dynamic XAPP	0%	18%	75%	6%	1%
static XAPP	100%	0%	0%	0%	0%

Table 7.3: Feature measurement overhead comparison between XAPP and our proposal.

the fine changes in speedup.

7.3 Overhead

Table 7.3 compares the execution time overhead of static XAPP against dynamic XAPP. The dynamic XAPP overhead can vary from seconds to days. Static XAPP has a constant overhead in milliseconds.

Dynamic XAPP is overly precise: it predicts the exact speedup number, which comes at the cost of low speed. Most often, programmers care about the range of speedup rather than an exact number. Static XAPP provides an accurate speedup range prediction which is 1000× faster than dynamic XAPP, at the precision-level that programmers care about. Table 7.3 presents the percentage of datapoints in our dataset whose overhead of execution using dynamic XAPP and static XAPP is in seconds, minutes, hours or days. As shown, although dynamic XAPP has low overhead for majority of benchmarks, it can take hours to days to collect features for some kernels. These kernels are usually the kernels that their native execution time takes 10-20 minutes, due to the large input parameters/files. Static XAPP, on the other hand, is input-independent, and its execution time is dominated by parsing the region of interest within the source code and statically extracting features, which usually takes less than a second.

7.4 Application Domain

Although static XAPP and dynamic XAPP have the same broad goal, to improve programmers' efficiency, they are not equally applicable to all applications. In this Section, we discuss when it is appropriate to use each technique. We prefer dynamic XAPP over static XAPP, when:

1. **There are dynamic library calls that can not be compiled into IR.** Static XAPP obtain properties from the intermediate representation (IR) of the program, hence it cannot analyze programs that use dynamic library calls which can not be integrated within IR at compile time.

2. **Unpredictable branch patterns.** CPU suffers from branch unpredictability more than GPU suffers from branch divergence. This indicates that kernels with irregular branching patterns do not always show slowdown on GPU. Since we can not capture branch unpredictability using static analysis, static XAPP is very likely to underpredict the speedup for codes with unpredictable branches. On the other hand, dynamic XAPP predicts GPU execution time, rather than speedup. Therefore, it does not suffer from not having a feature that captures branch unpredictability which only affects CPU execution time.

3. **Granularity of speedup prediction matters.** Static XAPP predicts speedup intervals, rather than actual speedup values. This level of granularity can be insufficient for region ordering or algorithmic exploration, if the predicted speedup interval is the same across different program regions or different algorithms.

7.5 Summary

In this chapter, we compared dynamic XAPP against static XAPP and showed that differences in their methodologies is a direct consequences of the differences in their goals. Static XAPP is a compiler-based tool that predicts speedup from the analysis of the intermediate representation (IR) of the sequential C/C++ code, while dynamic XAPP is a dynamic binary instrumentation tool that predicts speedup from the analysis of the final binary files of the sequential C/C++ code. We showed that the machine learning approach used for dynamic XAPP (the ensemble of stepwise-regression learners) is not appropriate for learning the speedup range, even with the perfect information about the features. We showed that dynamic XAPP accuracy dramatically drops, when it uses the same

set of features and/or the same training set as static XAPP. We also compared the runtime overhead of static XAPP and dynamic XAPP and showed that static XAPP is orders of magnitude faster than dynamic XAPP, particularly for applications whose native execution time takes too long. We concluded with a discussion about when it is appropriate to use each technique.

8 | Conclusion and Future Directions

Contents

The goal of this dissertation was to propose a framework that improves GPU programmers' productivity through estimating GPU speedup for any given CPU application/algorithm. The work was motivated by the fact that GPU programming is still challenging and time-consuming for the majority of programmers and the only way to know if a program is viable on GPU is to invest time and money to develop a GPU code on a real GPU hardware. Our work, presented in this dissertation, identified two methodologies for quick and accurate (static XAPP), or precise and accurate (dynamic XAPP) GPU speedup prediction. This chapter concludes the dissertation with a summary of results and technical contributions and presents possible directions for future work.

8.1 Summary of Results

We showed that our frameworks satisfy the six key properties of cross-architecture performance modeling:

- **Accuracy** – the degree to which the actual and predicted performance matches; Dynamic XAPP projects GPU speedup with 27% deviation from actual speedup. Static XAPP predicts speedup to be greater than or lower than 3 with 94% accuracy.

- **Precision** – the granularity of speedup prediction. Dynamic XAPP predicts the exact value of the speedup, which is overly precise. Static XAPP predicts speedup intervals with 86%, 77%, 70% and 62% average accuracy for 2, 3, 4 and 5 intervals, respectively.

- **Application-generality** – being able to model a wide variety of applications; For dynamic XAPP, there is nothing inherent to our technique that makes it incapable of predicting certain application types. Although the current implementation has features to capture the impact of textured memory and constant memory, it does not, since we do not have any kernel with such memory optimization within our dataset. Static XAPP cannot capture codes with dynamic library function calls that cannot be embedded within IR at compile time.

- **Hardware generality** – being easily extendable for various GPU hardware platforms; We evaluated our tools on two different NVIDIA cards and their accuracy was reasonably high across both. For dynamic XAPP, CV_{error} is 27% and 36% on platform 1 and 2, respectively. For static XAPP, the cross-validation accuracy of speedup/slowdown prediction is 86% and 83% on platform 1 and 2, respectively.

- **Runtime** – being able to predict performance quickly; Static XAPP and dynamic XAPP have similar training-phase overhead, which is mainly dominated by the speedup measurements of the kernels within our training set on the target GPU. This usually takes about 30 minutes for our dataset of 147 datapoints. Model construction also takes about 30 minutes for dynamicXAPP and 30 seconds for static XAPP. Considering the recurring overhead, dynamic XAPP is as slow as dynamic binary instrumentation – which introduces 10×-100× slowdown compared to the native execution – while static XAPP is as fast as static analysis – which takes less than a millisecond.

- **Programmer usability** For dynamic XAPP, a user only needs to tag her regions of interest. The entire process is automated. For static XAPP, a user can only tag her regions of interest, but if she augments her code with the number of loop iterations and branch probability, higher accuracy is expected.

8.2 Summary of Contributions

Contributions are stablished in four broad categories:

Contribution 1 The observation that for any GPU platform, GPU execution time can be formulated in terms of dynamically or statically-collected program properties as the only *variables*, while GPU hardware characteristics will be captured indirectly within the model's *coefficients* and/or the model structure, i.e the way the the variables interact. This dissertation is the first to formulate the problem this way.

Contribution 2 Finding a list of dynamic and static program properties that capture the GPU speedup.

Contribution 3 Discovering a set of engineering techniques to demonstrate that the problem, as formulated, was solvable. In particular, we show that established machine learning techniques (the ensemble of step-wise regression learners for dynamic XAPP and random forest for static XAPP) are sufficient to provide highly accurate speedup prediction.

Contribution 4 Developing binary instrumentation tools that automatically collects dynamic program properties from CPU binaries.

Contribution 5 Developing an LLVM-based compiler framework that automatically collects features from the intermediate representation of a CPU program.

Contribution 6 A number of qualitative contributions including: a detailed discussion and identification of static and dynamic program properties that influence GPU performance, a discussion of the robustness of our approach by presenting its cross-validation accuracy, sensitivity studies considering only hard-to-predict applications (low-speedup applications), and a detailed analysis of why ensemble prediction is required. We also discuss the ensemble model in terms of features selected and provide insight into our ensemble results.

8.3 Future Directions

In this work, we developed an automated performance prediction tool that can provide accurate estimates of GPU execution time for any CPU code *prior to developing the GPU code*. Broadly the idea is applicable to many accelerators, and this work can form the foundation for the growing body of work on accelerators. *This work is the first to develop such a technique, and the technique is robust and accurate.* Its mathematical and elegant framing of speedup is its key contribution, and this in turn opens up opportunities for many more use cases and future research directions, making a case for its significance.

The implications of this work are multifold. Narrowly, in the context of speedup prediction for GPUs, one direction is to determine how much additional training data can improve accuracy, and what is the accuracy "limit" of the machine-learning approach providing a large dataset. While our case study has demonstrated empirical examples of XAPP producing false positives, further exploration that develops a more rigorous and formal understanding of what can be learned effectively can be useful. The features we have determined to be interesting and the final regression model can complement GPU analytical models. While our specific implementations (static XAPP and dynamic XAPP) are accurate, improving their accuracy be developing a richer dataset and feature set is a promising direction for future work.

More broadly, the success of our XAPP framework opens up many directions of future work. First, this methodology can be naturally applied to many emerging programmable accelerators (like

FPGAs, coarse-grained reconfigurable accelerators, fixed-function accelerators, etc.) to determine accurate estimates of performance benefits rapidly and at a very early-stage. The primary limitation is the availability of training data. Second, this technique can be extended to learn other functions like power/energy, or predicting speedup from vectorized or multithreaded implementations, or predicting speedup including the memory copy time. Third, it seems plausible that performance counters of modern microprocessors can capture a subset (or non-overlapping set) of the program properties we have found to be useful. One direction of future work is to examine to what extent performance counters are sufficient. Finally, beyond predicting just single output metrics, we could learn correlation between fundamental program properties and properties of hardware-specific implementations to aid in auto-compilers. For example, we could define GPU-specific coding or programming transformations as the output features and use machine-learning to predict whether or not these transformations are to be applied. Such a framework can be combined with traditional compilers to auto-compile sequential codes for different accelerators.

A | Machine Learning Background

In this section, we first present background on linear regression covered in standard texts [72, 73]. We discuss the basics of how the *response* is modeled as a linear combination of *features* and how linear regression can capture non-linearity. We then discuss statistical metrics for evaluating the quality of a model. Finally, we introduce feature selection techniques to control non-linearity and overcome issues of overfitting.

Basic Linear Regression Given a set of n observations as training data, the goal of the regression analysis is to find a relationship between input features and the output response, such that the sum of squared errors (SSE) is minimum. Each observation consists of a vector of p features (also known as independent variables) $x_i = (x_{1i} \dots x_{pi})$ and a response (also known as dependent variable) y_i. $\hat{y}_i$ is formulated in terms of features and coefficients (β) as follows:

$$\beta = (\beta_0, \beta_1, ..., \beta_p) : \hat{y}_i = \beta_0 + \sum_{j=1}^{p} \beta_j x_{ji}$$

$$\text{s.t. } \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 \text{ is a minimum}$$

An underlying assumption of a standard linear regression is that the error value between prediction and response ($e_i = y_i - \hat{y}_i$) is a Gaussian random variable with zero mean.

Often times, basic features interact with each other in how they influence the output, which can be modeled by defining *derived features*. For example, if the product of three features (x_p, x_q, x_r) influences the response, we can define $x_s = x_p * x_q * x_r$ as a new feature. Similarly, we can define higher order power terms.

Regularized Regression Linear regression can easily get overfitted, if get used improperly. If the feature set is overly rich (i.e. if there are too many noisy / irrelevant features compared to the number of observations), it is possible for linear regression to fit the training data very well with very small (even zero) residual error on the training set, but performs poorly on unseen test set. The reason is that an overly rich model family will fit both the true feature–response relation as well as noises incidental to the training set.

The state-of-the-art technique to prevent overfitting is regularization. In particular, regularized regression using LASSO [74] is a method that controls overfitting. In addition, LASSO tends to produce so-called sparse solutions: all but a few of the resulting coefficients β_j will be zero. This is useful as a feature selection tool: the features with nonzero coefficients are selected by the model as being important to explain the responses. We borrow from canonical descriptions of LASSO [75] below. LASSO minimizes a regularized version of residual error:

$$\min_{\beta} \sum_{i=1}^{n} (\hat{y}_i - y_i)^2 + \lambda * \sum_{j=1}^{p} |\beta_j|, \tag{A.1}$$

where λ is a regularization weight that can be automatically tuned using a technique called cross-validation. Consider a fixed λ value. Cross-validation randomly partitions the training data into K folds. The k-th fold ($k = 1, \ldots, K$) with n/K observations is set aside in turn as a "mini test set," while a model $\hat{y}^{(k)}$ is trained on the remaining $K - 1$ folds. The residual of $\hat{y}^{(k)}$ is then computed on the set-aside fold:

$$\sum_{i \in \text{ fold } k} (\hat{y}_i^{(k)} - y_i)^2. \tag{A.2}$$

This is repeated for each of the K folds, and the residual is then averaged to obtain the so-called

cross-validation residual:

$$\frac{1}{K}\sum_{k=1}^{K}\sum_{i\in \text{ fold } k}(\hat{y}_i^{(k)}-y_i)^2. \tag{A.3}$$

This cross-validation residual represents an approximation to future performance on test data under that fixed λ. The whole procedure is then repeated for a different λ value. In the end, one chooses the λ that resulted in the best cross-validation residual. In many problems this technique has shown to be efficient in preventing overfitting and guiding feature selection.

Binary Decision Tree Given a set of n observations, the goal of decision tree is to create a model that predicts the value of output response in terms of input features, such that the misprediction ratio is minimum. The outcome of the technique is a tree-like structure, where internal nodes represent binary tests on the input features (e.g, the feature value is low or high), the edges between the nodes represent the outcome of the test, the leaf nodes represent the class labels, and the paths from the root to leaves represent classification rules. Decision trees are robust to noisy data.

Model comparison How well a model explains the *training data* is assessed using various statistical measures, including R^2 and *Adjusted* R^2. R^2 shows the fraction of variations in the output which can be explained by the model. It increases with the number of features, and hence cannot be used to compare models with a different number of features. *Adjusted* R^2 increases with a new feature only if it adds to the explanatory power of the model.

Feature Selection Feature selection is required when there are many redundant features and a limited number of datapoints, to reduce the risk of overfitting. Exhaustive, forward (start with empty model and add features) and backward (start with all features and eliminate features until explanatory power drops drastically) are different variations of feature selection.

Ensemble Prediction Ensemble prediction is a set of learned models whose predictions are combined in a certain way to provide prediction for new instances. It is a useful technique when

the base learners are unstable, or the dataset size is small [42].

Bibliography

[1] "A Library of Parallel Algorithms." `http://www.cs.cmu.edu/~scandal/nesl/algorithms.html`.

[2] K. Hoste and L. Eeckhout, "Microarchitecture-independent workload characterization," *Micro, IEEE*, vol. 27, no. 3, pp. 63 –72, 2007.

[3] K. Datta, S. Kamil, S. Williams, L. Oliker, J. Shalf, and K. Yelick, "Optimization and performance modeling of stencil computations on modern microprocessors," *SIAM review*, vol. 51, no. 1, pp. 129–159, 2009.

[4] J. Meng and K. Skadron, "Performance modeling and automatic ghost zone optimization for iterative stencil loops on gpus," in *Proceedings of the 23rd international conference on Supercomputing*, pp. 256–265, ACM, 2009.

[5] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, pp. 65–76, Apr. 2009.

[6] C. Nugteren and H. Corporaal, "The boat hull model: adapting the roofline model to enable performance prediction for parallel computing," in *PPOPP '12*, pp. 291–292, 2012.

[7] P. J. Joseph, K. Vaswani, and M. J. Thazhuthaveetil, "Construction and use of linear regression models for processor performance analysis," in *HPCA*, pp. 99–108, 2006.

[8] P. J. Joseph, K. Vaswani, and M. J. Thazhuthaveetil, "A predictive performance model for superscalar processors," in *Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on*, pp. 161–170, 2006.

[9] W. Jia, K. Shaw, and M. Martonosi, "Stargazer: Automated regression-based gpu design space exploration," in *ISPASS '12*.

[10] B. C. Lee and D. M. Brooks, "Accurate and efficient regression modeling for microarchitectural performance and power prediction," in *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ASPLOS XII, pp. 185–194, ACM, 2006.

[11] W. Wu and B. Lee, "Inferred models for dynamic and sparse hardware-software spaces," in *Microarchitecture (MICRO), 2012 45th Annual IEEE/ACM International Symposium on*, pp. 413–424, 2012.

[12] I. Baldini, S. J. Fink, and E. Altman, "Predicting gpu performance from cpu runs using machine learning," in *Computer Architecture and High Performance Computing (SBAC-PAD)*, pp. 254–261, IEEE, 2014.

[13] N. Ardalani, C. Lestourgeon, K. Sankaralingam, and X. Zhu, "Cross-architecture performance prediction (xapp): Using cpu code to predict gpu performance," in *Proceedings of the 48th International Symposium on Microarchitecture*, pp. 725–737, ACM, 2015. **Honorable Mention**.

[14] J. Meng, V. Morozov, K. Kumaran, V. Vishwanath, and T. Uram, "Grophecy: Gpu performance projection from cpu code skeletons," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, p. 14, ACM, 2011.

[15] S. Hong and H. Kim, "An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness," in *ISCA '09*.

[16] C. Nugteren and H. Corporaal, "A modular and parameterisable classification of algorithms," Tech. Rep. ESR-2011-02, Eindhoven University of Technology, 2011.

[17] M. R. Meswani, L. Carrington, D. Unat, A. Snavely, S. Baden, and S. Poole, "Modeling and predicting performance of high performance computing applications on hardware accelerators," *International Journal of High Performance Computing Applications*, vol. 27, no. 2, pp. 89–108, 2013.

[18] B. Lee and D. Brooks, "Illustrative design space studies with microarchitectural regression models," in *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pp. 340–351, 2007.

[19] E. İpek, S. A. McKee, R. Caruana, B. R. de Supinski, and M. Schulz, "Efficiently exploring architectural design spaces via predictive modeling," in *Architectural support for programming languages and operating systems*, ASPLOS XII, pp. 195–206, 2006.

[20] J. J. Yi, D. J. Lilja, and D. M. Hawkins, "A statistically rigorous approach for improving simulation methodology," in *High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings. The Ninth International Symposium on*, pp. 281–291, IEEE, 2003.

[21] R. H. Saavedra and A. J. Smith, "Analysis of benchmark characteristics and benchmark performance prediction," *ACM Transactions on Computer Systems (TOCS)*, vol. 14, no. 4, pp. 344–384, 1996.

[22] A. P. L. K. John, "Performance prediction using program similarity,"

[23] K. Hoste, A. Phansalkar, L. Eeckhout, A. Georges, L. K. John, and K. D. Bosschere, "Performance prediction based on inherent program similarity," in *In PACT*, pp. 114–122, ACM Press, 2006.

[24] B. Ozisikyilmaz, G. Memik, and A. Choudhary, "Efficient system design space exploration using machine learning techniques," in *Proceedings of the 45th annual Design Automation Conference*, DAC '08, pp. 966–969, 2008.

[25] W. Jia, K. A. Shaw, and M. Martonosi, "Starchart: hardware and software optimization using recursive partitioning regression trees," in *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*, pp. 257–268, IEEE Press, 2013.

[26] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W.-m. W. Hwu, "An adaptive performance modeling tool for gpu architectures," in *PPoPP '10*.

[27] A. Kerr, E. Anger, G. Hendry, and S. Yalamanchili, "Eiger: A framework for the automated synthesis of statistical performance models," in *High Performance Computing (HiPC)*, pp. 1–6, 2012.

[28] J.-C. Huang, J. H. Lee, H. Kim, and H.-H. S. Lee, "Gpumech: Gpu performance modeling technique based on interval analysis," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 268–279, IEEE Computer Society, 2014.

[29] J. Lai and A. Seznec, "Break down gpu execution time with an analytical method," in *Proceedings of the 2012 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*, pp. 33–39, ACM, 2012.

[30] Y. Zhang and J. D. Owens, "A quantitative performance analysis model for gpu architectures," in *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, pp. 382–393, IEEE, 2011.

[31] E. Schweitz, R. Lethin, A. Leung, and B. Meister, "R-stream: A parametric high level compiler," *Proceedings of HPEC*, 2006.

[32] S.-Z. Ueng, M. Lathara, S. S. Baghsorkhi, and W. H. Wen-mei, "Cuda-lite: Reducing gpu programming complexity," in *Languages and Compilers for Parallel Computing*, pp. 1–15, Springer, 2008.

[33] D. Mikushin and N. Likhogrud, "Kernelgen–a toolchain for automatic gpu-centric applications porting," 2012.

[34] T. B. Jablin, *Automatic Parallelization for GPUs*. PhD thesis, Princeton University, 2013.

[35] Y. Zhang and F. Mueller, "Auto-generation and auto-tuning of 3d stencil codes on gpu clusters," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, pp. 155–164, ACM, 2012.

[36] T. D. Han and T. S. Abdelrahman, "hicuda: High-level gpgpu programming," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 1, pp. 78–90, 2011.

[37] "Openacc: Directives for accelerators." `http://www.openacc.org`.

[38] T. Hoshino, N. Maruyama, S. Matsuoka, and R. Takaki, "Cuda vs openacc: Performance case studies with kernel benchmarks and a memory-bound cfd application," in *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*, pp. 136–143, IEEE, 2013.

[39] "Cuda Toolkit Documentation." `http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/#coalesced-access-to-global-memory`.

[40] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, "Demystifying gpu microarchitecture through microbenchmarking," in *Performance Analysis of Systems Software (ISPASS), 2010 IEEE International Symposium on*, pp. 235–246, March 2010.

[41] K. M. Ali and M. J. Pazzani, "Error reduction through learning multiple descriptions," *Machine Learning*, vol. 24, no. 3, pp. 173–202, 1996.

[42] T. G. Dietterich, "Ensemble methods in machine learning," *Multiple classifier systems*, pp. 1–15, 2000.

[43] L. Breiman, "Bagging predictors," *Machine learning*, vol. 24, no. 2, pp. 123–140, 1996.

[44] K. S. Fu, *Sequential Methods in Pattern Recognition and Machine Learning*. Academic Press, 1968.

[45] S. Kotsiantis and D. Kanellopoulos, "Discretization techniques: A recent survey," *GESTS International Transactions on Computer Science and Engineering*, vol. 32, no. 1, pp. 47–58, 2006.

[46] A. Liaw and M. Wiener, "Classification and regression by randomforest," *R news*, vol. 2, no. 3, pp. 18–22, 2002.

[47] A. Statnikov, L. Wang, and C. F. Aliferis, "A comprehensive comparison of random forests and support vector machines for microarray-based cancer classification," *BMC bioinformatics*, vol. 9, no. 1, p. 319, 2008.

[48] M. Liu, M. Wang, J. Wang, and D. Li, "Comparison of random forest, support vector machine and back propagation neural network for electronic tongue data classification: Application to the recognition of orange beverage and chinese vinegar," *Sensors and Actuators B: Chemical*, vol. 177, pp. 970–980, 2013.

[49] T. M. Mitchell, "Machine learning. 1997," *Burr Ridge, IL: McGraw Hill*, vol. 45, 1997.

[50] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[51] M. Kulkarni, M. Burtscher, C. Casçaval, and K. Pingali, "Lonestar: A suite of parallel irregular programs," in *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pp. 65–76, IEEE, 2009.

[52] M. Sinclair, H. Duwe, and K. Sankaralingam, "Porting CMP Benchmarks to GPUs," tech. rep., University of Wisconsin-Madison, 2011.

[53] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-M. W. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," *Center for Reliable and High-Performance Computing*, 2012.

[54] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *IISWC '09*.

[55] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, *et al.*, "The nas parallel benchmarks," *International Journal of High Performance Computing Applications*, vol. 5, no. 3, pp. 63–73, 1991.

[56] L. L. Pilla, "NAS Parallel Benchmarks CUDA version." `http://hpcgpu.codeplex.com`. Accessed May 22, 2015.

[57] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey, "Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu," in *ISCA 2010*.

[58] K. Hoste and L. Eeckhout, "Comparing benchmarks using key microarchitecture-independent characteristics," in *Workload Characterization, 2006 IEEE International Symposium on*, pp. 83–92, 2006.

[59] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *PLDI '05*.

[60] "The R project for statistical computing." `http://www.r-project.org/`.

[61] P. Turney, "Technical note: Bias and the quantification of stability," *Journal of Machine Learning*, vol. 20, 1995.

[62] E. Bauer and R. Kohavi, "An empirical comparison of voting classification algorithms: Bagging, boosting, and variants," *Machine learning*, vol. 36, pp. 105–139, 1999.

[63] P. Domingos, "Knowledge discovery via multiple models," *Intelligent Data Analysis*, vol. 2, no. 3, pp. 187–202, 1998.

[64] A. Van Assche and H. Blockeel, "Seeing the forest through the trees: Learning a comprehensible model from an ensemble," in *Machine Learning: ECML 2007*, pp. 418–429, Springer, 2007.

[65] C. Ferri, J. Hernández-Orallo, and M. J. Ramírez-Quintana, "From ensemble methods to comprehensible models," in *Discovery Science*, pp. 165–177, Springer, 2002.

[66] D. Jeon, S. Garcia, C. Louie, S. Kota Venkata, and M. B. Taylor, "Kremlin: Like gprof, but for parallelization," in *ACM SIGPLAN Notices*, vol. 46, pp. 293–294, ACM, 2011.

[67] P. A. Lachenbruch and M. R. Mickey, "Estimation of error rates in discriminant analysis," *Technometrics*, vol. 10, no. 1, pp. 1–11, 1968.

[68] L. Olshen, C. J. Stone, *et al.*, "Classification and regression trees," *Wadsworth International Group*, vol. 93, no. 99, p. 101, 1984.

[69] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, "Demystifying gpu microarchitecture through microbenchmarking," in *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*, pp. 235–246, IEEE, 2010.

[70] "Eigen." `http://eigen.tuxfamily.org`.

[71] C. Sanderson, "Armadillo: An open source c++ linear algebra library for fast prototyping and computationally intensive experiments," 2010.

[72] S. Chatterjee and B. Price, *Regression analysis by example*. A Wiley-Interscience publication, New York [u.a.]: Wiley, 2. ed ed., 1991.

[73] S. Weisberg, *Applied Linear Regression*. Hoboken NJ: Wiley, third ed., 2005.

[74] R. Tibshirani, "Regression shrinkage and selection via the lasso," *Journal of the Royal Statistical Society (Series B)*, vol. 58, pp. 267–288, 1996.

[75] "A simple explanation of the lasso and least angle regression." `http://www-stat.stanford.edu/~tibs/lasso/simple.html`.