8-2017

X-MAP A Performance Prediction Tool for Porting Algorithms and Applications to Accelerators

Ashrit Shetty
Clemson University

X-MAP
A Performance Prediction Tool for Porting Algorithms and Applications to Accelerators

---

A Thesis
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Master of Science
Computer Engineering

---

by
Ashrit Shetty
August 2017

Accepted by:
Dr. Melissa C. Smith, Committee Chair
Dr. Walter B. Ligon
Dr. Richard R. Brooks

Abstract

Most modern high-performance computing systems comprise of one or more accelerators with varying architectures in addition to traditional multicore Central Processing Units (CPUs). Examples of these accelerators include Graphic Processing Units (GPU) and Intel's Many Integrated Cores architecture called Xeon Phi (PHI). These architectures provide massive parallel computation capabilities, which provide substantial performance benefits over traditional CPUs for a variety of scientific applications.

We know that all accelerators are not similar because each of them has their own unique architecture. This difference in the underlying architecture plays a crucial role in determining if a given accelerator will provide a significant speedup over its competition. In addition to the architecture itself, one more differentiating factor for these accelerators is the programming language used to program them. For example, Nvidia GPUs can be programmed using Compute Unified Device Architecture (CUDA) and OpenCL while Intel Xeon PHIs can be programmed using OpenMP and OpenCL. The choice of programming language also plays a critical role in the speedup obtained depending on how close the language is to the hardware in addition to the level of optimization. With that said, it is thus very difficult for an application developer to choose the ideal accelerator to achieve the best possible speedup.

In light of this, we present an easy to use Graphical User Interface (GUI) Tool called X-MAP which is a performance prediction tool for porting algorithms and applications to architectures which encompasses a Machine Learning based inference model to predict the performance of an application on a number of well-known accelerators and at the same time predict the best architecture and programming language for the application. We do this by collecting hardware counters from a given application and predicting run time by providing this data as inputs to a Neural Network Regressor based inference model. We predict the architecture and associated programming language by pro-

viding the hardware counters as inputs to an inference model based on Random Forest Classification Model.

Finally, with a mean absolute prediction error of 8.52 and features such as syntax highlighting for multiple programming languages, a function-wise breakdown of the entire application to understand bottlenecks and the ability for end users to submit their own prediction models to further improve the system, makes X-MAP a unique tool that has a significant edge over existing performance prediction solutions.

Dedication

I dedicate this thesis to my parents and my academic advisor without whose constant support and guidance this thesis would not have been possible. I also dedicate this thesis to my younger brother and my fellow lab members, who have always stood by my side and been an inspiration.

Acknowledgments

This thesis was made possible by the help and support of the following faculty members, family members, and colleagues.

I begin by thanking my academic advisor Dr. Melissa Smith, who graciously accepted me as her pupil. Dr. Smith's words of wisdom, immense experience, constant encouragement, and insightful suggestions allowed for the completion of this research and thesis.

I would also like to thank Dr. Walter Ligon and Dr. Richard Brooks for serving on my committee and instructing me in various aspects of computer architecture and parallel computing.

I am also thankful for my parents and my brother, who provided constant support during my research and writing this manuscript.

I also owe a debt of gratitude to my fellow colleagues from the Future Computing Technologies Lab at Clemson University: Karan, Eddie, Nilim, Sufeng, Jesse and many others whom I fail to mention for providing me a stimulating work environment during this process.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

In the field of high-performance computing, most systems have a heterogeneous architecture. This heterogeneity is due to the availability of multiple accelerators, such as Graphic Processing Units (GPU) and Intel's Many Integrated Cores (MIC) architecture called Xeon Phi (PHI), to augment the host processor. While each accelerator is unique in its own sense, one common phenomenon observed across the board is the massive parallelism provided by each of them. This provision has allowed application developers to exploit the inherent parallelism in their code resulting in massive execution time speedups.

In addition to the accelerators themselves, there is a huge variety of programming languages that can be used to target these accelerators. For example, Nvidia GPUs can be programmed using Nvidia Compute Unified Device Architecture (CUDA) or OpenCL. On the other hand, Intel PHIs can be programmed using OpenMP or OpenCL. Each of these programming languages has their own advantages and disadvantages depending on the level of optimization and portability across architectures. As an example, let us look at Nvidia CUDA as a programming language that can guarantee the best possible speedup (when proper optimization and coding techniques are used) on a Nvidia GPU due to possible architecture level optimizations. But on the contrary, a developer can use Nvidia CUDA only to program CUDA-enabled GPUs. As a second example, we look at OpenCL as an option that boasts compatibility across CPU, GPU and PHI architectures but does not guarantee maximum speedup due to lack of architecture level optimization. As demonstrated with the above two examples, we have the option to either achieve the best possible speedup or the luxury of being able to run the same code on different architectures with minor modifications.

With these many options available at our disposal, it is possible that a developer will face a dilemma as to which architecture will best suit the current needs. To make things even more difficult, a developer might commit to a particular architecture and a programming language only to find out later that it does not provide the optimal speedup. This process of trial and error to find the best possible architecture and programming language combination can be both tiresome and time-consuming. Further, as rapidly as technology is changing, it is impossible to exhaust all combinations available.

Traditional performance predictions tools, which we will be discussed in more detail in Chapter 2, utilize analytical methods to predict performance which has limited functionality both in terms of the number of architectures supported and the prediction accuracy. While some other proposed methods overcome the low prediction accuracy by utilizing regression techniques, they lack support for multiple architectures and programming languages. These challenges in addition to the fact that no current work in the literature encapsulates the prediction framework into a neatly packed GUI tool, makes the work proposed here all the more important. Moreover, the proposed work reduces the burden on the developer who no longer has to individually setup each prediction model to predict the performance of a particular implementation. The time thus saved can be utilized in the development of the actual code for the architecture and programming language combination, ultimately providing the maximum speedup.

To alleviate all the above-mentioned problems, we propose an easy to use Graphical User Interface (GUI) Tool called X-MAP which is a performance prediction tool for porting algorithms and applications to architectures. The tool lets a user insert his/her implementation for any of the supported architectures and provides a graphical visualization of the predicted run time, predicted architecture and predicted programming language for all of the supported combinations of architecture and programming language. As shown in Figure 1.1, X-MAP lets users map performance from a wide selection of architectures and programming languages.

The remainder of this thesis is structured as follows: In Chapter 2 we review previous work in the field of performance prediction and also look at some of the GUI-based performance prediction approaches that aim to predict time and architecture respectively. Additionally, we will analyze the advantages and disadvantages of these tools to better understand the problem and thus improve our approach. Chapter 3 focuses on the software architecture for our toolkit and presents the flow of data between the front-end and the back-end. In Chapter 4, we will take a look at the Qualitative

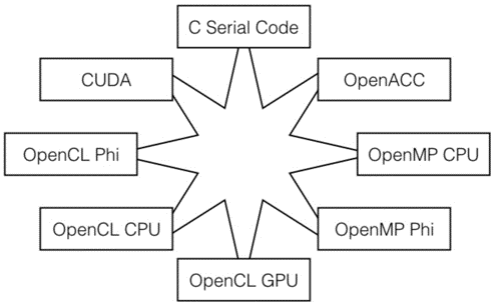

Figure 1.1: Performance mapping from one architecture to another

Model, Quantitative Model and the application profiler that form the basis of our toolkit's back-end. Following this is a brief discussion related to the different elements of the front-end of our system and how each of these components helps the user in understanding the application performance bottlenecks and the best suitable architecture. We then look at the prediction performance of our toolkit by running a real life application and providing step-by-step instructions on how to use the toolkit. Finally, in Chapter 8 we provide conclusions and discuss the various potential branches for future work.

Chapter 2

Related Work

This chapter discusses related work in the field of performance prediction and how it motivates and contributes to the overall development of this research. The following sections segregate the prediction models into one of the three categories: Analytical Models, Regression Models, and Models with GUI-based interface. Section 2.1 discusses some of the works that propose Analytical Models for performance prediction. These models utilize features from the architecture to develop a prediction model. In Section 2.2, the works focus on Regression based models that make use of Hardware Counters, discussed further in Chapter 5, to train their models. Finally, Section 2.3 deals with related works in the performance prediction domain that encapsulate their model in an easy to use GUI-based tool.

2.1 Analytical Models

The Roofline Model [25] and the Boat Hull Model [23] are two of the classical prediction models for predicting the performance of floating point applications on a multicore architecture. The work presented in the Boat Hull Model is an improvement over the Roofline Model. Both models only provides an upper limit for the run time prediction.

The Roofline Model focuses mainly on predicting the floating point performance on the following four multicore CPU architectures: Intel Xeon, AMD Opteron, Sun UltraSPARK and IBM Cell. The model considers architectural constraints and concentrates on providing an upper limit to the performance gains that can be obtained on the given architecture. For example, if knows the

maximum performance gain that can be obtained on the given architecture, the developer can focus on optimizations that will thereby get the most out of the architecture. The paper also discusses some of the fallacies in their model regarding not taking into consideration modern CPU features such as caches and prefetching. Thus, the Roofline Model makes a simplified yet very good attempt at floating point performance prediction but fails to provide additional detailed execution time information. Moreover, the model also does not support accelerator architectures such as GPUs and PHIs and hence limits its usage scope.

The Boat Hull Model modifies the Roofline Model to include additional information such as the class of the algorithm. The addition of this information allows the authors to fine-tune optimizations to the Roofline Model for that particular class. In addition to the latter changes, the authors also changed the prediction quantity from 'number of floating point operations' to 'execution time' thereby broadening the application of the model. The advantages of the Boat Hull Model over the Roofline Model are that it is straightforward to understand and requires little architectural information. The authors focus on predicting the performance of test applications on a Nvidia GTX 470 GPU, which is a key drawback in the lack of supported architectures.

Succeeding works presented in [9, 27, 15, 10, 13] take a slightly more complicated approach for performance prediction. From these works, we can conclude that there is a strong correlation between execution time and micro-architecture features such as instruction pipeline, memory access, thread count and memory bandwidth. Further, these features are some of the bottlenecks currently preventing applications from performing in an optimal manner. Thus understanding the relationship between them and the execution time, will aid in performance prediction. For example, if the major stake of an application runtime is dedicated to memory access then an architecture with faster memory access capabilities such as a CPU will be a much better choice than opting for an accelerator. Although these models perform better in terms of prediction accuracy than the Roofline Model and Boat Hull Model, the lack of multi-architecture support is a serious drawback.

2.2 Machine Learning Models

The Machine Learning Models rely on Hardware Counters, which will be discussed further in Chapter 6, that are provided as inputs to their prediction model/framework. Hardware Counters, as their name suggests, are counters that are part of the architecture and collect vital information

about the instructions executed. For example, a Floating Point Operations counter will increment itself everytime a floating point operation is performed.

The models proposed in [11, 8, 16, 18, 26] are based on taking an implementation for one architecture and porting it to another architecture. This approach is very much in line with our proposed work as even we believe that profiling application code through the use of hardware counters is one of the best and most efficient ways to capture and analyze application behavior.

The work proposed by Baldini [11] shows the use of a Machine Learning Classifier to determine if an application will experience an execution time benefit by running on a GPU. The model collects dynamic instruction profiles when running the default implementation and provides this information as input to the classifier. The profile consists of features based on computation instructions, loads, and code branching. With that said, the classifier has an error rate of 23% with drawbacks being the lack of support for accelerators other than GPUs and absence of an easy to use interface.

Similarly, the paper by Ardalani [8] focuses on predicting the actual execution time rather than classifying, thus making it a regression problem. The implementation starts with the collection of dynamic instruction profiles when running the default implementation and then provides it as input to the regressor. In contrast to what [11] considers as important features, this work takes into consideration features such as the mix of arithmetic operations, the working set size of the data and the number of concurrent operations. Moreover, while the Analytical Models focused on micro-architecture dependent features, this work uses micro-architecture independent features making the prediction more versatile across architectures. Finally, the proposed model has a prediction error of 36% and again lacks support for accelerators other than GPUs and an easy to use interface.

Following a similar approach, the models presented by Ipek [16] and Lee [18] perform performance prediction by providing the hardware counters as inputs to a Neural Network. From an execution time prediction standpoint, Neural Networks are better because of their robustness to noise in training data and fast evaluation of the learned target function [20]. Both of these properties help us build a model that can better capture target architecture features and at the same time create an inference model that can quickly output the prediction for a test application.

The model proposed by Ipek [16] uses machine learning to predict performance over a large multi-dimensional space defined by program inputs. The implementation starts by collecting a sample dataset of a collection of points spread evenly across parameter space. For each point,

they obtain the execution time on actual hardware and then divide the dataset into training and test sets. The training set is provided as input to a Multi-Layer Fully Connected Feed Forward Neural Network that is trained to reduce the percentage prediction error. Finally, the trained model is evaluated against the test set for which they report an error between 5% and 7%. The model is trained to predict the performance of parallel applications for multicore CPU architectures on Thunder [5] and BlueGene/L [1] systems at Lawrence Livermore National Laboratory. Thus the model does not attempt to predict the performance of accelerator architectures such as GPUs and PHIs, which would have made it more compelling. Finally, by not encompassing their work in an easy to use visual toolkit, it is difficult for an application developer to make use of their offerings.

The study by Lee [18] concentrates on the comparison between the use of Regression and Neural Networks for performance prediction. The authors show that Regression offers greater transparency and statistical understanding while Neural Networks offer greater usability, automation, and robustness to noise in training data. Similar to the model proposed in [16], the training set is provided as input to a Multi-Layer Fully Connected Feed Forward Neural Network using a Backpropagation algorithm and Gradient Descent for convergence. The model also adds a momentum factor to ensure faster Gradient Descent convergence. The percentage prediction error after running an inference on the test set offers comparable accuracy for Neural Network and Regression Models with the main difference being the interquartile range where we observe a greater spread in Regression compared to Neural Network. Again, the models focus primarily on predicting CPU performance and thus fails to support accelerator architectures.

2.3 GUI-Based Toolkits

HPCToolkit [7] is set of tools for measurement and analysis of applications on heterogeneous systems. It utilizes sampled values of timers and performance counters to measure an applications work, resource consumption, and bottlenecks and maps them to the calling context of their occurrence. HPCToolkit works with a wide variety of programming languages with support for fully optimized applications that are statistically or dynamically linked. The toolkit imposes very low overheads which help it to scale to large parallel systems. With the help of hpcviewer and hpctraceviewer, it is possible to visualize performance data in code-centric views and also analyze performance variation across threads. In addition, the visualization tool can also render trace views at multiple

levels of abstraction in a very short time.

The next tool is Intel Vtune Amplifier [19] which is a performance analysis tool for design and development of single and multithreaded applications. The tool helps developers analyze algorithms and provides tips to get the most out of the hardware. Moreover, it aids developers in determining the compute intensive parts of the code, unoptimized sections of the code, available synchronizations methods and also information on hardware related issues such as data sharing, cache misses, branch misprediction, etc.

Following this we have Vampir [22] which provides an easy to use framework for visualizing program behavior at different levels. Some of the major highlights of the tool are its ability to convert performance data into different performance views, support for navigation and zooming into the latter generated views and identifying the inefficient sections of the code. Another key feature of the tool is its ability to monitor accelerators such as GPUs and PHIs. The application trace collected by the toolkit provides CUDA and OpenCL support to understand runtime behavior of hardware accelerators.

The Nvidia Visual Profiler [3] is a cross-platform profiling toolkit for optimizing the performance of CUDA-based applications. It focuses on providing vital information such as performance bottlenecks in the applications in a neatly configured graphical view. Since CUDA activities are not just limited to the GPU, it is possible to have a unified timeline for the both the CPU and GPU which is very important in an accelerator based heterogeneous architecture where the CPU offloads its task to the GPU. The profiler also keeps tracks of memory transfers, kernel launches and other API functions which is useful when trying to understand application overheads. Finally, the guided application analysis feature provides a step-by-step analysis and optimization guidance. Through the use of powerful graphical visualizations, it clearly points out optimization opportunities in the application.

All the above-mentioned tools perform really well when it comes to analyzing the performance of the application for a given architecture. But as we can see, none of the tools focus on predicting the performance on a different architecture or discuss the potential of achieving better performance through the use of accelerators on a heterogeneous system. Moreover, not one tool is capable of replacing the other because each tool focuses only on one architecture and one programming language. While some tools like Vampir do support multiple programming languages, but it only supports one architecture. Finally, none of the tools give us access to the underlying hardware

counter information which is very important for fine tuning of the applications from a machine level perspective. Thus, it is the need of the hour to develop a tool that can fill in all the above-mentioned drawbacks in terms of being able to predict the performance of multiple platforms and support analysis of existing implementations for multiple architectures and programming languages.

2.4 Summary

In this chapter, an overview of the related work that lead to the motivation behind the thesis research is given. The chapter focuses on three different types of performance prediction and analysis toolkits based on analytical models, machine learning models and toolkits with a graphical user interface. For each of the above mentioned models and toolkits the chapter describes the advantages and disadvantages to further improve our tool.

Chapter 3

X-MAP Architecture

3.1 Architecture

In this Chapter, we will discuss the architecture of our toolkit, which is comprised of a front-end and a back-end. The back-end is responsible for profiling the application and predicting the execution runtime and best architecture using the Quantitative and Qualitative Model respectively. On the other hand, the front-end is responsible for the user level interaction such as loading and editing of source code and visualization of the prediction results.

As shown in Figure 3.1 the user first starts by loading the source code into the toolkit, which is represented by the Load Source Code block. The source code is then forwarded to the back-end of our toolkit, wherein it is sent to the Application Profiling block. The Application Profiling engine is responsible for profiling the source code using either Performance API (PAPI) or CUDA Profiling Tools Interface (CUPTI) depending on the source code programming language.

Alternatively, the user also has the option to directly upload application profiling data, which is represented by the Upload Profile block. This option lets the developer use an existing profile without the need for the source code.

Following, the application profile data collected using either of the above mentioned means is provided as input to the Qualitative and Quantitative prediction models. The Qualitative Model is a Random Forest Classifier and is responsible for predicting the best architecture, programming language and the combination of the two. On the contrary, the Quantitative Model is responsible for predicting the execution time of the application on each of the supported platforms.

The outputs for the above mentioned models are prediction confidence in percentage and execution time in seconds, which are pushed back to the front-end of the X-MAP architecture. In the front-end, the data is parsed to the visualizer, which generates graphs to visualize the prediction confidence and the execution time. The three blocks, Architecture Prediction, Programming Language Prediction, and Platform Prediction utilize the prediction confidence information and plot bar charts of percentage confidence for best architecture, programming language and platform respectively. Similarly, the Execution Time Prediction block utilizes the execution time data and plots bar chart for execution time (in seconds) on different platforms.

Finally, the Hardware Counter Visualization block takes the application profile as input and visualizes them using graphs for developers to analyze the application. The analysis includes functional breakdown to identify compute intensive parts of the program and visualization of the hardware counters to understand program behavior on the machine level.

3.2 Summary

In this chapter, the underlying software architecture for the toolkit is described. The architecture consists of a front-end, which is in-charge of user actions and visualization of prediction data. On the contrary, the back-end of the system is responsible for profiling of source code and running inferences using the prediction models to predict the best architecture, programming language, platform and execution time.

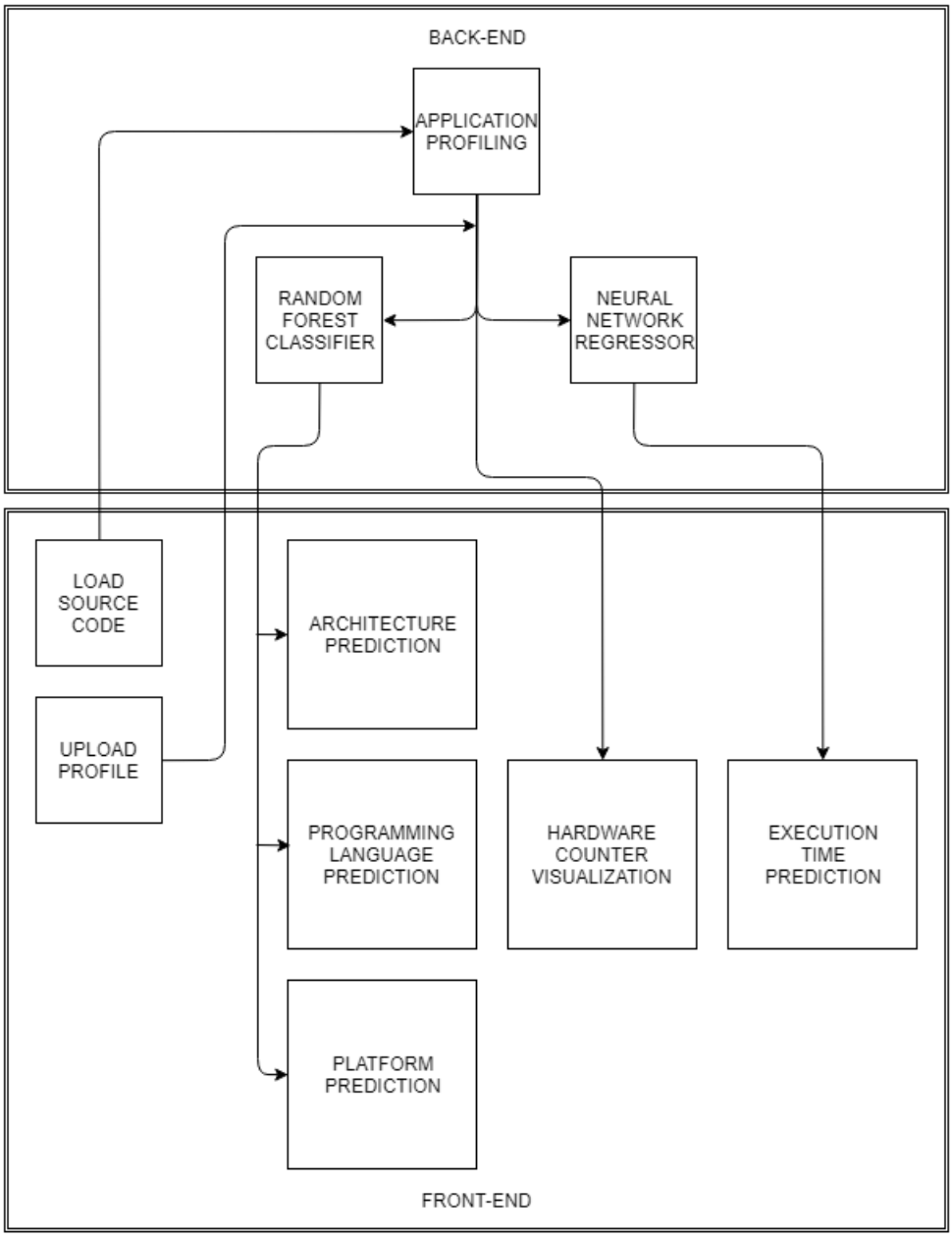

Figure 3.1: Toolkit Architecture

Chapter 4

Back-end Implementation

The back-end of our performance prediction tool consists of a Qualitative and a Quantitative Model. The Qualitative Model is a Random Forest Classifier, which is used to decide the best architecture to obtain maximum application speedup. On the other hand, the Quantitative Model is a Neural Network Regressor that predicts the execution time for the application on a given architecture. The following sections present a brief description of our models but for more detailed information please refer to our project website [4].

4.1 Hardware Counters

Hardware Counters are a set of special purpose registers, which are part of the microprocessor architecture, that store counts of hardware-related activities during computer software execution. These counters play an important role in performance analysis, application tuning and performance prediction. The number of Hardware Counters available on an architecture is limited and also varies from architecture to architecture. Moreover, the number of events that we intend to measure is significantly more than the number of counters and hence we use implementations such as Performance API [21] and CUDA Profiling Tools Interface [2] which program the counter with the index of the event we want to monitor.

The advantages of using Hardware Counters over software techniques is that they provide access to a wealth of detailed performance information related to architecture features such as functional units, caches, and main memory with a very low overhead. Secondary benefits of using

Hardware Counters over software techniques include no source code modification and ease of use. On the other hand, Hardware Counters traditionally face problems such as being unable to correlate the event metrics back to source code and a limited number of these counters. Moreover, with out-of-order scheduling and execution of multiple instructions, the values present in the Hardware Counters may retire at any moment in time depending on memory access, cache hits, and stalling of the instruction pipeline. This behavior can lead to assignment of Hardware Counters events to the wrong instruction, making their values invalid. These drawbacks have been recently overcome with the introduction of Instruction Based Sampling [14] which helps in per task data association.

4.1.1 PAPI : Performance API

Performance API (PAPI) [21] provides a standard application programming interface for accessing Hardware Counters available on most modern microprocessors. The implementation involves two interfaces to the underlying Hardware Counters with the first being a simple high-level interface for acquiring simple performance measurements and the second being a fully customizable low-level interface for more in-depth requirements.

In addition to ease-of-use, PAPI also provides portability across different platforms. Since it uses the same routines to control and access the Hardware Counters, a lowest common denominator set of events has been predefined to ensure support for different platforms. When using these standardized events, the developer only has to perform compilation and linking with no source code modifications. In a case of inconsistency where a particular event does not exist on the current platform, appropriate error codes are generated which significantly reduces the porting effort for applications using PAPI.

From a software standpoint, PAPI can again be divided into two layers. The upper layer us made up of the API and the machine independent support functions. The lower layer provides a machine independent environment to the architecture dependent functions and data structures. These functions access the underlying operating system, kernel extensions, or the assembly functions to access the counter registers. Based on the platform, PAPI utilizes the most efficient of the latter three options.

One more useful feature provided by PAPI is counter overflow prevention for both the underlying Operating System and the libraries. Since each Hardware Counter can be incremented multiple times in a single clock cycle and due to increasing clock speeds, it is highly likely for the

counters to overflow and hence having the ability to prevent this is a very useful feature. Additional features include portable implementation of asynchronous notifications when counter values exceed a threshold set by the user. This feature is important for generating accurate histograms of performance interrupts based on hardware metrics and provides a solid base for all line-level performance analysis software.

A list of available PAPI Event Counters and a brief description for each of them is provided in Appendix A.

4.1.2 CUPTI : CUDA Profiling Tools Interface

The NVIDIA CUDA Profiling Tools Interface (CUPTI) [2] provides performance analysis tools with detailed information about application behavior running on NVIDIA GPUs. It introduces powerful mechanisms to enable performance analysis tools such as the NVIDIA Visual Profiler (Nvprof) [3], TAU [12], and Vampir Trace [22] to interact with Hardware Counters on NVIDIA GPUs and provide valuable information to an application developer. For the creation of profiling and tracing tools, CUPTI provides the following four APIs:

1. Activity API: The Activity API allows for asynchronous trace collection of an application's CPU and GPU activities. The activities are reported in the form of data structures for each activity. The API then stores these data structures in a buffer, which is transferred to the client on a time to time basis.

2. Callback API: The Callback API allows users to register a callback to the application code. This callback is invoked when the profiled application calls CUDA runtime or the driver functions. Each callback function is given a unique ID and is grouped into one of the four available domains for an easy association.

3. Event API: The Event API allows the user to query, configure, start, stop and read the hardware counters on the device. Like PAPI, even CUPTI refers to performance counters as events that are defined as a countable activity occurring on the device. Each event is given a unique ID and placed into one of the categories. The categories are a subset of a larger set of domains, which are subsets of groups and group set respectively. This type of grouping lets users analyze their program from different levels of sophistication.

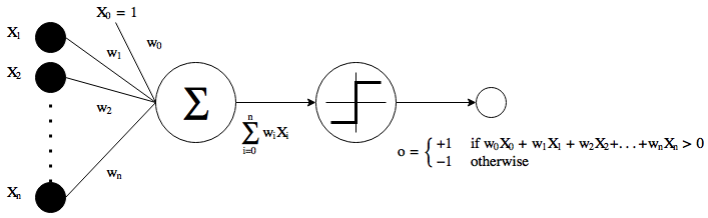

Figure 4.1: Perceptron

4. Metric API: The Metric API is responsible for the collection of application metrics calculated from one or more event values and thus can be referred to as derived performance counters. Similar to event counters, even metric counters are grouped together into categories that indicate the type of characteristic measured by the metric.

A list of available CUPTI Event Counters and a brief description for each of them is provided in Appendix B.

4.2 Neural Network Regression

Artificial Neural Networks are a useful method for learning real-valued, discrete-valued, and vector-valued functions from training samples. It is appropriate for problems with the following characteristics:

1. Samples are represented as attribute-value pairs
2. The output function is discrete-valued, real-valued, or a vector of the latter two classes
3. The training dataset contains errors
4. Long training times are acceptable
5. Faster inference of the learned output function is required
6. The level of abstraction is unimportant

The problem of predicting the run-time using Hardware Counters satisfies all the above criteria and hence is a very good scenario for using Neural Network Regression.

We begin by first understanding the concept of Perceptron, which is the basic building block for a Neural Network. As shown in Figure 4.1 Perceptron takes a vector of real-valued inputs,

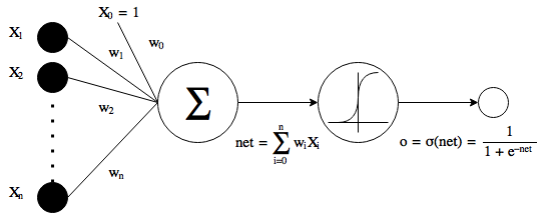

Figure 4.2: Sigmoid threshold unit

calculates a linear combination of these inputs and then outputs +1 if the result is greater than a threshold and -1 otherwise. Mathematically, it can be represented as follows [20]:

$$o(x1, ..., xn) = \begin{cases} +1, & \text{if } w_0 + w_1x_1 + w_2x_2 + ... + w_nx_n > 1 \\ -1, & \text{otherwise} \end{cases} \tag{4.1}$$

The next step for the Perceptron is to learn the values of the weights $w_0, ..., w_n$ for which we will utilize the Gradient Descent and the Delta Rule. The key idea behind the use of Delta Rule is to use Gradient Descent to search for the possible values of weights that will best fit the training dataset. Moreover, if the training dataset is not linearly separable, the Delta Rule converges towards the best approximation to the output function.

The Gradient Descent Rule provides the substrate for the Backpropagation algorithm, which is known to learn networks with many interconnected Perceptron units. This quality is important because single Perceptrons can only represent linear data space, which does not meet our requirements. Therefore, we must use Multilayer networks trained with the Backpropagation algorithm, which can represent nonlinear data space.

We know that multiple layers of cascaded linear units produce a linear output and since we prefer networks generating nonlinear functions, we need a unit whose output is both differential and a nonlinear function of its input. The solution is to use sigmoid as our threshold function resulting in a Perceptron unit as shown in Figure 4.2.

We now focus on the learning algorithm, which in our case is the Backpropagation algorithm. This algorithm learns the weights of a Multilayer layer network with a fixed set of fundamental units and interconnections. It uses Gradient Descent to minimize the squared error between the network output and the actual value for the output. The weight update is iterated thousands of times and a variety of termination conditions might be used to stop the training process. The choice of

termination condition is important because if the number of iterations is too few then we might not converge and on the other hand, if the number of iterations is too many then it can lead to overfitting.

Moreover, we know that our plane of convergence consists of many local minima and since we want to terminate at the best possible solution, we add a small momentum factor to our weight update rule. The added momentum keeps the ball rolling through small local minima or along flat regions in the surface where the ball would stop. At the same time, it also gradually increases the step size of the weight update, thereby speeding up convergence.

4.2.1 TensorFlow Neural Network Regressor

We implement our Neural Network Regression model in Tensorflow [6], which is an interface for expressing and executing machine learning algorithms. It has support for a wide variety of heterogeneous systems ranging from mobile devices to compute clusters. TensorFlow is highly flexible in the sense that it can be used to train and run inferences on algorithms for Neural Networks and is widely used in fields such as speech recognition, computer vision, robotics, information retrieval, natural language processing and geographic information extraction.

A TensorFlow system comprises of a client that uses a Session interface to communicate with the master and one or more workers. The workers, in this case, refers to computational devices such as CPU cores and GPU devices. TensorFlow also supports distributed systems and thus has a local implementation for single-node systems and a distributed implementation for multi-node systems.

The fundamental element used to build a TensorFlow network is called a Tensor: a data structure including signed and unsigned integers ranging from 8 bits to 64 bits, floats, doubles, complex numbers, and strings.

We implement our regressor using Tensorflow's built-in functions for a neural network regressor and further modify the code to obtain better prediction results. Our network consists of three hidden layers in addition to the input and the output layer. The number of nodes in the input layer is variable and depends on the number of hardware counters that are highly correlated to run time prediction. The output layer has only one node that outputs the predicted run time. For the number of nodes in the hidden layers, we start with twice the number of input nodes and then fine tune it by performing a series of experiments to verify which configuration gives the best prediction accuracy. For example, when training with hardware counters collected from a serial C

implementation, the number of nodes in the hidden layers are 20, 15 and 5 respectively. Moreover, Table 4.1 gives information on our selection of hyperparameters for our network which we selected through experimentation.

Hyperparameter	**Value**
Learning Rate	0.001
Loss Function	MSE
Batch Size	3
Training Iteration	1200
Momentum	0.5
Weight Decay	0.1
Dropout	0

Table 4.1: Hyperparameter values of the neural network for our quantitative model.

4.3 Random Forest Classification

4.3.1 Decision Tree Classifier

Decision tree learning is an approximation technique for discrete-valued output functions where the trained function is represented as a decision tree. This type of training method is a popular inductive inference algorithm and can be used in areas such as diagnosis of medical cases to assessing the credit risk of a loan application. A Decision Tree Classifier performs exceptionally in cases where the hypothesis space has limited outputs as opposed to our previous model where the hypothesis space was infinite. Therefore a Decision Tree Classifier is appropriate for problems with the following characteristics:

1. Samples are represented as attribute-value pairs
2. The output function is discrete-valued
3. The target outputs are disjunctive classes
4. The training dataset contains errors
5. The training data has missing attribute-value pairs

Thus, our problem of predicting the best architecture and programming language using Hardware Counters satisfies all the above criteria and hence is a very good use case for a Decision

Tree Classifier.

A Decision Tree Classifier starts with the construction of the actual tree for the attributes. The root attributes are chosen by means of a statistical test to determine its classification accuracy for all the training samples. The child for the root node is created for each possible value of the root attribute and the training examples are sorted down to the branch. The entire process is repeated for all of the training samples similar to a greedy search algorithm. The end result is a Decision Tree Classifier, which based on the attributes at different levels of the tree, classifies an inference into one of the many output classes.

4.3.2 Random Forest Classifier

While Decision Tree Classifiers perform very well in most use case scenarios they are plagued with the following drawbacks:

1. Determining the depth of the tree
2. Handling training dataset with continuous attributes
3. Selecting attributes with different training errors
4. Handling data with missing attributes

While some of these problems can be eliminated by techniques such as pruning, some other serious drawbacks such as overfitting cannot be avoided. Hence it is important to find an alternative solution to better suit our classification needs.

We thus propose the use of a Random Forest Classifier, which is an extension to the Decision Tree Classifier with the following features:

1. Excellent accuracy
2. Efficient operation on large datasets
3. Gives a numerical estimation of what variables have high correlation
4. Estimates missing data and maintains accuracy when large quantities of data are missing
5. Avoids overfitting

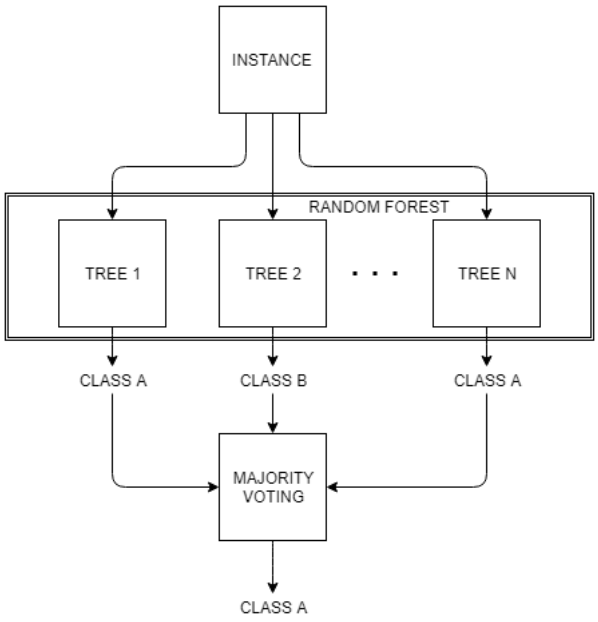

Figure 4.3: Random Forest

The function of a Random Forest Classifier is similar to a Decision Tree Classifier with the only difference being that Random Forest grows many classification trees. Each tree thus gives a classification and then the forest, which is a collection of the above trees, selects the class with the most votes. Figure 4.3 shows the function in a diagrammatic representation.

4.3.3 Scikit Learn Random Forest Classifier

The Scikit Learn Random Forest Classifier [24] is a meta-estimator that trains a number of Decision Tree Classifiers on various subsets of training samples from the original training dataset and performs averaging to reduce training error and control overfitting. The size of the subset of training samples are always same as the original input sample size but are drawn with replacement.

For our Qualitative Model, the inputs to our Random Forest Classifier are the hardware counters and the output is the probability with which our classifier classifies the input into each of the output bins. The output bins could either be the four architectures, four programming languages, or the ten implementations depending on our classification requirements. We finally select the bin with the highest prediction confidence. Finally, for the hyperparameters themselves, we select our number of trees to be equal to 1500, which gives us the best prediction accuracy with high efficiency.

4.4 Summary

In this chapter, we describe the back-end of the system, which comprises of a Qualitative Model based on a Random Forest Classifier and a Quantitative Model based on a Neural Network Regressor. The Random Forest Classifier is implemented using Scikit Learn and takes in Hardware Counters as input and provides with architecture, programming language or platform as output. On the other hand, the Neural Network Regressor is implemented using TensorFlow and takes in Hardware Counters as input and provides execution time on different platforms as output.

Chapter 5

Front-end Implementation

In this Chapter, we will take a look at the Front-end of our toolkit and discuss in detail the various elements used in building our tool. Figure 5.1 showcases the application window the user is presented on launching the toolkit. The Front-end was designed using QT, which is a C++ based GUI development environment and consists of the following elements:

1. Text Editor
2. Buttons
3. Dropdown Menus
4. Tables
5. Graphs

In the following sections, we will discuss each of the elements in detail and understand the various interactions between them to synchronize the back-end of our toolkit with the elements in the front-end.

5.1 Text Editor

Figure 5.2 showcases the text editor where the user is able to write and edit his/her code prior to running the performance prediction models. The editor encompasses a Syntax Highlighter application that displays C and C++ files with custom syntax highlighting. The advantage of

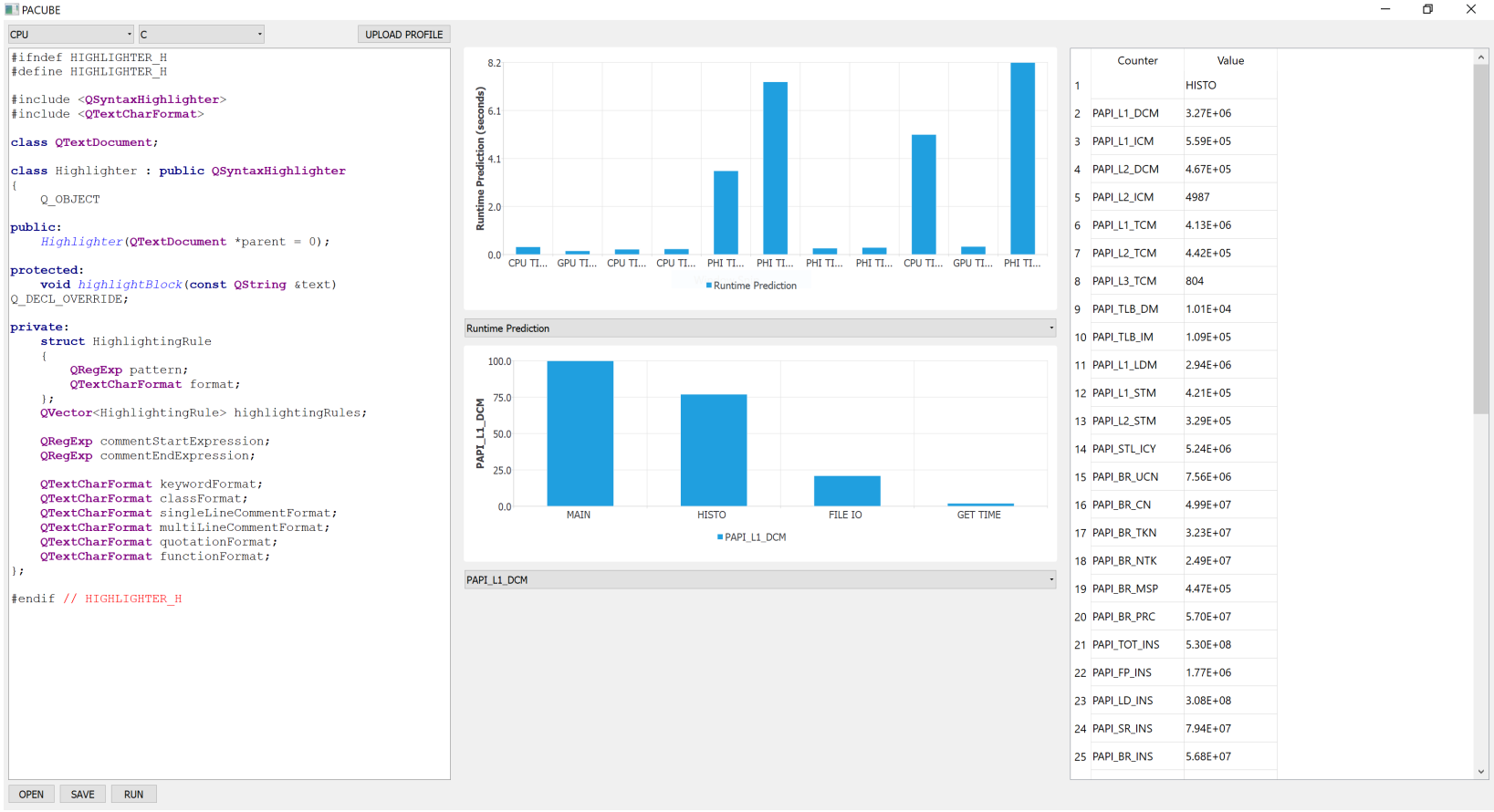

Figure 5.1: Main application window for the toolkit showcasing the various building blocks

```
#include <algorithm>
#include <cmath>
#include <iostream>

#include "cross_platform/GLUT.hpp"

#include "Height.hpp"
#include "Ocean.hpp"

/*
Initializes the variables and allocates space for the
vectors.
*/
Ocean::Ocean(const double p_lx, const double p_ly, const
int p_nx, const int p_ny, const double p_motion_factor)
:
    lx(p_lx),
    ly(p_ly),
    nx(p_nx),
    ny(p_ny),
    motion_factor(p_motion_factor) {
    height0I.resize(nx+1);
    height0R.resize(nx+1);
    HR.resize(nx+1);
    HI.resize(nx+1);
    hr.resize(ny+1);
    hi.resize(ny+1);
    for(vec_vec_d_it it=HR.begin() ; it!=HR.end() ; it+
+) it->resize(ny+1);
    for(vec_vec_d_it it=HI.begin() ; it!=HI.end() ; it+
+) it->resize(ny+1);
    for(vec_vec_d_it it=hr.begin() ; it!=hr.end() ; it+
+) it->resize(nx+1);
    for(vec_vec_d_it it=hi.begin() ; it!=hi.end() ; it+
+) it->resize(nx+1);
    ffty.reserve(nx);
    fftx.reserve(ny);
    for(int i=0 ; i<nx ; i++) ffty.push_back(new FFT(ny,
&HR[i], &HI[i]));
    for(int i=0 ; i<ny ; i++) fftx.push_back(new FFT(nx,
&hr[i], &hi[i]));
}

/*
Free memory.
*/
Ocean::~Ocean() {
    for(int i=0 ; i<nx ; i++) delete ffty[i];
    for(int i=0 ; i<ny ; i++) delete fftx[i];
}

/*
```

Figure 5.2: Text editor

using the built in the text editor is that the user does not have to exit the tool to edit the code, which is extremely useful when making minor changes such as syntax correction and indentation. Additionally, since the editor has all of the modern features present in most modern text editors, the user will find it compelling to use the included editor.

5.2 Buttons

The buttons in the tool initiate various actions responsible for triggering the backend for our application. We have four buttons Open, Run, Save and Upload and we will discuss the functionality provided by each of them in the following subsections.

5.2.1 Open

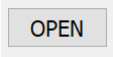

Figure 5.3: Open button

The Open button is shown in Figure 5.3 and is responsible for opening code files into the text editor. It opens a dialog window as shown in Figure 5.4 that allows the user to select C and C++ code files to view and edit in the built-in text editor.

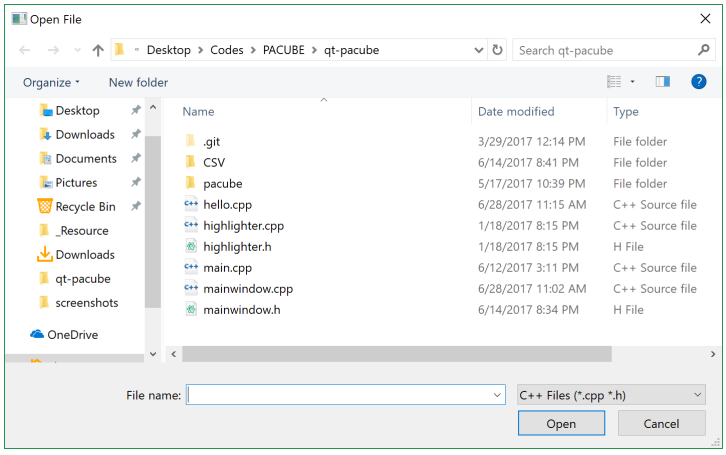

Figure 5.4: Dialog to allow users to open the source code file

5.2.2 Save

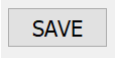

Figure 5.5: Save button

Figure 5.5 shows the Save button which is used to save the current text editor buffer to a file whose name and save location is decided by the user by typing it into the popup dialog box shown in Figure 5.6.

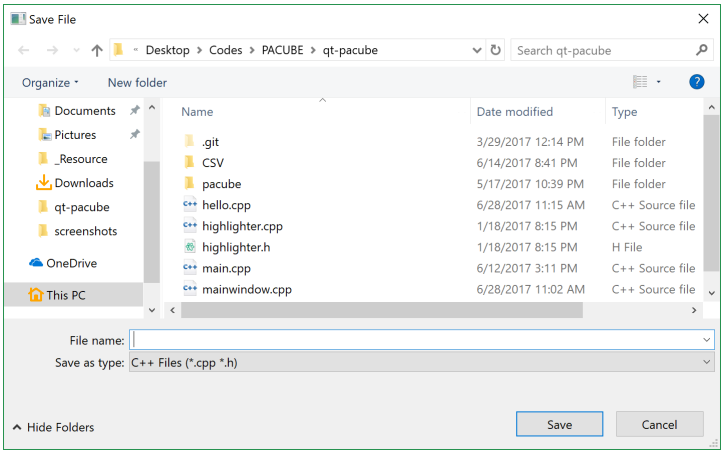

Figure 5.6: Dialog to allow users to save the source code file

5.2.3 Upload

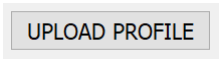

Figure 5.7: Upload button

The Upload button, shown in Figure 5.7 is used to load pre-profiled application performance profiles that contain hardware counter information. This feature saves time and effort related to application profiling, which can contribute a significant amount of our toolkit runtime.

5.2.4 Run

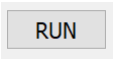

Figure 5.8: Run button

The Run button shown in Figure 5.8, as the name suggests, triggers the start of our back-end, which involves running the application profiler, qualitative model, quantitative model and generates the visualization for our performance predictions.

5.3 Dropdown Menus

The Dropdown Menus are responsible for setting the various model variables and also to view specific visualization from the large selection of the graphs that are generated by the prediction model. The toolkit encompasses the following four Dropdown Menus and will discuss the functionality for each of them in their corresponding subsections:

1. Architecture Selection
2. Programming Language Selection
3. Prediction Quantity Selection
4. Hardware Counter Selection

5.3.1 Architecture Selection

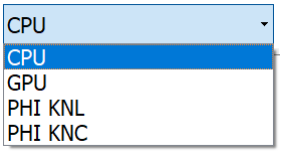

Figure 5.9: Architecture selection dropdown menu

The Architecture Selection Dropdown Menu, as shown in Figure 5.9, is responsible for letting the user select the target architecture for the code in the text editor or the uploaded application

profile. Currently, our toolkit supports four main architectures as listed below:

1. CPU - Intel Broadwell
2. GPU - Nvidia Kepler
3. PHI - Knights Corner (KNC)
4. PHI - Knights Landing (KNL)

5.3.2 Programming Language Selection

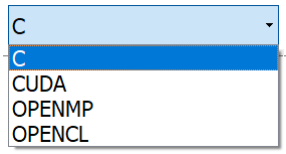

Figure 5.10: Programming language selection dropdown menu

The next Dropdown Menu is the Programming Language Selection Menu shown in Figure 5.10, which is used to set the programming language of the code in the text editor or the uploaded application profile. Again, our toolkit supports the following four major programming languages:

1. C
2. CUDA
3. OpenMP
4. OpenCL

5.3.3 Prediction Quantity Selection

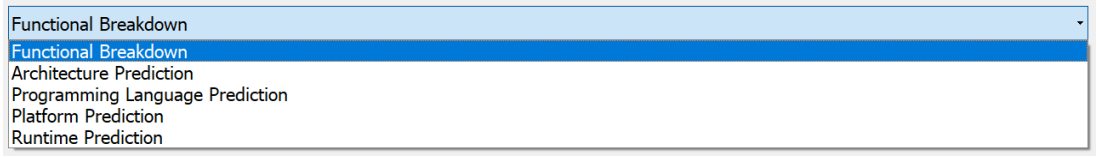

Figure 5.11: Prediction quantity selection dropdown menu

After running the prediction model in the background, we are able to generate a variety of predictions and their corresponding visualizations. Since presenting all graphs in the same window is difficult, we present the user the with the Prediction Quantity Selection Dropdown Menu, as shown in Figure 5.11, to toggle the visibility of the required graph. The Dropdown Menu consists of the following selections and the description for each is provided in the Graphs Section of this Chapter:

1. Functional Breakdown
2. Architecture Prediction
3. Programming Language Prediction
4. Platform Prediction
5. Runtime Prediction

5.3.4 Hardware Counter Selection

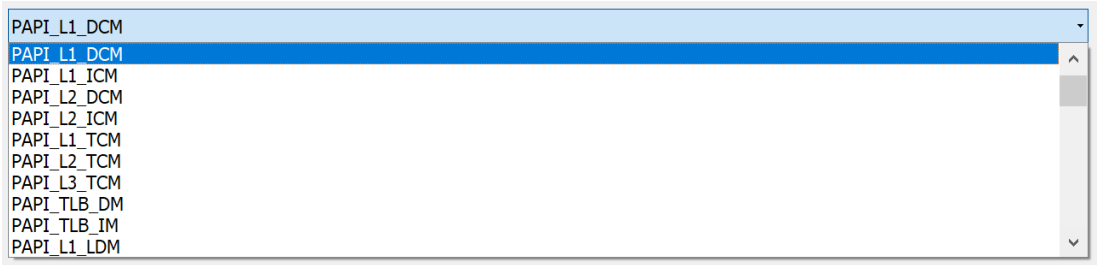

Figure 5.12: Hardware counter selection dropdown menu

The Hardware Counter Selection Dropdown Menu lets the user select the Hardware Counter that he/she wishes to visualize. This option lets the application developer look at the machine level optimization opportunities. Moreover, by visualizing the various hardware counters, it is possible to find architectural bottlenecks and in turn select a better performing architecture to suit our needs. As shown in Figure 5.12 the Dropdown Menu consists of a list of all the available Hardware Counters collected when profiling the application.

5.4 Table

The Hardware Counter Table in Figure 5.13 presents the data collected by profiling the application in an orderly manner. This format is important because data generated from the profiles can be intimidating to the common user and thus by presenting only the valuable information the developer can ultimately be more productive. We also present this data in the form of graphs but having a numerical form of output is helpful when comparing two closely valued Hardware Counters.

5.5 Graphs

The toolkit provides visualization for all predictions performed by the qualitative and quantitative models. In addition to the predictions, the toolkit also helps the developer visualize key aspects of the application such as Functional Breakdown, which is very important to understand how much the application spends in each part of the program and thereby narrow down bottlenecks. In this case, almost 75% of the runtime is being spent in the HISTO function can be subjected to speedup improvements. Moreover, by being able to visualize the Hardware Counters collected during the profiling stage, it is possible to scrutinize the code on the machine level and make better optimizations.

	Counter	Value
1		HISTO
2	PAPI_L1_DCM	3.27E+06
3	PAPI_L1_ICM	5.59E+05
4	PAPI_L2_DCM	4.67E+05
5	PAPI_L2_ICM	4987
6	PAPI_L1_TCM	4.13E+06
7	PAPI_L2_TCM	4.42E+05
8	PAPI_L3_TCM	804
9	PAPI_TLB_DM	1.01E+04
10	PAPI_TLB_IM	1.09E+05
11	PAPI_L1_LDM	2.94E+06
12	PAPI_L1_STM	4.21E+05
13	PAPI_L2_STM	3.29E+05
14	PAPI_STL_ICY	5.24E+06
15	PAPI_BR_UCN	7.56E+06
16	PAPI_BR_CN	4.99E+07
17	PAPI_BR_TKN	3.23E+07
18	PAPI_BR_NTK	2.49E+07
19	PAPI_BR_MSP	4.47E+05
20	PAPI_BR_PRC	5.70E+07
21	PAPI_TOT_INS	5.30E+08
22	PAPI_FP_INS	1.77E+06
23	PAPI_LD_INS	3.08E+08
24	PAPI_SR_INS	7.94E+07
25	PAPI_BR_INS	5.68E+07

Figure 5.13: Output of application profile in tabular format

5.5.1 Functional Breakdown Graph

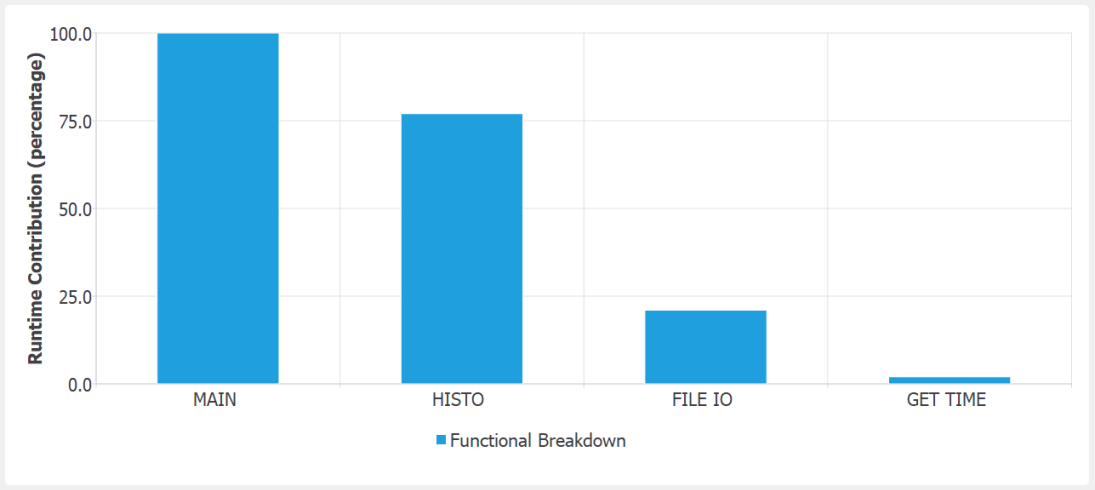

Figure 5.14: Functional breakdown of the execution time for the application

The Functional Breakdown graph as shown in Figure 5.14 provides an execution time breakdown of the different functions of the application. The information is conveyed in the form of a bar chart that helps the developer understand how much time is spent on each part of the program and thus develop appropriate strategies to further optimize the program.

5.5.2 Architecture Prediction Graph

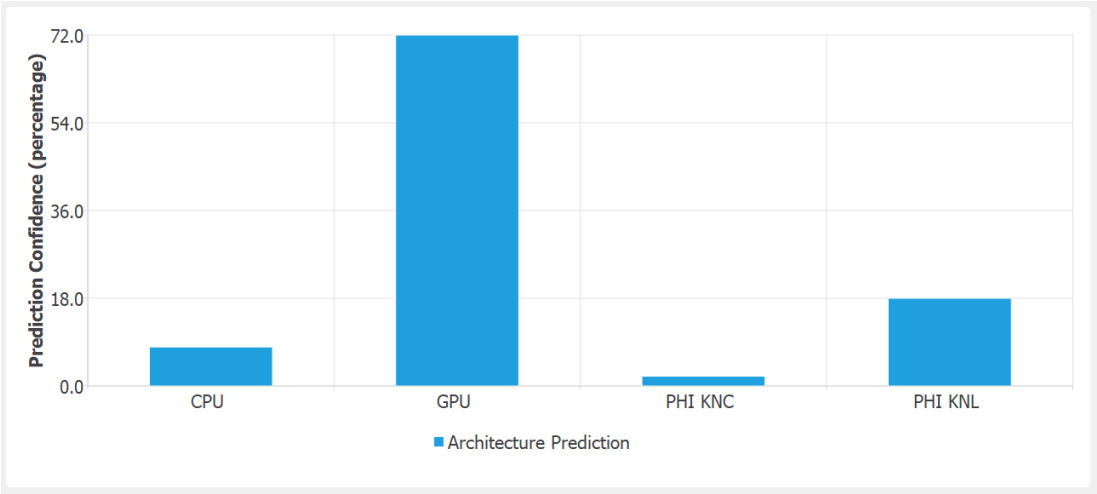

Figure 5.15: Percentage confidence for architecture prediction

The Architecture Prediction graph in Figure 5.15 visualizes the output of the Qualitative Model responsible for predicting the best architecture for the application under study. As mentioned previously, we currently support four different architectures and thus our Architecture Prediction graph provides the confidence level for the architecture prediction results.

5.5.3 Programming Language Prediction Graph

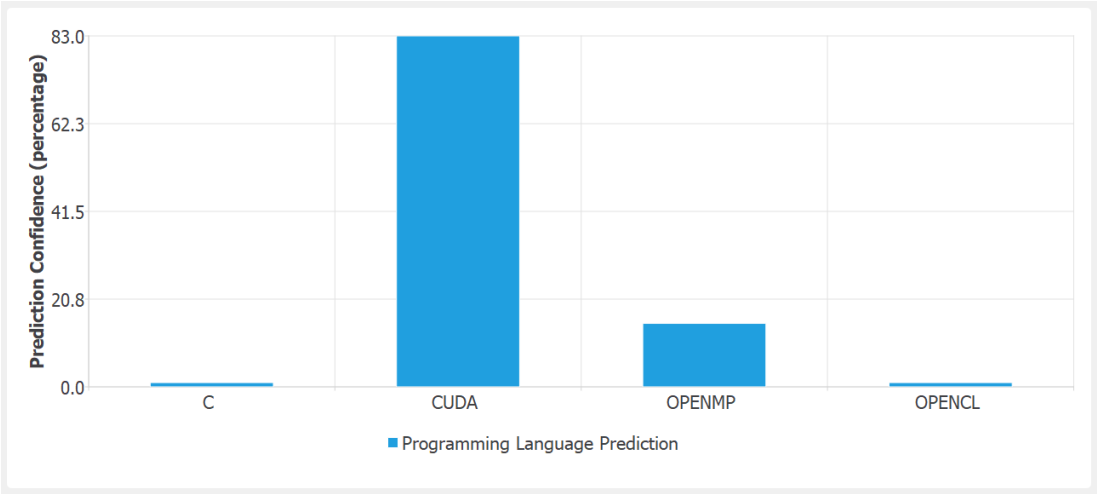

Figure 5.16: Percentage confidence for programming language prediction

Similar to the Architecture Prediction graph, the Programming Language Prediction graph in Figure 5.16 suggests the best programming language for the application under study. We again support four programming languages and hence provide the confidence level for the model suggestions.

5.5.4 Platform Prediction Graph

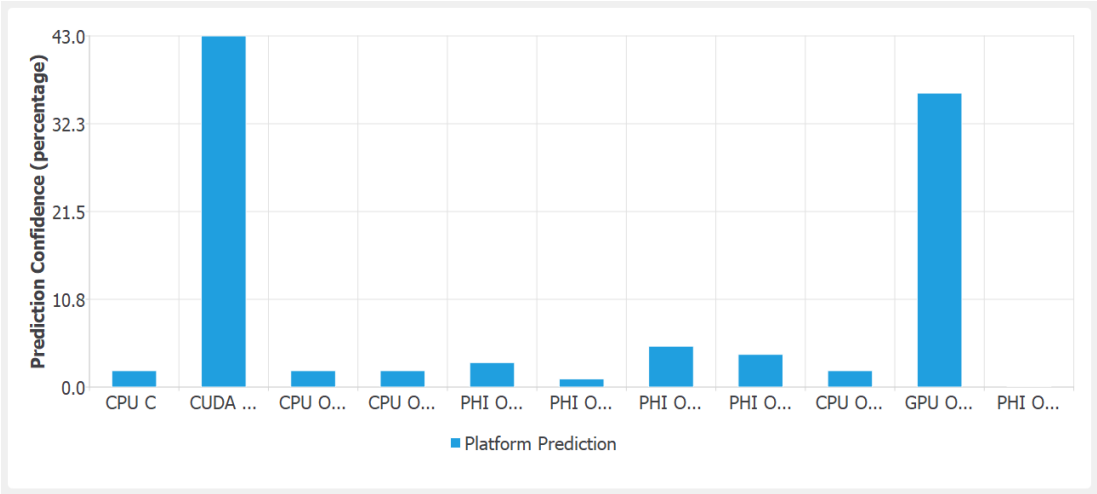

Figure 5.17: Percentage confidence for platform prediction

The Platform Prediction graph in Figure 5.17 is a combination of the above two graphs and predicts the best platform, a combination of architecture and programming language, for the application. We have eleven different platforms rising from the combination of previously mentioned architectures and programming languages and thus the graph represents the prediction confidence for each of the total platforms.

5.5.5 Runtime Prediction Graph

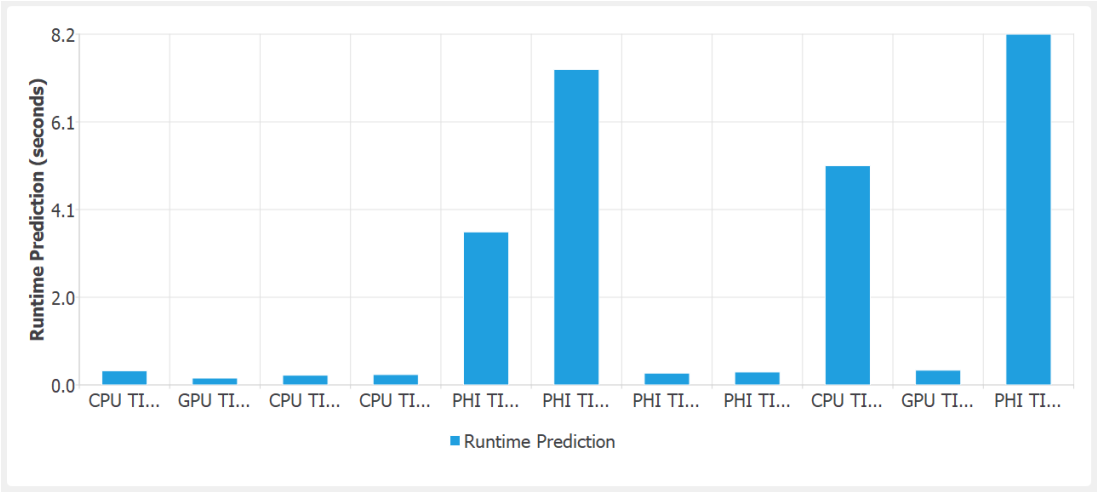

Figure 5.18: Predicted execution time for each platform

The Runtime Prediction graph shown in Figure 5.18 presents the output of the Quantitative Model, which is predicts the execution time of the application on each of the supported platforms. On the X axis there are eleven different platforms while the Y axis indicates the time in seconds. These graphs show the predicted total runtime for the application on each of the platforms.

5.5.6 Hardware Counter Graphs

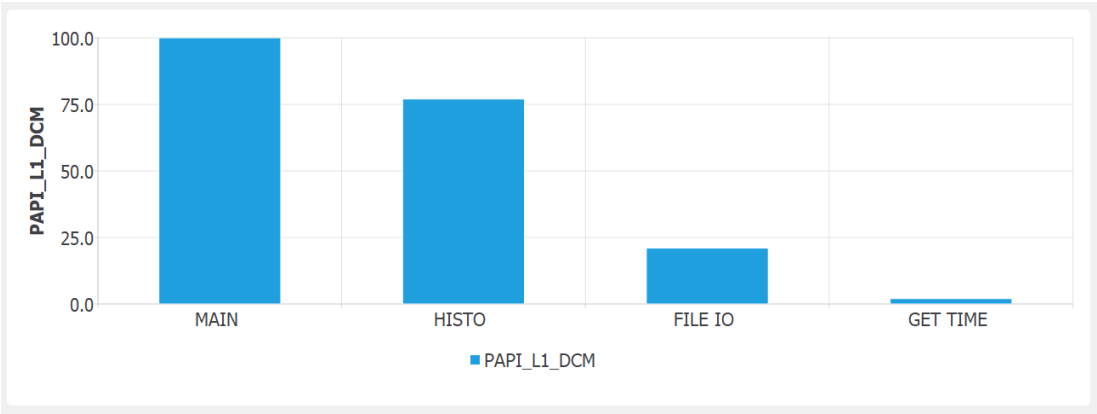

Figure 5.19: Functional breakdown of the hardware counters for the application

The Hardware Counter graph shown in Figure 5.19 is a function-wise visualization of the Hardware Counter value selected using the Hardware Counter Selection Dropdown Menu discussed earlier. This view allows the developer to explore the correlation between the Hardware Counters and the contribution of the individual functions to the total values and with this knowledge the user can make fine-tuned optimizations to gain maximum application performance.

5.6 Summary

In this chapter, we focus on the the front-end of the system consisting of a number of visualization components that can be used to view graphs for functional breakdown, prediction results and the hardware counters. Additionally, the GUI also comprises a number of buttons and dropdown menus which provide the users with a wide variety of programming language and architecture options. Finally, the text editor with its ability to highlight code syntax forms a vital part of the system front-end.

Chapter 6

Results and Analysis

In this Chapter, we will discuss the prediction accuracy of the toolkit from a developer's standpoint. We will test our results using a real world application called Lulesh [17], which simulates the motion of materials relative to each other when subject to forces. The application has a highly optimized version of the code for all of our supported platforms and thus provides us with across the board base results for prediction comparisons. Moreover, we will also analyze the prediction results for a Histogram application, which focuses on predicting the execution time for different input sizes.

6.1 Performance Analysis Results

For the demonstration, we will start with the analysis of the CPU version of Lulesh and profile it to obtain the Hardware Counters that can then be used as input to the prediction models. Using the Open button on our toolkit we first load the code into our Text Editor and begin our one step prediction process by pressing the Run button. If we wish to profile the application on a different platform, we can use the Upload Profile button to load the profile data and then perform prediction. Nonetheless, after pressing the Run button, we see that the Hardware Counter table is populated and so are the visualizations for the different predictions.

The first graph we are interested in is the Functional Breakdown graph, which is shown in Figure 6.1. According to the breakdown, we see that our application spends the most time on CalcKinematicsForElems(), followed by CalcMonotonicQGradientsForElems(), and CalcFBHourglassForceForElems(). Thus, if the developer wants to focus on optimizing a particular function

then these three functions should be on the priority list to achieve maximum speedup.

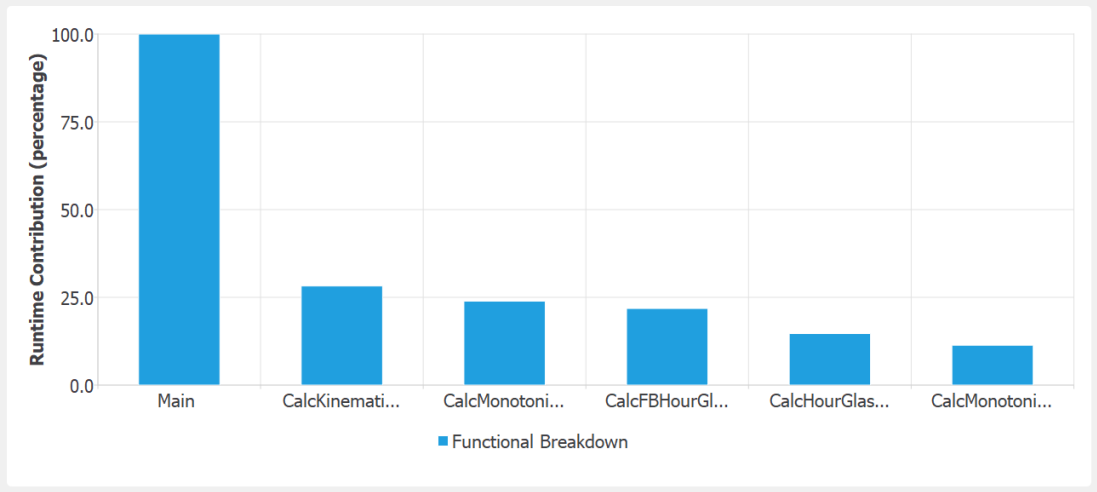

Figure 6.1: Functional breakdown of the execution time for Lulesh

6.2 Qualitative Model Results

The next graph is the Architecture Prediction graph shown in Figure 6.2 where the best architecture for the application at hand is the GPU architecture. The second best architecture according to our Qualitative Models is PHI-KNL, which appears to provide a significantly large speedup over its predecessor, the PHI-KNC architecture. Thus the developer has the option to select the GPU architecture for maximum application speedup or go with PHI-KNL, which has slightly reduced speedup but more CPU-like characteristics in terms of programming language.

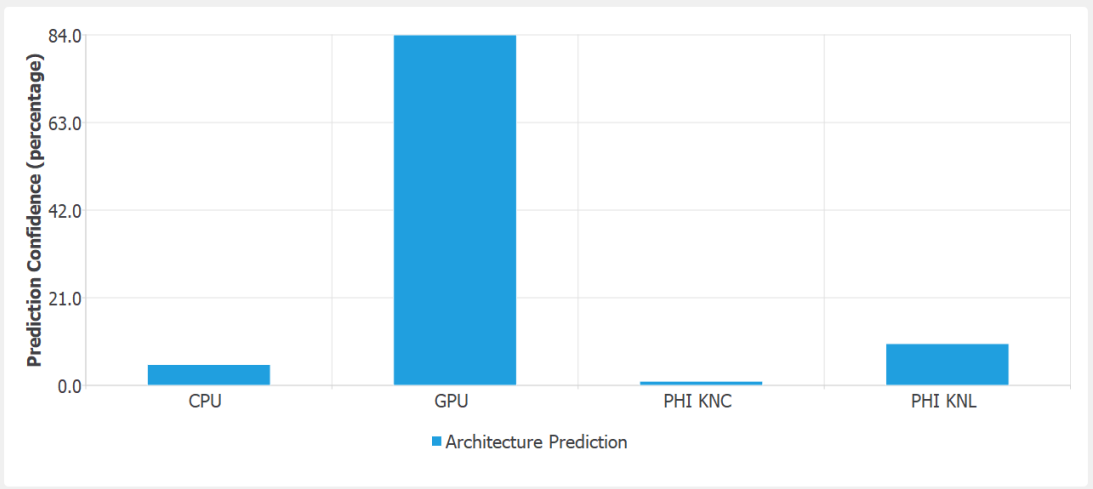

Figure 6.2: Percentage confidence of architecture prediction for Lulesh

Following this, we have the Programming Language Prediction graph shown in Figure 6.3 which predicts CUDA as the best programming language for obtaining maximum speedup. The next predicted language is OpenCL, which again can be used to program GPUs. While the difference in execution time for both programming languages running on the GPU is marginal, OpenCL provides a significant advantage when it comes to portability. Since OpenCL was designed keeping heterogeneous architectures in mind, it is straight forward to port a GPU-based OpenCL application to a CPU-based OpenCL application and vice versa. Thus, we again provide the developer with the option for maximum application speedup or maximum application portability.

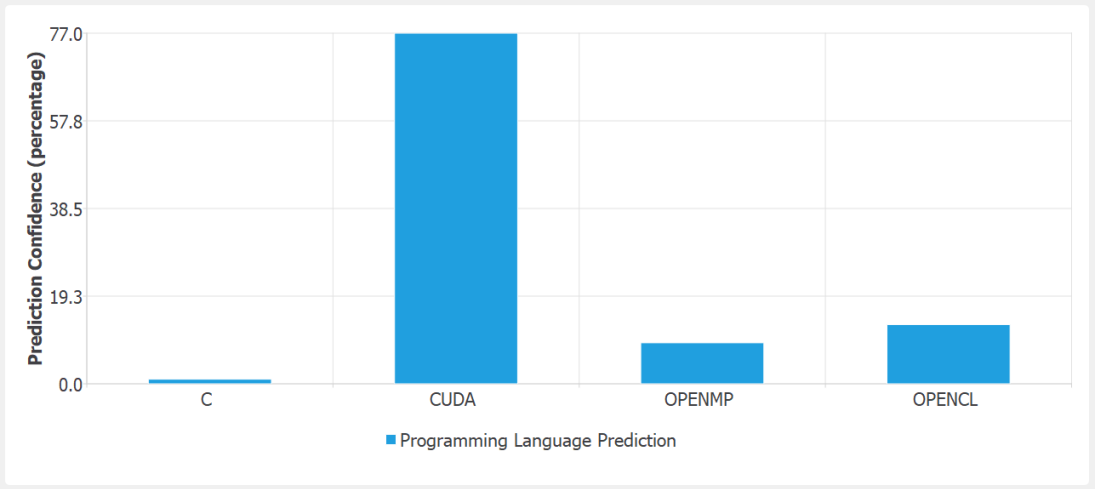

Figure 6.3: Percentage confidence of programming language prediction for Lulesh

Finally, we see that the Platform Prediction graph shown in Figure 6.4 is in sync with the previous two results and predicts GPU-CUDA to be the best platform followed by GPU-OpenCL for our test application. The advantage of having the Platform Prediction graph is that it helps the developer have an overview of the available architecture and programming language combinations and compare the performance provided by each combination. Moreover, it also highlights the drawbacks of the architectures and the programming languages themselves as some architectures and programming languages are not compatible. For example, Nvidia GPUs can only be programmed using Nvidia CUDA or OpenCL while AMD GPUs can only be programmed using OpenCL. Thus, by informing the developer of all the options, it allows the developer to select the best available platform.

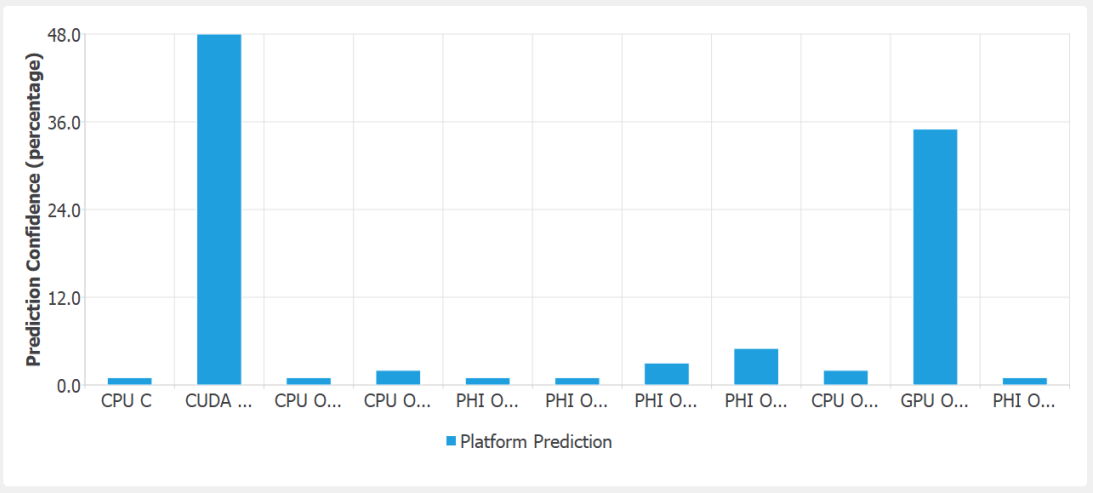

Figure 6.4: Percentage confidence of platform prediction for Lulesh

6.3 Quantitative Model Results

The Quantitative Model lets us perform the execution time prediction for each platform, which is represented in graphical form in Figure 6.5. According to the graph, the GPU implementations using CUDA and OpenCL will have the least execution time followed by the PHI-KNL implementation using OpenMP. This result proves that our Qualitative Model is just as accurate as the Quantitative Model without providing the in-depth execution time information.

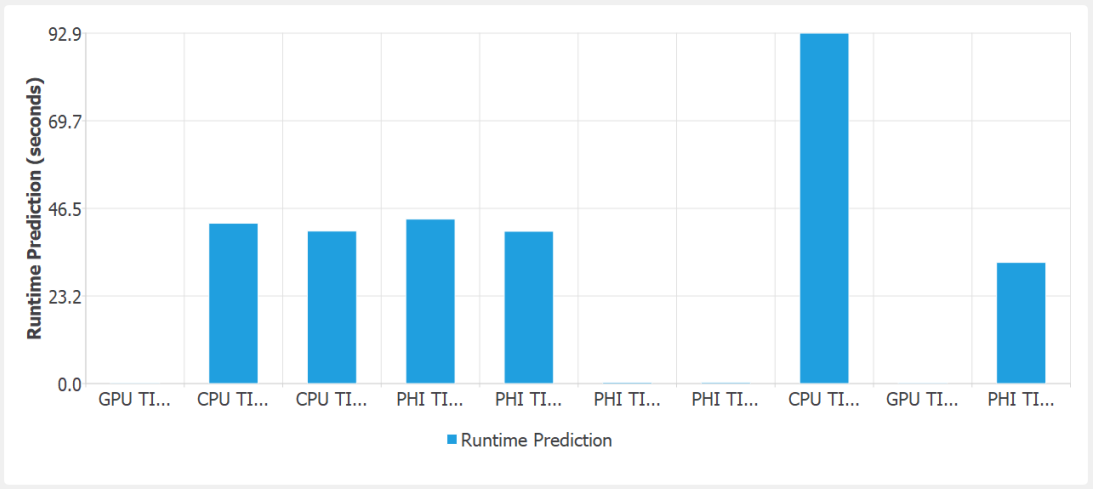

Figure 6.5: Predicted execution time on each platform for Lulesh

Platform	Actual	CPU	GPU-CU	CPU-OMP	PHI-OMP	Avg. MAE
CPU	119.11	NA	85.54	126.72	161.37	27.81
GPU-CUDA	0.0567	0.1796	NA	0.2164	1.2617	0.49
CPU-OMP08	38.29	42.51	33.14	41.73	65.69	10.05
CPU-OMP16	34.26	40.44	29.24	NA	59.13	12.02
PHI-OMP120	51.18	43.61	39.79	44.59	48.68	7.01
PHI-OMP240	47.59	40.38	33.93	39.44	NA	9.67
PHI-OMP136	0.2491	0.3391	0.1682	0.3076	2.3791	0.59
PHI-OMP272	0.1843	0.3024	0.1428	0.2573	2.1399	0.54

Table 6.1: Run time prediction results (in seconds) for Lulesh using Hardware Counters collected from four different platform.

Finally, to verify the accuracy of our prediction models, we run all the available implementation of Lulesh and record their execution times as presented in Table 6.1. The first column labels the implementation we are trying to predict, the second column is the actual run time collected by running the application on the specified hardware and the remaining four columns are run times predicted using hardware counters from four different platforms. We see that the average mean absolute error for our prediction is 8.52 seconds and if we look closely we observe that the model

over predicts in majority of the cases. Thus, we can be assured that the run time the developer will observe on implementing the application on a different platform will be bound by the predicted run time. Moreover, since Lulesh lacks an OpenCL implementation, we were unable to record their execution time and compare them with the predicted values.

In order to evaluate the prediction accuracy of the model for an application with varying input sizes, we provide the model with Hardware Counters for the Histogram application running on default input size. The model then predicts the execution time for the same application with varying input sizes. The prediction results are shown in Table 6.2 and we see that the model is able to rightly capture the input scaling behaviour of the application and predict the execution time with an average Mean Absolute Error of 2.4.

Input Size	**Actual**	**Predicted**	**MAE**
X-SMALL	3.63	5.72	2.09
SMALL	7.84	8.19	0.35
DEFAULT	16.11	NA	NA
LARGE	28.46	33.26	4.8
X-LARGE	42.57	44.93	2.36

Table 6.2: Run time prediction results (in seconds) for different input sizes of Histogram application using Hardware Counters collected from default input size.

6.4 Summary

In this chapter, we presented the workflow of our toolkit using two different applications. The first application is Lulesh, which simulates the motion of materials relative to each other when subject to forces. The toolkit predicts GPU to be the best architecture and CUDA to be the best programming language along with GPU-CUDA to be the best architecture and programming language combination. Moreover, of each of the platforms, the predicted mean absolute execution time error is 8.52. Following this, for predicting execution time for varying input sizes, we use an application which calculates the histogram of an image. The mean absolute prediction error in this case is 2.4, thus proving that the toolkit takes into consideration input scaling for execution time prediction.

Chapter 7

Conclusion and Future Scope

7.1 Conclusion

In this thesis, we have successfully designed and developed the front-end and made significant contribution towards the back-end to create a performance prediction tool that uses hardware counters from an already existing implementation to predict the performance on an accelerator-based system. We looked at some of the pre-existing prediction models and performance analysis tools and evaluated in detail their advantages and disadvantages. The fact that the prediction models lacked a graphical user interface and the performance analysis tools lacked a prediction model is the motivation for this thesis research.

We provided a discussion of the toolkit architecture and focus on the details concerning the back-end and the front-end of the toolkit. The back-end consists of a Random Forest Classifier based Qualitative Model and a Neural Network Regressor based Quantitative Model. On the other hand, the front-end has been designed and developed in C++ using QT application design framework.

We finally evaluated the prediction performance of our toolkit by testing it on a real world application called Lulesh, which simulates the motion of materials relative to each other when subjected to forces. We walk-through the different steps a user must follow to in order to predict the performance of his/her application on a wide variety of platforms. We observe that the average mean absolute error for our prediction is 8.52 seconds and observe that the model over predicts in a majority of the cases. Thus, we get an upper bound for the execution time and the developer will observe a time less than the predicted time for the final implementation. Moreover, we test the

accuracy of our prediction model for an application with varying input sizes. The results show that our prediction model is able to capture an application's input scaling behavior. We also demonstrate the additional features present in our toolkit that help a developer visualize some of the common performance analysis data such as functional breakdown and the different hardware counters. All of this information helps the developer to identify bottlenecks and make better optimization decisions in his/her application development.

In conclusion, we have designed and developed a performance analysis toolkit that overcomes the drawbacks of existing performance prediction models and performance analysis tools. By using our toolkit, a developer will be able to investigate the best architecture and the programming language combination for their application all without implementing it for the target platform. This capability will save significant amount of time and money as having prior knowledge of the expected execution time will let the developer decide if porting is a good solution or not.

7.2 Future Scope

While our prediction model performs on par with other prediction models, in the future other researchers might come up with better prediction models and thus having the ability to replace the prediction model in the back-end will be a benefit. Hence we would like to update our toolkit with the ability for developers to load his/her custom prediction model. Moreover, we plan to improve our back-end model through better training methods and introducing training inputs that have a higher correlation to the execution time, thus keeping our tool up-to-date with other prediction models. Additionally, we plan to implement methods for users to provide us with application data that can be used to train our model to further improve its accuracy.

Other improvements to the back-end include classification of applications into different categories so that we can use different prediction models for each category. Code characteristics such as input data dimension and algorithm complexity can be used to classify applications into different categories. This capability will ensure better predictions since a model specific to a given class of applications is bound to capture fine grain application characteristics.

Improvements on the user interface and visualization side include adding visualizations for the input data and how the program accesses it over the course of its execution. This addition will aid the programmer in understanding data dependencies and data access patterns, which are very

important in accelerator architectures to obtain maximum speedup.

We believe all of these improvements in addition to regular model updates will make our tool even more useful as a performance prediction and analysis tool.

Appendices

Appendix A Hardware counters

Platform	Hardware Counters
CUDA-K40	PAPI-SR-INS, PAPI-LD-INS, PAPI-L1-DCM, PAPI-FP-OPS, PAPI-TOT-CYC
OpenMP-16	PAPI-LD-INS, PAPI-SR-INS, PAPI-L1-DCM, PAPI-FP-OPS, PAPI-TOT-CYC
OpenMP-08	PAPI-LD-INS, PAPI-SR-INS, PAPI-L1-DCM, PAPI-FP-OPS, PAPI-TOT-CYC
OpenMP-240	PAPI-LD-INS, PAPI-TOT-CYC, PAPI-BR-INS, PAPI-FP-OPS, PAPI-SR-INS, PAPI-LD-INS
OpenMP-120	PAPI-LD-INS, PAPI-TOT-CYC, PAPI-BR-INS, PAPI-FP-OPS, PAPI-SR-INS, PAPI-LD-INS
OpenMP-272	PAPI-LD-INS, PAPI-TOT-CYC, PAPI-BR-INS, PAPI-FP-OPS, PAPI-SR-INS, PAPI-LD-INS
OpenMP-136	PAPI-LD-INS, PAPI-TOT-CYC, PAPI-BR-INS, PAPI-FP-OPS, PAPI-SR-INS, PAPI-LD-INS
CPU OpenCL	PAPI-LD-INS, PAPI-SR-INS, PAPI-L1-DCM, PAPI-FP-OPS, PAPI-TOT-CYC
GPU OpenCL	PAPI-LD-INS, PAPI-SR-INS, PAPI-L1-DCM, PAPI-FP-OPS, PAPI-TOT-CYC
Xeon Phi OpenCL	PAPI-LD-INS, PAPI-TOT-CYC, PAPI-BR-INS, PAPI-FP-OPS, PAPI-SR-INS, PAPI-L1-DCM

Table 1: Hardware counters used to train the CPU-C model

Platform	Hardware Counters
CPU-C	inst-executed, inst-issued2, gst-inst-32bit, gst-request, l2-subp2-write-sector-misses, l2-subp0-write-sector-misses, l2-subp3-write-sector-misses, l2-subp1-write-sector-misses, not-predicated-off-thread-inst-executed, thread-inst-executed, gld-inst-32bit, gld-request, fb-subp1-write-sectors, fb-subp0-write-sectors
OpenMP-16	inst-executed, inst-issued2, gst-request, gst-inst-32bit, l2-subp2-write-sector-misses, gld-request, gld-inst-32bit, inst-issued1
OpenMP-08	inst-executed, inst-issued2, gst-request, gst-inst-32bit, l2-subp2-write-sector-misses, gld-request, gld-inst-32bit, inst-issued1
OpenMP-240	inst-executed, inst-issued2, gst-request, gst-inst-32bit, l2-subp2-write-sector-misses, gld-request, gld-inst-32bit, inst-issued1
OpenMP-120	inst-executed, inst-issued2, gst-request, gst-inst-32bit, l2-subp2-write-sector-misses, gld-request, gld-inst-32bit, inst-issued1
OpenMP-272	inst-executed, inst-issued2, gst-request, gst-inst-32bit, l2-subp2-write-sector-misses, gld-request, gld-inst-32bit, inst-issued1
OpenMP-136	inst-executed, inst-issued2, gst-request, gst-inst-32bit, l2-subp2-write-sector-misses, gld-request, gld-inst-32bit, inst-issued1
CPU OpenCL	global-st-mem-divergence-replays, global-store-transaction, –l1-global-store-transaction, inst-issued1, gld-request
GPU OpenCL	inst-issued1, gld-request, gld-inst-32bit, fb-subp0-write-sectors, l2-subp1-total-write-sector-queries, l2-subp0-total-write-sector-queries
Xeon Phi OpenCL	global-st-mem-divergence-replays, global-store-transaction, –l1-global-store-transactions, l2-subp0-total-write-sector-queries, l2-subp0-write-l1-sector-queries

Table 2: Hardware counters used to train the CUDA-K40 model

Platform	Hardware Counters
CPU-C	PAPI-FP-INS-mean, PAPI-TOT-INS-thread0, PAPI-LD-INS-thread0, PAPI-DP-OPS-mean, PAPI-SR-INS-thread0
CUDA-K40	PAPI-FP-OPS-total, PAPI-DP-OPS-total, PAPI-LD-INS-total, PAPI-TOT-INS-total, PAPI-REF-CYC-thread0
OpenMP-08	PAPI-TOT-INS-thread0, PAPI-LD-INS-thread0, PAPI-REF-CYC-thread0, PAPI-SR-INS-thread0, PAPI-FP-INS-thread0, PAPI-L2-DCH-thread0, PAPI-LD-INS-total
OpenMP-240	PAPI-TOT-CYC-thread0, PAPI-REF-CYC-thread0, PAPI-SR-INS-thread0, PAPI-TOT-INS-thread0, PAPI-LD-INS-thread0
OpenMP-120	PAPI-TOT-CYC-thread0, PAPI-REF-CYC-thread0, PAPI-SR-INS-thread0, PAPI-TOT-INS-thread0, PAPI-LD-INS-thread0
OpenMP-272	PAPI-TOT-CYC-thread0, PAPI-REF-CYC-thread0, PAPI-SR-INS-thread0, PAPI-TOT-INS-thread0, PAPI-LD-INS-thread0
OpenMP-136	PAPI-TOT-CYC-thread0, PAPI-REF-CYC-thread0, PAPI-SR-INS-thread0, PAPI-TOT-INS-thread0, PAPI-LD-INS-thread0
CPU OpenCL	PAPI-TLB-DM-mean, PAPI-L3-TCW-thread0, PAPI-L2-STM-thread0, PAPI-L3-DCA-mean, PAPI-L2-DCM-total
GPU OpenCL	PAPI-FP-OPS-total, PAPI-DP-OPS-total, PAPI-LD-INS-total, PAPI-TOT-INS-mean, PAPI-REF-CYC-thread0
Xeon Phi OpenCL	PAPI-TLB-DM-thread0, PAPI-L3-TCA-thread0, PAPI-L3-DCA-thread0, PAPI-L2-DCM-thread0, PAPI-L2-TCM-thread0

Table 3: Hardware counters used to train the CPU-OpenMP model

Platform	Hardware Counters
CPU-C	PAPI-VEC-INS-total, PAPI-TOT-INS-mean, PAPI-LD-INS-mean, PAPI-L1-DCA-thread0, PAPI-SR-INS-thread0
CUDA-K40	PAPI-VEC-INS-total, PAPI-TOT-INS-total, PAPI-LD-INS-total, PAPI-L1-DCA-mean, PAPI-L1-ICA-mean, PAPI-SR-INS-thread0
OpenMP-16	PAPI-TOT-CYC-thread0, PAPI-LD-INS-mean, PAPI-L1-ICA-thread0, PAPI-L1-ICA-total, PAPI-SR-INS-thread0, PAPI-VEC-INS-mean
OpenMP-08	PAPI-TOT-CYC-thread0, PAPI-LD-INS-mean, PAPI-L1-ICA-thread0, PAPI-L1-ICA-total, PAPI-SR-INS-thread0, PAPI-VEC-INS-mean
OpenMP-120	PAPI-TOT-CYC-thread0, PAPI-L1-DCM-thread0, PAPI-L1-ICA-thread0, PAPI-L2-LDM-mean, PAPI-LD-INS-thread0
OpenMP-272	PAPI-TOT-CYC-thread0, PAPI-LD-INS-mean, PAPI-L1-ICA-thread0, PAPI-L1-ICA-total, PAPI-SR-INS-thread0, PAPI-VEC-INS-mean
OpenMP-136	PAPI-TOT-CYC-thread0, PAPI-LD-INS-mean, PAPI-L1-ICA-thread0, PAPI-L1-ICA-total, PAPI-SR-INS-thread0, PAPI-VEC-INS-mean
CPU OpenCL	PAPI-TLB-DM-thread0, PAPI-L2-LDM-thread0, PAPI-L1-DCM-thread0, PAPI-TOT-CYC-thread0, PAPI-L1-ICA-thread0
GPU OpenCL	PAPI-VEC-INS-total, PAPI-TOT-CYC-thread0, PAPI-LD-INS-total, PAPI-L1-DCA-mean, PAPI-SR-INS-thread0
Xeon Phi OpenCL	PAPI-TLB-DM-thread0, PAPI-L2-LDM-thread0, PAPI-L1-DCM-thread0, PAPI-TOT-CYC-thread0, PAPI-L1-ICA-thread0

Table 4: Hardware counters used to train the PHI-OpenMP model

Bibliography

[1] BlueGene/L supercomputer. *https://asc.llnl.gov/computing_resources/bluegenel/*.

[2] NVIDIA CUDA Profiling Tools Interface (CUPTI). *https://developer.nvidia.com/cuda-profiling-tools-interface/*.

[3] NVIDIA Visual Profiler. *https://developer.nvidia.com/nvidia-visual-profiler/*.

[4] PACUBE : Performance Prediction for Porting Applications and Algorithms to Architectures. *https://www.github.com/PACUBE/*.

[5] Thunder supercomputer. *https://www.llnl.gov/linux/thunder/*.

[6] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[7] Laksono Adhianto, Sinchan Banerjee, Mike Fagan, Mark Krentel, Gabriel Marin, John Mellor-Crummey, and Nathan R Tallent. Hpctoolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 22(6):685–701, 2010.

[8] Newsha Ardalani, Clint Lestourgeon, Karthikeyan Sankaralingam, and Xiaojin Zhu. Cross-architecture performance prediction (xapp) using cpu code to predict gpu performance. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 725–737. ACM, 2015.

[9] Sara S Baghsorkhi, Matthieu Delahaye, Sanjay J Patel, William D Gropp, and Wen-mei W Hwu. An adaptive performance modeling tool for gpu architectures. In *ACM Sigplan Notices*, volume 45, pages 105–114. ACM, 2010.

[10] Vasanth Balasundaram and Ken Kennedy. *A technique for summarizing data access and its use in parallelism enhancing transformations*, volume 24. ACM, 1989.

[11] Ioana Baldini, Stephen J Fink, and Erik Altman. Predicting gpu performance from cpu runs using machine learning. In *2014 26th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 254–261. IEEE, 2014.

[12] Robert Bell, Allen D Malony, and Sameer Shende. Paraprof: A portable, extensible, and scalable tool for parallel performance profile analysis. In *European Conference on Parallel Processing*, pages 17–26. Springer, 2003.

[13] Laura Carrington, Mustafa M Tikir, Catherine Olschanowsky, Michael Laurenzano, Joshua Peraza, Allan Snavely, and Stephen Poole. An idiom-finding tool for increasing productivity of accelerators. In *Proceedings of the international conference on Supercomputing*, pages 202–212. ACM, 2011.

[14] Paul J Drongowski. Instruction-based sampling: A new performance analysis technique for amd family 10h processors. *Advanced Micro Devices*, 2007.

[15] Sunpyo Hong and Hyesoon Kim. An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 152–163. ACM, 2009.

[16] Engin Ipek, Bronis R De Supinski, Martin Schulz, and Sally A McKee. An approach to performance prediction for parallel applications. In *Euro-Par 2005 Parallel Processing*, pages 196–205. Springer, 2005.

[17] Ian Karlin, Jeff Keasler, and Rob Neely. Lulesh 2.0 updates and changes. Technical Report LLNL-TR-641973, August 2013.

[18] Benjamin C Lee, David M Brooks, Bronis R de Supinski, Martin Schulz, Karan Singh, and Sally A McKee. Methods of inference and learning for performance modeling of parallel applications. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 249–258. ACM, 2007.

[19] Rama Kishan Malladi. Using intel® vtune performance analyzer events/ratios & optimizing applications. *http:/software. intel. com*, 2009.

[20] Ryszard S Michalski, Jaime G Carbonell, and Tom M Mitchell. *Machine learning: An artificial intelligence approach*. Springer Science & Business Media, 2013.

[21] Philip J Mucci, Shirley Browne, Christine Deane, and George Ho. Papi: A portable interface to hardware performance counters. In *Proceedings of the department of defense HPCMP users group conference*, volume 710, 1999.

[22] Wolfgang E Nagel, Alfred Arnold, Michael Weber, Hans-Christian Hoppe, and Karl Solchenbach. Vampir: Visualization and analysis of mpi resources. 1996.

[23] Cedric Nugteren and Henk Corporaal. The boat hull model: adapting the roofline model to enable performance prediction for parallel computing. In *ACM Sigplan Notices*, volume 47, pages 291–292. ACM, 2012.

[24] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[25] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[26] Leo T Yang, Xiaosong Ma, and Frank Mueller. Cross-platform performance prediction of parallel applications using partial execution. In *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*, pages 40–40. IEEE, 2005.

[27] Yao Zhang and John D Owens. A quantitative performance analysis model for gpu architectures. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pages 382–393. IEEE, 2011.