A Static Analysis-based Cross-Architecture Performance Prediction Using Machine Learning

Newsha Ardalani*
Baidu Research
newsha@baidu.com

Urmish Thakker*
Arm ML Research
urmish.thakker@arm.com

Aws Albarghouthi
UW Madison
aws@cs.wisc.edu

Karu Sankaralingam
UW Madison
karu@cs.wisc.edu

ABSTRACT

Porting code from CPU to GPU is costly and time-consuming; Unless much time is invested in development and optimization, it is not obvious, a priori, how much speed-up is achievable or how much room is left for improvement. Knowing the potential speed-up a priori can be very useful: It can save hundreds of engineering hours, help programmers with prioritization and algorithm selection.

We aim to address this problem using machine learning in a supervised setting, using solely the single-threaded source code of the program, without having to run or profile the code. We propose a static analysis-based cross-architecture performance prediction framework (Static XAPP) which relies solely on program properties collected using static analysis of the CPU source code and predicts whether the potential speed-up is above or below a given threshold. We offer preliminary results that show we can achieve 94% accuracy in binary classification, in average, across different thresholds.

1. INTRODUCTION

Porting code from CPU to GPU is a slow and tedious process; Not only the program needs to be restructured to extract maximum parallelism, data organization needs to be re-arranged too in order to benefit from different levels of GPU memory hierarchy. A tool that can *quickly* and *accurately* predict the speed-up could be extremely useful. It can not only save programmers' time but also help with algorithm selection and prioritization. From a programmers' standpoint, heuristic-based performance estimations are far from accurate; GPU's architecture and programming paradigm are significantly different than CPU's and GPU programs' performance are very sensitive to events like branch divergence and memory divergence. This work belongs to the broad category of performance prediction literature, which can be categorized as follows:

- **Execution-based** techniques rely on dynamic binary instrumentation to obtain program properties. Binary instrumentation slows the program execution by 10-1000× which can be very costly depending on the original program execution time. Collected features can be fed into a performance prediction model, analytical or machine-learning, to obtain performance on a target machine [1, 2, 3, 4].
- **Human-based** approaches like Roofline model [5] and Boat-hull model [6] avoid the overhead of binary instrumentation but relies on humans to estimate features, and thus can be imprecise and slow.
- **IR-based** approach; we introduce this novel branch which relies merely on information available at the intermediate representation (IR) of a program, and thus avoid human involvement and slowdowns of binary instrumentation. We make this insightful observation that program properties obtainable with simple static analysis are sufficiently explanatory to predict cross-architecture performance. This observation does not imply that the correlation between static program properties and speedup is by any means straightforward. In fact, we require sophisticated machine learning algorithms to discover this correlation.

We envision this tool to be useful/integrated in different scenarios including:

- **Integrated Development Environments (IDEs)**; having an IDE environment where a developer can highlight a portion of the code to estimate the possible speed-up on a platform of choice can be highly useful.
- **Device Placement Optimization**: Device placement optimization algorithms can benefit from an accurate prediction of speed-up when a particular algorithm is executed on a specific platform. An execution-based method can significantly slow-down such an algorithm and a human-based method will require continuous feedback. A tool like ours can help quickly and without intervention, to determine the possible speed-ups for various devices.

2. METHODOLOGY

We are operating within a small dataset regime as the size of our dataset is very small (156 datapoints). We briefly explain our machine learning approach, including the preparation phase, model construction phase, the details of the training and test sets, and the software/hardware platforms used in evaluation.

Notations A *datapoint* is a pair of single-threaded CPU code and the associated GPU code. The CPU code is characterized in terms of its feature vector and the GPU code is used to measure the CPU-to-GPU speedup. A *feature vector* is the set of program properties, outlined in Section 3, estimated per CPU code and presented in the form of a binary vector.

Preprocessing Steps Compared to dynamic binary instrumentation, static analysis can be orders of magnitude faster. However, the estimated features can be less precise as they lack information about the dynamics of execution. We trade-off precision for accuracy by discretizing the estimated feature values into two to three levels, using the *equal frequency binning* algorithm. We also discretize the output value

*The work done while student at UW-Madison.

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

Table 1: Program statements

(speedup) into two ranges, low and high; from the developer's perspective, the decision to port a code to GPU rests more on the range of the speedup achievable (low or high) and less on the actual value of the speedup. However, depending on the importance of the kernel, what considered as a high speedup range for one case might be low for another. Therefore, we allow the user to denote the cutoff that breaks the speedup range into low and high. We use the user-provided cutoffs to label each datapoint in our training set before model construction.

Machine Learning Approach We employ the random forest (RF) algorithm to construct a speedup classifier. Our RF model is an ensemble of 1000 decision trees, where each tree is constructed using a random subset of features and training datapoints. We identify a set of 10 program properties that are sufficiently accurate using static analysis (see Section 3). Alternatively, this problem could have been formulated as an end-to-end deep learning problem, where the CPU source code could have been parsed through a recurrent or transformer model to implicitly discover the features and predict the performance on the target accelerator. We could not use this approach as we were operating in a small dataset regime.

Dataset We collect our datapoints from the widely-known GPU benchmark suites, including Lonestar [7], Rodinia [8], and NAS [9, 10]. The codes available in these suites are mainly well-suited datapoints for GPU by design, and thus our dataset is highly biased. To balance our dataset, we develop our own microbenchmarks and add some negative examples – obviously ill-suited codes for GPUs – to our dataset. Collectively, this effort will give us ~ 80 datapoints, which we refer to as *core kernels*. In order to increase our dataset size further, we use a set of tricks prescribed by [4]; we manually develop alternate CPU and GPU implementations by perturbing core kernels. For example, we add or subtract a piece of code that is well-suited or ill-suited for GPU, to both CPU and GPU implementations.

Evaluation We use *leave-one-out cross-validation* (LOOCV) to evaluate the accuracy of our technique, which is widely-used for evaluation of small dataset problems.

3. PROGRAM FEATURES

In traditional machine learning approach, we need to manually define the essential set of features required for characterizing a desired output. Here, we describe a generic CPU program model and an associated static analysis framework that computes a number of important program features for GPU speedup prediction.

3.1 Program Model

We will assume that we are given a sequential CPU program P in a standard representation (e.g., LLVM's intermediate representation). Program instructions are categorized as shown in Table 1. We will use MEM to denote the set of all memory load and store instructions that appear in P. Similarly, we will use ARITH to denote arithmetic operations in P, and CTRL to denote conditional branches.

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

Figure 1: Example CPU code

3.2 Program Features and Static Extraction

Assume for the moment that for a given a program P, the developer has annotated the region of the code—the loop or loops—they wish to parallelize. We call this region the *parallel band* (PBAND). We refer to the rest of the code enclosed within the PBAND, as *kernel body* (KBODY). Figure 1 explains this with a simple example. In this example, the outer for-loop is the parallel band – as indicated by *#pragma parallel SXAPP* – and the region enclosed within (line 3-11) is the kernel body. While our features are statically determinable, for the purposes of illustration, we will assume that we are given an input I of the program P. Using I, we can characterize the number of times an instruction s is executed as a function of I, which we call the *expected occurrence frequency* of s and denote by $f_I(s)$. Note that this function, f_I, can only be discovered dynamically. However, as we shall see in Appendix, our approach is robust to the values of f_I and we can elide f_I computation.

The set of (numerical) features computed from P is formally defined and described in Table 2. In what follows, we provide a thorough exposition of these features and the rationale behind choosing them. We note that, while these features are numerical, they will be later *discretized* automatically by our machine learning algorithms.

1. Memory coalescing is a high-impact feature on GPU speedup; it captures the possibility of global memory accesses to be coalesced. A non-coalesced memory access can reduce the global memory bandwidth efficiency to as low as 1/32, which negatively affects the speedup [11]. Specifically, this feature characterizes the percentage of memory instructions in the KBODY that are considered *coalesced*. We weight each operation $s \in$ MEM by its occurrence frequency $f_I(s)$. Given a memory operation $s \in$ MEM, we consider $coalesced(s)$ to be true *iff* one of the following holds: (1) Its memory index expression is loop-invariant with respect to all the loops within the PBAND. Intuitively, this means that all threads access the same memory location. (2) Its memory-index expression is loop-invariant with respect to all the loops within the PBAND, except the innermost one. The innermost-loop induction variable should appear with a $multiplier \leq 1$ in the memory-index expression. Intuitively, this means that consecutive threads are accessing to consecutive or same

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

Table 2: Program features, their formal definition, and how they impact GPU speedup

memory locations. In our running example in Figure 1, there are two memory operations: `tree[0]` is coalescable, as the memory index is loop-invariant; `tree[j]` is considered non-coalescable, as the memory index j depends on *key* which depends on i, the induction variable of the loop in PBAND.

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

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

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

5-10. Instruction intensities The lower part of Table 2 contains features that measure whether the CPU code, when ported to GPU, will exploit the strengths of GPUs. For instance, the *arithmetic intensity* feature is a measure of how well the arithmetic operations can hide memory latency, and is defined as the ratio of the number of arithmetic operations to the number of memory operations. To estimate the number of memory operations/arithmetic operations statically, we weigh each operation s by its occurrence frequency $f_I(s)$.

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

	Predicted: Low	**Predicted: High**	
Actual: Low	TN: 61	FP: 7	68
Actual: High	FN:1	TP: 87	88
	62	94	

Accuracy = (TP+TN)/(TP+TN+FP+FN) = 94%
Positive Predictive Value (PPV) = TP/(TP+FP) = 93%
Negative Predictive Value (NPV) = TN/(TN+FN) = 98%

Table 3: Binary classifier accuracy (speedup cutpoint = 3)

3.3 Expected Occurrence Frequency

The above feature extraction assumed the existence of a function f_I that specifies the expected occurrence frequency of each program instruction. While f_I is not statically determinable, we have empirically validated that our model is robust to changes in f_I. Specifically, the expected occurrence frequency of an instruction is a function of (1) loop-trip counts of loops enclosing the instruction, and (2) the probability of taking branches that lead execution to the instruction. In Appendix A, we show that our technique is robust to variation in loop-trip count and branch probability and it can predict speedup with 91% accuracy, with no knowledge about the dynamic input, using a simple heuristic.

4. RESULTS AND ANALYSIS

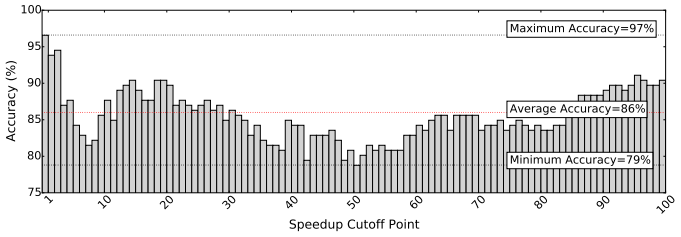
In what follows, we first show the accuracy of our model for *a* binary speedup classifier. Next, we show our technique is robust across different cutoffs and platforms. Finally, we analyze accuracy for a multiclass classifier.

4.1 Model Accuracy

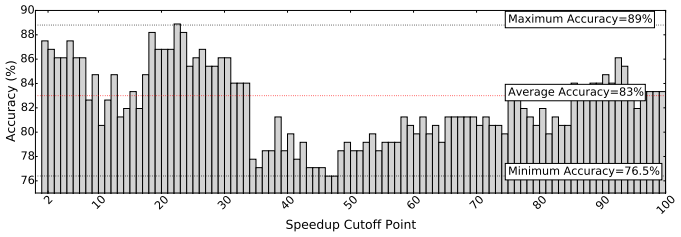
Table 3 summarizes the accuracy results for a speedup classifier with the cutoff at 3. We classify the speedup as low or high with 94% accuracy. The Positive Predictive Value (PPV) and Negative Predictive Value (NPV) are 93% and 98%, respectively. The high NPV value suggests that our tool is very effective in saving programmers' time from porting a low-speedup application to GPU.

4.2 Model Stability

To study the impact of cutoff choice on accuracy, we vary the speedup cutoff values from 0 to 100 in steps of 1. For



(a) Platform 1 (GTX750)



(b) Platform 2 (GTX660)

Figure 2: Model Stability. The x-axis represents the cutoff point that divides the speedup range into low and high. Kernels with speedup $\leq$ x will be labeled as low (L) and kernels with speedup $>$ x will be labeled as high (H). The y-axis shows the cross-validation accuracy for a model that is constructed with a dataset labeled as such.

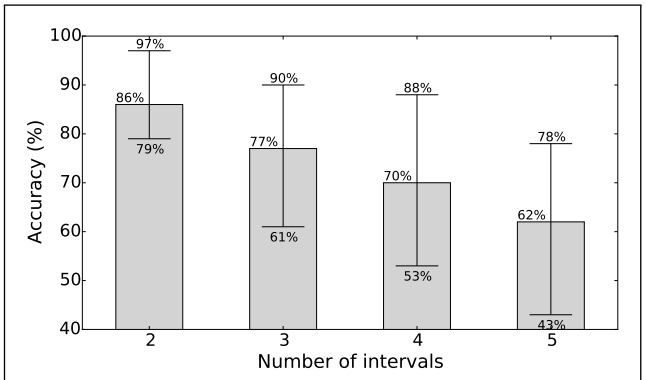
	Platform 1	**Platform 2**
Microarchitecture	Maxwell	Kepler
GPU model	GTX 750	GTX 660 Ti
# SMs	7	14
# cores per SM	192	192
Core freq.	1.32 GHz	0.98 GHz
Memory freq.	2.5 GHz	3 GHz
CPU model:	Intel Xeon Processor E3-1241 v3 (8M Cache, 3.50 GHz)	

Table 4: Hardware platforms pecifications.

each cutoff, we relabel our dataset and construct a new model, and measure its LOOCV accuracy. Figure 2(a) shows the prediction accuracy for different cutoffs on one GPU platform (platform 1 in Table 4). As shown, our technique maintains minimum, average and maximum accuracy of 79%, 86% and 97%, respectively. Note here that the slight differences in accuracy across different cutoffs is partly due to changes in the number of datapoints within each interval. Too many or too little datapoints in a bin can bias the model and hurt the generalization accuracy. Figure 2(b) shows similar results for another GPU platform (Platform 2 in Table 4). Since speedup distribution is different across different platforms, we observe different accuracy results for different speedup cutoffs. Our technique maintains minimum, average and maximum accuracy of 76.5%, 83% and 89%, respectively, on the second platform. In conclusion, our technique is robust to variations in cutoffs and platforms.

4.3 Multi-class classification

We also study if we can predict speedup at a finer granularity, in other words, classifying the speedup in more than two bins. Figure 3 represents the minimum, maximum and average prediction accuracy, as we increase the number of bins from 2 to 5. The minimum, maximum and average accuracy are measured across different models constructed with different speedup cutoffs. For instance, the second bar (3 intervals) represents the accuracy across all models constructed with two speedup cutoffs, (x_1,x_2), where x_1 varies from 1



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

Figure 3: Multiclass classification accuracy. The Table below shows the range of cutoffs each bar is averaged across. (l,u,s) at row i and column j shows that cutoff x_i for j intervals sweeps between l and u in steps of s.

to 20 in steps of 1 and x_2 varies from x to 100 in steps of 1. As expected, the model accuracy drops as the number of intervals (classes) increases. This is expected as we get less datapoints in each interval.

5. RELATED WORK

The application of program analysis in cross-platform performance prediction has been previously explored, primarily in the context of design space exploration [12, 13, 14, 15, 16, 17], finding the best CPU platform amongst many CPU microprocessors with different ISAs based on program similarity [1, 18, 2], understanding performance bottlenecks of multicore architectures [5], and finding GPU acceleration based on CPU implementation [19, 20, 3, 4]. As discussed in Section 1, all of these studies are either execution-based, which introduces 10-100$\times$ slowdown, or human-based, which is slow and imprecise. Compiler community has explored techniques to automate GPU code generation from CPU code [21, 22, 23, 24]. However, their scope of applicability is limited to affine programs. Hoshin et. al. [25] shows that GPU codes generated OpenACC are 50% slower than hand-optimized ones. Static analysis has been previously used in the context of program optimization to predict the impact of an optimizion on performance [26]. Many researchers have investigated GPU design space exploration and performance prediction based on GPU program properties [27, 28, 29, 30]. However, these techniques require a GPU code to start with.

6. CONCLUSION

In this paper we have developed a new speedup prediction technique that relies only on the source code. It has been believed that program properties needed for predicting GPU speedup must necessarily be obtained from the dynamic

execution of the program. Our paper makes a fundamental intellectual contribution in demonstrating that statically determinable program properties are sufficiently explanatory for developing a machine-learning based speedup predictor.

7. REFERENCES

[1] R. H. Saavedra and A. J. Smith, "Analysis of benchmark characteristics and benchmark performance prediction," *TOCS*, 1996.

[2] K. Hoste, A. Phansalkar, L. Eeckhout, A. Georges, L. K. John, and K. D. Bosschere, "Performance prediction based on inherent program similarity," in *PACT*, 2006.

[3] I. Baldini, S. J. Fink, and E. Altman, "Predicting gpu performance from cpu runs using machine learning," in *SBAC-PAD '14*.

[4] N. Ardalani, C. Lestourgeon, K. Sankaralingam, and X. Zhu, "Cross-architecture performance prediction (xapp): Using cpu code to predict gpu performance," in *MICRO*, ACM, 2015.

[5] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Commun. ACM*, 2009.

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

[7] M. Kulkarni, M. Burtscher, C. Casçaval, and K. Pingali, "Lonestar: A suite of parallel irregular programs," in *ISPASS*, 2009.

[8] 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*.

[9] 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," *IJHPCA*, 1991.

[10] L. L. Pilla, "NAS Parallel Benchmarks CUDA version." `http://hpcgpu.codeplex.com`.

[11] "Cuda Toolkit Documentation." `http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/`.

[12] J. J. Yi, D. J. Lilja, and D. M. Hawkins, "A statistically rigorous approach for improving simulation methodology," in *HPCA*, 2003.

[13] H. Vandierendonck and K. De Bosschere, "Many benchmarks stress the same bottlenecks," in *Workshop on Computer Architecture Evaluation Using Commercial Workloads*, 2004.

[14] E. Ïpek, S. A. McKee, R. Caruana, B. R. de Supinski, and M. Schulz, *Efficiently exploring architectural design spaces via predictive modeling*. 2006.

[15] W. Wu and B. C. Lee, "Inferred models for dynamic and sparse hardware-software spaces," in *MICRO*, 2012.

[16] B. C. Lee and D. M. Brooks, "Accurate and efficient regression modeling for microarchitectural performance and power prediction," in *ASPLOS*, 2006.

[17] B. Lee and D. Brooks, "Illustrative design space studies with microarchitectural regression models," in *HPCA*, 2007.

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

[19] J. Meng, V. Morozov, K. Kumaran, V. Vishwanath, and T. Uram, "Grophecy: Gpu performance projection from cpu code skeletons," in *SC*, 2011.

[20] 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," *IJHPC*, 2013.

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

[22] S.-Z. Ueng, M. Lathara, S. S. Baghsorkhi, and W. H. Wen-mei, "CUDA-lite: Reducing gpu programming complexity," in *LCPC*, 2008.

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

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

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

[26] C. Dubach, J. Cavazos, B. Franke, G. Fursin, M. F. O'Boyle, and O. Temam, "Fast compiler optimisation evaluation using code-feature based performance prediction," in *CF*, 2007.

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

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

[29] W. Jia, K. A. Shaw, and M. Martonosi, "Starchart: hardware and software optimization using recursive partitioning regression trees," in *PACT*, 2013.

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

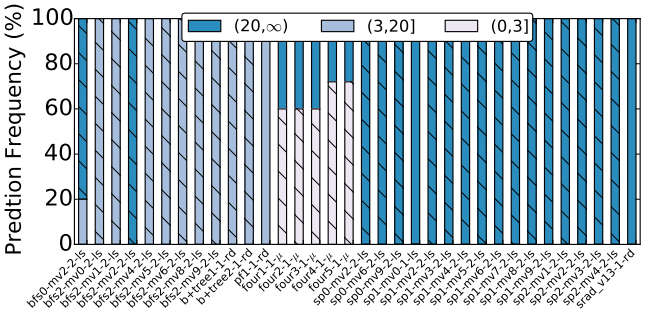


Figure 4: Speedup prediction sensitivity to branch ratio.

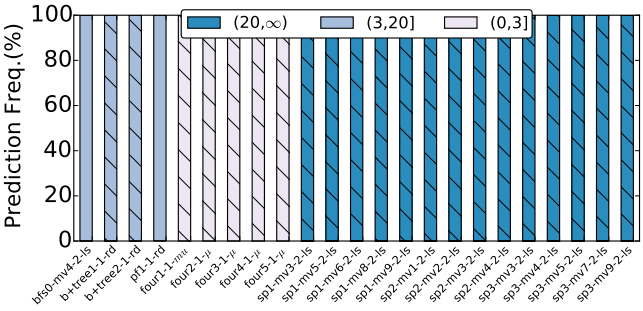


Figure 5: Speedup prediction sensitivity to loop trip count.

Appendix A. Impact of Input Dynamics on Speedup Range Prediction

The dynamic value of program properties defined in Section 3 is primarily controlled by two input-dependent factors: the number of iterations of each loop (also referred to as loop trip count) and the probability of each branch being taken (also referred to as branch taken probability). We make this insightful observation that speedup is mostly robust to changes in the loop trip count and branch taken probability. Therefore, a simple heuristic such as the majority vote selection over different predicted values of speedup for different values of loop trip counts and branch taken probabilities is a good enough estimator of the actual speedup range. Specifically, we sweep each loop's trip count from 10 to 1000 and each branch taken probability from 0% to 100% and predict the speedup for each value of trip count and branch taken probability and get the majority vote over the predicted values. We show that this simple technique is surprisingly sufficient to capture the impact of program dynamics, and thus we can predict speedup with 91% accuracy, with no knowledge about the dynamic input. The reason the feature vectors, and consequently speedup range is robust to variations in dynamic variables can be summarized as follows: (1) The features are defined as ratios of two dynamic events, and usually numerator and denominator scale similarly when dynamic variable changes. (2) They are discretized, which makes the discretized feature values robust to small changes in their actual value. (3) Cutpoints are close to the extreme ends, so the change in dynamic value of the feature usually keep it within the same region. (4) Variations in the dynamic features that control performance, often affect CPU and GPU execution time in the same direction, and therefore speedup range remains unchanged.

Sensitivity to Loop Trip Count Figure 4 shows the distribution of speedup predictions while we vary the branch probability for each branch from 0% to 100% in steps of 25%. On the X-axis, we have all the kernels in our dataset that have at least one conditional branch within their kernel body. Each stack bar represents the probability that a speedup prediction belongs to any of the following speedup ranges, $(0,3)$, $(3,20]$, and $(20,\text{inf})$. Specifically, we use our static-analysis tool to estimate the feature vector per a branch probability combination, and feed it into our speedup prediction model to get one speedup range prediction per a branch probability combination. To give an example, $b+tree1-1-rd$, a kernel with three conditional branches, gets 5^3 feature vectors and 5^3 speedup range predictions. Across all 125 predictions, the majority vote heuristics suggests that the speedup belongs to 3 to 20 range. To fact check this, we show the actual speedup value for each application by hatching the corresponding stack bar. As shown, in all but three cases the actual speedup range matches with the majority prediction. This implies that majority vote heuristic is sufficient to predict the speedup range with 91% accuracy, with no knowledge about the branch probabilities.

Sensitivity to Branch Taken Probabiltiy Figure 5 shows the similar results for loop trip count. On the X-axis, we have all the kernels in our dataset that have at least one loop within their kernel body. We vary the values of trip-counts from 1 to 1000 in logarithmic steps and use majority vote prediction to predict the speedup. As shown, in all but two cases the actual speedup range matches with the majority vote prediction. This indicates that majority vote heuristic is sufficient to predict the speedup range with 91% accuracy, with no knowledge about the loop trip counts.