Cross-Architecture Performance Prediction (XAPP) Using CPU Code to Predict GPU Performance

Newsha Ardalani Clint Lestourgeon Karthikeyan Sankaralingam Xiaojin Zhu
University of Wisconsin-Madison
{newsha, clint, karu, jerryzhu}@cs.wisc.edu

ABSTRACT

GPUs have become prevalent and more general purpose, but GPU programming remains challenging and time consuming for the majority of programmers. In addition, it is not always clear which codes will benefit from getting ported to GPU. Therefore, having a tool to estimate GPU performance for a piece of code before writing a GPU implementation is highly desirable. To this end, we propose Cross-Architecture Performance Prediction (XAPP), a machine-learning based technique that uses only single-threaded CPU implementation to predict GPU performance.

Our paper is built on the two following insights: i) Execution time on GPU is a function of program properties and hardware characteristics. ii) By examining a vast array of previously implemented GPU codes along with their CPU counterparts, we can use established machine learning techniques to learn this correlation between program properties, hardware characteristics and GPU execution time. We use an adaptive two-level machine learning solution. Our results show that our tool is robust and accurate: we achieve 26.9% average error on a set of 24 real-world kernels. We also discuss practical usage scenarios for XAPP.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Modeling techniques; I.3.1 [**Hardware Architecture**]: Graphics processors

Keywords

GPU, Cross-platform Prediction, Performance Modeling, Machine Learning

1. INTRODUCTION

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 GPU *directly* without algorithm redesign/change. In many environments the above four scenarios get intermingled. This paper 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 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, 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 (we demonstrate a concrete end-to-end usage scenario in Section 6). (iv) *Worthwhile to port a region of CPU code*: In some cases, algorithm change (sometime radical) is required to get high performance and some GPU gurus assert that the CPU code is useless. However, accurate CPU-based GPU prediction can inform the programmer whether algorithmic change is indeed required when tasked with porting a region of CPU code.

In summary, CPU-based GPU performance predic-

MICRO-48, December 05-09, 2015 Waikiki, HI, USA

ACM 978-1-4503-4034-2/15/12..$15.00.
DOI: http://dx.doi.org/10.1145/2830772.2830780.

	CPU→GPU Prediction [1, 2, 3]	GPU→GPU Prediction [8]	Auto-Compile [4, 5, 6] [7, 9]	**XAPP**
Accuracy	Low	Medium	High	High
Usability	Medium	Low	Medium	High
App Generality	High	Low	Low	High
HW Generality	High	Low	High	High
Speed	High	High	High	High

Table 1: A comparison among the state-of-the-art.

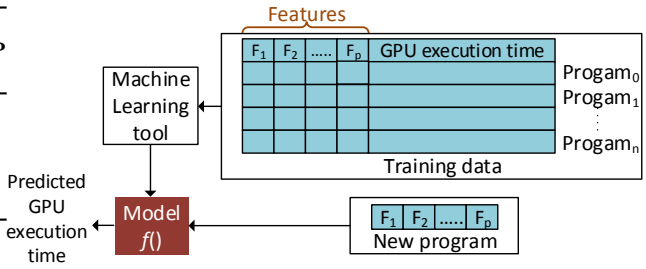

Figure 1: XAPP overall flow.

tion 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.

An ideal GPU performance prediction framework should have several key properties: *accuracy* – the degree to which the actual and predicted performance matches; *application-generality* – being able to model a wide variety of applications; *hardware generality* – being easily extendable for various GPU hardware platforms; *speed* – being able to predict performance quickly; and *programmer usability* – having low programmer involvement in the estimation process.

The literature on GPU performance prediction from GPU code, sketches, and other algorithmically specialized models can be repurposed for our problem statement and evaluated using our five metrics [1, 2, 3, 4, 5, 6, 7]. Table 1 categorizes them according to these five metrics. As shown in the Table, no existing work can achieve all five properties. We further elaborate on these works in Section 7.

To satisfy all five properties, we introduce and evaluate XAPP (GPU Performance eStimator), an automated performance prediction tool that can provide highly accurate estimates of GPU performance, when provided a piece of CPU code *prior to developing the GPU code*. We anticipate programmers will use this tool early in the software development process. Note that our tool does not predict how to port a code to GPU, but how much speedup is achievable if ported into an optimized version.

Our paper 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[1] collected from the CPU implementation and GPU execution time measured from the GPU implementation.

[1]Program property, program feature or program characteristic are terms used interchangeably in this paper.

Based on the above observations, we build XAPP, which is a machine learning tool that predicts GPU execution time based on quantitative program properties derivable from the CPU code. Figure 1 shows the overall flow of XAPP. During a one-time training phase (per GPU platform), it uses a corpus of training data, comprising program features and measured GPU execution time, to learn a function that maps program properties to GPU execution time. To predict GPU execution time for a new program, one measures its features and applies the function to obtain GPU execution time. XAPP repurposes commonly available binary instrumentation tools to quantitatively measure program features. We evaluated XAPP using a set of 24 real-world kernels and compared our speedup prediction[2] with the actual speedup measured on two different GPU platforms. These kernels represent a wide range of application behaviors (speedup range of 0.8× to 109×) and these platforms represent very different GPU cards with different micro-architectures. Our results show that we can achieve an average error of 26.9%, on a Maxwell GPU and 36.0% on a Kepler GPU.

Contributions This paper has two contributions. The primary contribution of this paper is the observation that for any GPU platform, GPU execution time can be formulated in terms of program properties as *variables* and GPU hardware characteristics as *coefficients*. A *variable* changes from one application to another, while a *coefficient* is fixed for all applications and needs to be captured once for each platform. Here we define *program property* as a *feature* that is inherent to the program or algorithm and is independent of what hardware the program runs on. For example, the mix of arithmetic operations, the working set size of the data and the number of concurrent operations are all examples of program properties which can be quantified and measured in various ways regardless of what type of machine the program is running on. Hoste and Eeckhout provide an elaborate treatment and measurement of such program properties, calling them microarchitecture-independent characteristics [10]. The second contribution is a set of engineering techniques to demonstrate that our tool is effective.

The rest of this paper is organized as follows. Section 2 outlines the foundations of our work. Section 3 explains the program properties that correlate with

[2]We convert our execution time prediction into speedup prediction using measured CPU time.

GPU execution time. Section 4 describes our machine learning technique. Section 5 and Section 6 present quantitative evaluation. Section 7 discusses related work and Section 8 concludes.

2. GPU EXECUTION TIME IS CORRELATED WITH PROGRAM BEHAVIOR

Main Observation: Considering some GPU platform x, our observation is that some mathematical function maps program properties (features) to GPU execution time. Considering features $f_0, f_1, f_2, ...$, mathematically our observation is 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 observation is the key novel contribution of our work.

In mathematical terms, our observation is indeed simple. However, it enables us to collect program properties from any implementation, including the CPU implementation, GPU implementation or algorithm. Given this observation, we take the following steps:

1. **Feature definition (Section 3)** 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 4)** 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 5,6)** 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 paper.

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. DEFINING PLAUSIBLE PROGRAM 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 to define a list of features. We started with a list of features that have been used in previous workload characterizations, and defined a few additional 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, branching behavior and how 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 2 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. Section 4.4 describes the tools we use to measures these properties on applications. Below we describe a few example features to outline how we arrived at some of the non straight-forward features.

shMemBW and noConflict Bank conflicts in shared memory are 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. $noConflict$, specifically captures the case where the $stride$ is an odd number.

gMemBW and coalesced Non-coalesced memory accesses are also 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

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	Captures bank conflicts in shared memory.
coalesced	0 - 1	See Section 3	Captures global memory coalescing.
shMemBW	0 - 1	See Section 3	Captures shared memory bank-effectiveness.
gMemBW	0 - 1	See Section 3	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	Captures the branching pattern.

Table 2: List of program properties used as input features.

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. *coalesced* specifically captures the case where i is one.

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 branches within them are not going in the same direction.

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.

4. MACHINE LEARNING MODEL

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. In our work we use machine-learning (specifically an ensemble of regression learners for which we include a short primer in the Appendix). 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. 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.

4.1 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

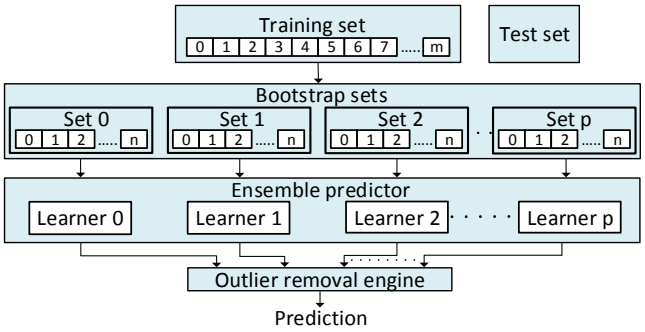

Figure 2: Model construction overview.

appear in one 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 [11, 12, 13]. In Subsection 5.3, we discuss the analyses we made that ultimately lead us to use an ensemble solution.

4.2 Implementation Details

We employ an adaptive two-level machine learning technique. We explain our model construction procedure, which entails the details of the regression model at the first level and the ensemble algorithm at the secod level.

Level 1: Regression We use forward feature selection stepwise regression [14] 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 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 [15].

Level 2: Ensemble Prediction Figure 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 evaluation in the end. 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 $n \approx 0.63m$ [13]. We then construct p individual models[3] 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 using measured CPU time.

4.3 Training Data and Test Data

We examined many prevalent benchmarks suites, namely Lonestar [16], Parsec subset [17], Parboil [18], Rodinia [19], NAS subset [20, 21] and some in-house benchmarks based on the throughput kernels [22]. 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. We consider two algorithms to be similar as long as the computational complexity matches. That is, common optimizations (such as loop reordering, loop blocking and overlapped tiling) that change the order of memory accesses, but do not change the number of memory operations, are not algorithmic mismatch and can be used in our dataset. To give an example, matrix multiplication 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

[3]In this paper, we use base learner and individual model interchangably, the latter being an informal but more intuitive term.

Model	ninst*(	Lbdiv32	+mem	+gMemBW	+gMemBW:Lbdiv32	+pages	+pages:Lbdiv32	+stride0	+stride0:pages	+dp	+blocks
Coefficient		0.0290038	0.0126532	-0.0228180	-0.0070995	-0.0380114	0.1076867	-0.0027127	-0.0196313	-0.0045341	0.0968233
Adjusted R^2			0.1781	0.2704	0.3173	0.3685	0.4697	0.5061	0.5412	0.5649	0.5776

Model	+blocks:pages	+shMemBW	+shMemBW:mem	+ilp256	+ilp256:stride0	+arith	+div	+div:mem	+coalesced	+coldRef)
Coefficient	0.0010343	-0.0036923	-0.0072603	-0.0014900	-0.0036693	-0.0016565	-0.0057103	-0.0064187	0.0130530	0.0017546
Adjusted R^2	0.596	0.6024	0.6634	0.6678	0.6915	0.6969	0.699	0.7152	0.7162	0.717

Table 3: An instance of a regression model generated at first level.

	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 4: Hardware platforms pecifications.

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.

4.4 Hardware Platforms and Software Infrastructure

To demonstrate robustness, we considered two different hardware platforms (summarized in Table 4) for which we automatically predict speedups. All the explanations in this paper are for Platform-1, and we use Platform-2 to test the *HW generality* property. We used MICA [10] and Pin [23] to obtain program properties. The tools that we wrote for `Lbdiv` and others (highlighted in grey in Table 2) 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 [24]. 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.

Kernel	Suite	Actual Speedup	Predicted Speedup	Relative Error%
μ3	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: Accuracy of a representative ensemble model.

5. 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.

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. In the next Subsection, we discuss why we need to use ensemble solution. We then try to provide insight into our ensemble result. We then compare our solution against existing solutions, in terms of the metrics introduced in Section 1. Finally, we list the limitations and extensions of our tool.

5.1 Accuracy

Summary: Table 5 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, XAPP is accurate.

To study the accuracy of a model, the common practice is to use the model to predict the output for a

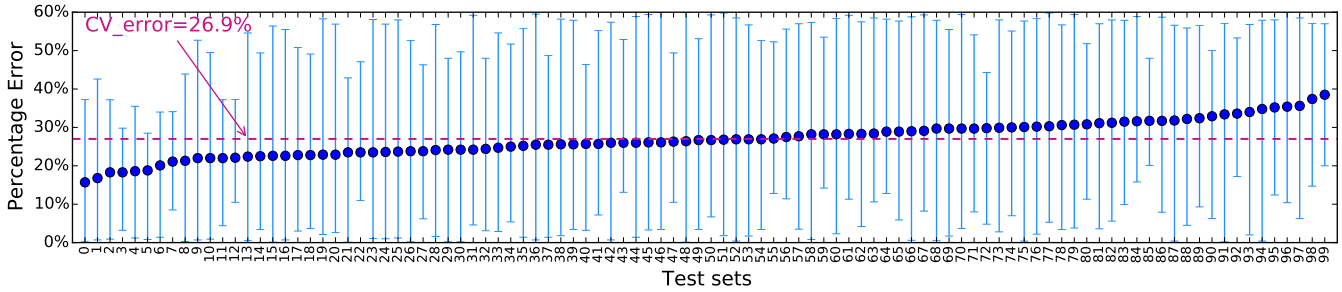

Figure 3: Ensemble technique accuracy across 100 different sets of test and train.

set of diverse datapoints that never appeared in the training process, measure the prediction 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 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 Robustness

Summary: Figure 3 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 XAPP is very robust.

A machine learning technique is robust, if any given set of test or train generates similar models with similar accuracy [25]. To evaluate the robustness of our technique, we draw 100 different pairs of test and train, and construct 100 different ensemble models[4] , as outlined in Subsection 4.2. Figure 3 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, 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.

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

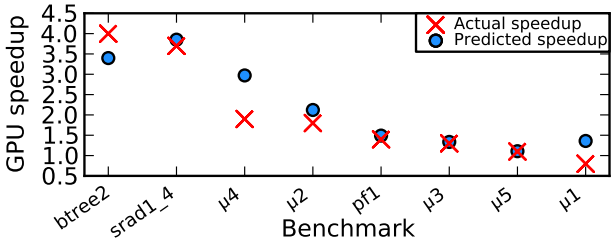

Figure 4: Low-speedup prediction accuracy.

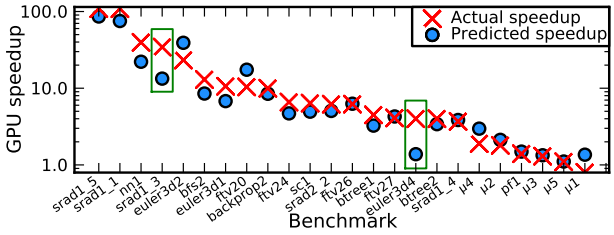

Figure 5: Accuracy results across all QFT kernels.

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 4 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 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 correctly, and are often close to the actual value.

5.3 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

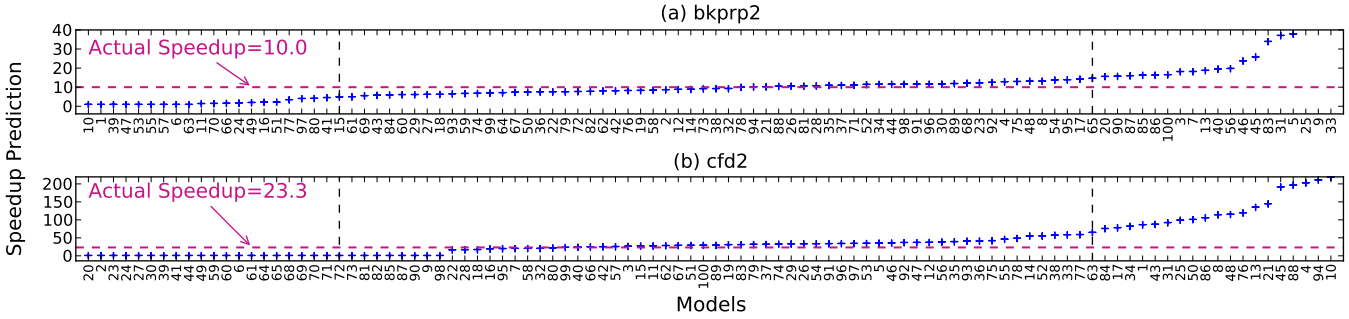

Figure 6: High sensitivity of single learners to the choice of train and how ensemble and outlier removal can fix it.

	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 6: Model disagreement Matrix.

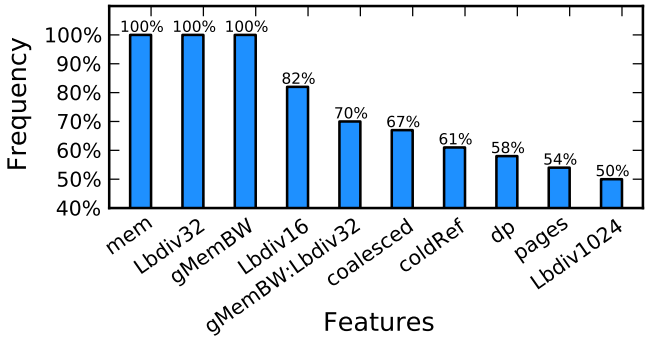

Figure 7: Highly correlated features with GPU execution time.

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 6(a) shows speedup predictions for *bkprop*2, 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 6(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 *bkprop*2 are actually good for $cfd2$. We call this phenomenon model disagreement, and we show that this is prevalent between any two programs; Table 6 shows model disagreement across the 10 different programs that appeared in Table 5, 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 6 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 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 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%.

5.4 Model Interpretation

Summary: Figure 7 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 [26, 27, 28, 29]. 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 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). [27, 28, 29]. 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 7 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-point arithmetic vs. integer or float arithmetic (*dp*) is intuitive. Therefore, we only focus on the non-intuitive pairwise feature.

gMemBW × **Lbdiv32**, which appeared across 70 models (with negative sign) captures how perfect memory coalescing (high $gMemBW$) can cancel out the increasing impact of high branch divergence (high $Lbdiv32$) on execution time.

5.5 Other Metrics

We now evaluate our tool in terms of the other four properties introduced in Section 1.

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 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 XAPP has high application generality.

HW 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 4. 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 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 3 hours. (2) Recurring Ovehread: The user needs to gather features for the candidate program. This takes seconds to minutes — the instrumentation run introduces a 10× to 20× slowdown to native execution. Speedup projection completes in milliseconds — it is a matter of computing the function obtained in the previous phase.

Good for GPU	Easy for Human	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 7: XAPP prediction space. The last column shows example code in case-studies in Figure 8.

5.6 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.

6. END TO END CASE STUDIES

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

6.1 Is 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 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 7 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 8 shows the interesting regions of these CPU codes. The boxes indicate percentage execution based on profiling information for that region of the CPU code. These applications were picked intentionally to specifically highlight that our tool is *not* perfect. 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

```
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 8: Case study kernel regions

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 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 XAPP as an adviser to get an estimate, but ultimately users should pay attention. As we get more training data, 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 -ve.

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 XAPP corroborates this.

CS2,K1: Good for GPU, Hard for Human, False -ve.

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. XAPP predicts a speedup of 15.6 while the measured speedup is 34.4. Even though 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 -ve.

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 XAPP predicts correctly. This case study shows that XAPP can predict the speedup of programs that might appear ambiguous to humans.

CS2,K3: Good for GPU, Easy for Human, True +ve.

This kernel is extremely simple and uses the `logf` function and hence should have a good speedup. XAPP predicts correctly and this is easy for humans as well.

CS3: Good for GPU, Hard for Human, True +ve.

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. XAPP predicts a relatively high value of speedup (21.29) that almost seems counterintuitive. 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.

6.2 Using XAPP

Programmers are often times tasked with porting a CPU code to a GPU platform. In this scenario, 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 XAPP's predictions for those kernels. The results are shown in Figure 8, case study 2. 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 [30].

7. RELATED WORK

CPU → GPU Performance Projection Existing techniques are not intended for accurate speedup prediction. The Roofline model is a simple analytical model that can provide upper-bound projections given platform-specific source code [1]. Roofline is not intended for accurate speedup prediction, but to provide high-level insight on potential performance limiting factors. The boat-hull model [3, 2] has similar goals as our paper, but approaches the problem from an algorithm-template standpoint. They build a model that can be viewed as a hybrid mechanistic model that uses information about an algorithm to make performance projections. While accurate, their approach is limited to "structured" algorithms like convolution and FFT, and cannot handle arbitrary code. Meswani et. al. [31] 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. [32] have proposed a binary predictor which can predict slowdown/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.

GPU → GPU Performance Projection Related work in this category are intended for GPU design space exploration and performance tuning [8, 15, 33, 34]. Further investigation is required to re-purpose their models to use CPU code as input.

Automatic GPU Code Generation GROPHECY is a novel approach that uses analytical models and code-skeletons to make performance prediction [4]. Like our work, the goal is to make the predictions prior to writing GPU code. However, GROPHECY requires the programmer to write code skeletons, which could be quite time-consuming, and also relies on the underlying GPU analytical models being accurate - this can be problematic as GPUs evolve, and first-order models can't capture their performance accurately. Auto-compilation from C/C++ to CUDA or GPU binaries is orthogonal to our work [5, 6, 7, 9]. Thus far, these efforts have not produced high quality code. 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. For example: nn1 and srad1_3. We conclude OpenACC is not yet effective and lacks application generality. We acknowledge that, if such compilers do succeed, tools like XAPP become irrelevant. Tools like XAPP can help guide the development of such compilers.

Use of ML models Machine Learning have been used for program classification [35, 36], design space exploration [37, 38, 39, 15, 40, 41], performance modeling [42, 43, 44] and hardware/software co-design [45].

8. CONCLUSION

In this paper, 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*. Our 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. We built an ensemble model using forward selection as the base learner. When applied to our test set, which was selected randomly from real-world kernels, our tool showed 26.9% avearge error.

The key contribution of our 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 first to observe this platform independent correlation.

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 provided 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 comple-

ment GPU analytical models. While our specific implementation (XAPP) is accurate, improving its accuracy is a promising direction for future work.

More broadly, our observation opens up many directions of future work. First, this observation 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 availablity 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 that we have already found to be useful. One direction of future work is to examine the extent to which performance counters are sufficient. Finally, beyond predicting just single output metrics, we could learn the 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. In all of these cases, the availability of good training data (which is very human intensive to generate) limits the effectiveness of the approach. A final open question is whether microbenchmarks, ideas from auto-tuning, or random-program generation from hardware verification, can be repurposed to generate high-quality training data automatically.

Acknowledgments

We would like to thank the anonymous reviewers, Tony Nowatzki and the members of Vertical Research Group for their useful feedback on this work. Support for this research was provided by NSF under the following grants CCF-1162215, CNS-1228782, CNS-1218432.

APPENDIX

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. 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 coeffcients (β) as follows:

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

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 Gausian 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 feaure. Similarly, we can define higher order power terms.

Model comparison: How well a model explaines 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: 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 [12].

A. REFERENCES

[1] 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.

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

[3] 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.

[4] 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.

[5] 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.

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

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

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

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

[10] K. Hoste and L. Eeckhout, "Comparing benchmarks using

key microarchitecture-independent characteristics," in *Workload Characterization, 2006 IEEE International Symposium on*, pp. 83–92, 2006.

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

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

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

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

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

[16] 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.

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

[18] 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.

[19] 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*.

[20] 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.

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

[22] 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*.

[23] 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*.

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

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

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

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

[28] 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.

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

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] G. Wu, J. L. Greathouse, A. Lyashevsky, N. Jayasena, and D. Chiou, "Gpgpu performance and power estimation using machine learning," in *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, pp. 564–576, IEEE, 2015.

[35] B. Piccart, A. Georges, H. Blockeel, and L. Eeckhout, "Ranking commercial machines through data transposition," in *Proceedings of the 2011 IEEE International Symposium on Workload Characterization*, IISWC '11, (Washington, DC, USA), pp. 3–14, IEEE Computer Society, 2011.

[36] J. Chen, L. K. John, and D. Kaseridis, "Modeling program resource demand using inherent program characteristics," *SIGMETRICS Perform. Eval. Rev.*, vol. 39, pp. 1–12, 2011.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] S. Sharkawi, D. DeSota, R. Panda, R. Indukuru, S. Stevens, V. Taylor, and X. Wu, "Performance projection of hpc applications using spec cfp2006 benchmarks," in *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pp. 1–12, 2009.

[45] 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.