Alexander Lee (ahl256)
GPUs, Spring 2022
4/22/2022

Final Project: Cross-GPU Kernel Performance Prediction

Abstract

Since their introduction as discrete hardware devices, GPUs have expanded beyond their initial role as graphics accelerators and have become a commonplace coprocessor for high-performance heterogeneous computing, particularly for applications exhibiting strong data parallelism. GPU hardware has evolved rapidly in response to demand from these additional workloads, and introduction of new hardware offers the potential for increased performance for applications developed with parallel processing in mind. The ability to predict performance of GPU-accelerated applications across different hardware may thus help practically inform hardware acquisition and application deployment strategies.

This paper describes a modeling framework using a combination of hardware and software features extracted from scientific applications in the Rodinia benchmark suite, using a variety of statistical learning models to predict application execution time across GPU contexts. The model results are captured across a variety of sampling strategies and are compared for both performance and stability. Prediction for "real-world" contexts is emphasized, and while the experimental setup is naturally limited in scope, the best-performing model is able to achieve accuracy scores (measured in terms of R^2 values for test predictions) of 0.73 to 0.97 in the most realistic sampling scenarios while exhibiting several markers of model stability.

I. Introduction

As the market for GPUs has broadened and vendor support for enabling general-purpose APIs for GPU hardware has matured, heterogenous hardware systems incorporating general-purpose GPUs (GPGPUs) for high-performance computing have become more widespread. Various programming frameworks for systems incorporating parallel hardware including CUDA, OpenACC, OpenCL, and OpenMP remain in active development, and hardware manufacturers continue to refine and redesign GPU architectures in response to market demand. As interest in leveraging parallel computing hardware in various application domains continues to grow in tandem with hardware developments, the ability to predict parallel application performance on new or alternative hardware may help guide decisions related to technology investment and inform tradeoffs involved in deciding on what hardware to deploy particular applications.

This paper presents a model that attempts to predict performance of a parallel application (for which the term "kernel" may be used interchangeably in the discussion) using a combination of hardware features and both static and dynamic features of the kernel itself. The remainder of the paper is organized as follows: Section II surveys literature on related performance prediction problems; Section III expands upon the modeling problem and reviews some related challenges; Sections IV – VI describe the modeling approach and experimentation conducted for this research; Section VII concludes.

II. Survey

The existing body of literature on performance prediction questions concerning GPGPUs is quite broad, and falls into at least two major discrete categories: performance prediction for a specific architecture under varying conditions (for example, [1], [2], [3]), and cross-architecture performance prediction, whether across GPUs (as in [4], [5]) or across the CPU-GPU gap (as in [6], [7]). Similar design questions arise in other parallel computing contexts, as in [8] discussing parallel application performance prediction on multicore processors, and some research extends the modeling exercise to the realm of hardware/software co-design, as in [9] where the authors attempt to model the impact of specific hardware architectural features to help predict performance of future hardware designs, or [10] where GPU performance prediction metrics are used in design space exploration.

Modeling techniques vary across this research, though most models fall under either the general category of simulation or that of statistical (or machine learning) predictive modeling. Features of the models vary as well, but tend to include elements of specific hardware (or simulations thereof), elements of the software to be run on parallel hardware, or both. This paper falls under the cross-architecture performance prediction category, and attempts to model cross-GPU performance prediction using a combination of hardware and software features, which are discussed in depth in Sections IV and V below.

III. Problem Definition and Challenges

Abstractly, the research aim of this paper is to attempt to predict the performance of a kernel on a GPU given information about its performance on a different GPU. This gives rise to a number of additional questions which must be answered first, including how to model the kernel and the hardware features, how to define “performance,” how to select a specific predictive model, and how to evaluate results. An additional challenge was raised by the nature of the experimental context, for which only 5 different GPUs (all from the same vendor) were available.

In light of these questions and following some initial experimentation, the concrete aim of this paper is to answer the following question: given available numerical and dummy data extracted from a software kernel and hardware data related to the specifications of various GPU devices (net of the limitation mentioned above), can we predict end-to-end execution time of an application designed for parallel execution on a system running a GPU device not included in the parameters of the model for that application? The question aims to consider performance in “real-world” contexts where applications will be subject to Amdahl’s law, though the benchmark suite of applications used for experimentation is designed to take advantage of parallel hardware and presumably exhibits selection bias towards applications where that parallelism may afford worthwhile speedup over sequential execution. The subsequent section explores in greater detail how the experiment was designed before turning to its execution and results in the remainder of the paper.

IV. Proposed Idea

The proposed general model is comprised of three categories of features, with parallel application execution time as the target variable. The first category is hardware features related to the available devices in the experimental setup. The latter two categories are software features of selected parallel applications, bucketed into “static” software features related to the application source code, and “dynamic” software features extracted from profiling the code in execution. The model incorporates both hardware and software features in the expectation that the interaction of these features will dictate execution time of the applications, and in all three categories, features were selected for variability (in an attempt to avoid multicollinearity) and representativeness of expected exploitation of parallelism (or potentially missed opportunity for such).[1]

The hardware features selected for the model are a subset of information obtained from querying the five available devices in the experimental environment.[2] A summary of these devices is in Table 1 below, followed by the hardware features extracted from each of the devices in Table 2:

GPU	Architecture	CC	Launch Year
NVIDIA GeForce GTX TITAN Black	Kepler	3.5	2014
NVIDIA GeForce RTX 2080 Ti	Turing	7.5	2018
NVIDIA TITAN V	Volta	7.0	2017
NVIDIA GeForce GTX TITAN X	Maxwell 2.0	5.2	2015
NVIDIA GeForce GTX TITAN Z	Kepler	3.5	2014

Table 1: GPUs used for model feature collection and calculation of target execution times.

Feature Name	Feature Type	Explanation
hw_cc_52	dummy	Flag for CC >= 5.2
hw_cc_70	dummy	Flag for CC >= 7.0
hw_cc_75	dummy	Flag for CC >= 7.5
hw_total_global_mem_kb	numeric	Total global memory in KB
hw_clock_rate_khz	numeric	Execution unit clock rate in KHz
hw_multiprocessor_count	numeric	Number of SMs
hw_async_engine_count	numeric	Number of async compute engines
hw_memory_bus_width	numeric	Memory bus width in bits
hw_memory_clock_rate_khz	numeric	Memory clock rate in KHz
hw_l2_cache_size_bytes	numeric	Level 2 cache size in bytes
hw_max_threads_per_sm	numeric	Maximum thread count per SM

Table 2: Hardware features proposed for modeling.

[1] By "expected exploitation of parallelism," it is meant that the features either reflect in some way to the parallel processing power of the GPU itself (speed or number of execution units, memory available to execution units, etc.), or are software elements specifically attempting to take advantage of parallel hardware, such as loop depth of kernel calls, usage of features like shared memory or streams, or GPU API calls performed during test execution.

[2] All devices used for experimentation were NVIDIA GPUs resident on the NYU Courant CUDA cluster, and were used for collecting all features of the model data except for static software features, which were obtained by inspection of the source code locally.

These features were selected as a subset of the queried device attributes that reflected both variability across the available hardware (other potentially relevant features related to shared and constant memory, register availability, and grid dimensionality were discarded as they were equal across the five available devices), and potential for meaningful parallel performance impact on a software kernel. The compute capability (CC) of each device was dummied out to avoid malign interpretation as a linear feature, though it was retained in general as a potentially suitable proxy for additional performance-related features not captured elsewhere.

All software features were extracted from 17 applications available in version 3.1 of the Rodinia benchmark suite [11]. While the Rodinia suite covers heterogeneous programming generally rather than applications specifically targeting GPUs, the suite was selected as representative of real-world applications spanning a variety of scientific domains and computational techniques (which subsequently place different types of demands on the parallel hardware), and all applications in the suite have an implementation in CUDA.[3] A summary of the applications from which software features were extracted is presented in Table 3 below:

Application Name	Computational Technique	Problem Domain
b+tree	Graph Traversal	Search
backprop	Unstructured Grid	Pattern Recognition
bfs	Graph Traversal	Graph Algorithms
dwt2d	Spectral Method	Image/Video Compression
gaussian	Dense Linear Algebra	Linear Algebra
heartwall	Structured Grid	Medical Imaging
hotspot	Structured Grid	Physics Simulation
huffman	Finite State Machine	Lossless Data Compression
lavaMD	N-Body	Molecular Dynamics
lud	Dense Linear Algebra	Linear Algebra
myocyte	Structured Grid	Biological Simulation
nn	Dense Linear Algebra	Data Mining
nw	Dynamic Programming	Bioinformatics
particlefilter	Structured Grid	Medical Imaging
pathfinder	Dynamic Programming	Grid Traversal
srad_v2	Structured Grid	Image Processing
streamcluster	Dense Linear Algebra	Data Mining

Table 3: Software applications selected for feature extraction from Rodinia suite.

[3] While the initial intent was to include data from all kernels in the suite within the model, not all were able to be compiled by the author on all of the available GPU hardware. The kernels included in the model data cover all computational techniques present in the suite except for sorting.

The static software features extracted from each of these applications are listed in Table 4 below:

Feature Name	Feature Type	Explanation
sw_s_sloc_total	numeric	Total lines of code in application
sw_s_sloc_kernel	numeric	Lines of kernel code in application
sw_s_pct_ksloc	percentage	Percentage of total code in kernel
sw_s_shared_use	dummy	Flag for use of shared memory
sw_s_const_use	dummy	Flag for use of constant memory
sw_s_num_kern	numeric	Number of kernels launched
sw_s_looped_kern	dummy	Flag for loop of kernel
sw_s_loop_in_kern	dummy	Flag for loops within kernel
sw_s_max_lik_depth	numeric	Maximum loop depth within kernel
sw_s_approx_regs_pt	numeric	Approximate registers used per thread

Table 4: Static software features extracted from Rodinia applications.

These features were chosen after manually inspecting the source code of several kernels for elements that potentially reflected advantageous (or potentially disadvantageous) uses of parallelism. The feature for percentage of source lines of code within the actual parallel kernel is a (somewhat crude) attempt to capture the effects of Amdahl's law in the model, and was inspired by a similar technique illustrated in [12]. While tracking both the lines of code and the percentage may seem redundant, all were retained for the modeling phase to allow the relative magnitude of the former to potentially proxy for general application complexity, while the latter proxies for Amdahl's law more specifically. Different applications exhibited different behavior in handling loops in and around kernels, so numerous features were dedicated to these varied approaches to usage of the parallel hardware resources. The count of approximate registers per thread was made manually (and may thus contain some inaccuracies); in the case that an application contained multiple kernel launches, values were averaged over kernels.

The initial idea for the model was to use only hardware and static software features, but after extracting the features in Table 4 for the 17 Rodinia applications used for modeling, they were deemed likely insufficient to capture meaningful differences among the software applications on their own. Thus, the software features were extended to include dynamic features captured via the *nvprof* tool from NVIDIA [13]. The dynamic features included in the model are listed in Table 5 below:

Feature Name	Feature Type	Explanation
sw_d_multi_dev	dummy	Flag for multiple device use
sw_d_memcpy_d2h	numeric	Number of device-to-host memcpys
sw_d_memcpy_h2d	numeric	Number of host-to-device memcpys
sw_d_cudafree	numeric	Number of cudaFree calls
sw_d_cudalaunchkernel	numeric	Number of kernel launches
sw_d_cudamalloc	numeric	Number of cudaMalloc calls
sw_d_cudamemcpy_s	numeric	Number of synchronous memcpys
sw_d_cudamemcpy_a	numeric	Number of asynchronous memcpys
sw_d_cudamemset	numeric	Number of memset calls
sw_d_stream_use	dummy	Flag for stream use
sw_d_cudathreadsync	numeric	Number of thread sync calls

Table 5: Dynamic software features extracted from Rodinia applications.

The dynamic features were narrowed down from the superset of profiled calls captured by *nvprof* such that there was meaningful representation of the features across the application suite, and in certain cases, features were collapsed from numeric to dummy variables to simplify the data before modeling. The dynamic features primarily reflect application memory usage, along with computational features related to thread management and kernel launch patterns. While only one application (dwt2d) appeared to utilize multiple devices, this behavior was retained as a dummy variable as a deliberate example of attempting to exploit an additional level of parallelism across the hardware.

After determination of these three classes of features, the planned approach was to analyze the results across a range of statistical and machine learning models for efficacy, and perhaps equally importantly, stability across various application domains and tested hardware. The manner in which the experiments were set up across the modeling suite is discussed next in Section V. The specific models chosen and analysis of results across these models are discussed in Section VI.

V. Experimental Setup

The setup of the experiment prior to performing the actual modeling and analysis consisted of three core phases. The first phase consisted of feature and target data collection, on which the models would ultimately be fitted. Hardware information was gathered from the five GPUs listed in Table 1 using NVIDIA APIs for querying devices in their respective cluster computing environments. Static software features were extracted by examining source code on the author's local machine, and the "linecounter" tool [14] (available as an extension for the VSCode editor) was used extensively for computing the first three features listed in Table 4 related to source code line counts. The count of approximate registers per thread was performed manually by inspecting kernels for parameters and internally declared variables; due to the completely manual nature of feature extraction and its relative complexity compared to other manually-extracted features like kernel loop depth, it is likely subject to greater potential inaccuracy than any other feature used in the model. The dynamic software features were obtained from running the *nvprof* tool from NVIDIA on each of the 17 applications. These features were all extracted from the NVIDIA

GeForce RTX 2080 Ti hardware.[4] The target variable of end-to-end application runtime was obtained by compiling and running each kernel on each of the available hardware devices.[5] In all cases, the values used for target times were the average of five trials. Of note, the devices from which target times were collected are available for simultaneous access and use rather than being reservable resources; thus, performance measurements may have been affected by other jobs being run on the hardware that were transparent to the author. To ameliorate this issue somewhat, all target times were collected during a single contiguous multi-hour window; however, other workloads may have influenced the target times and thus potentially altered the implications of the model results. While this issue was generally unavoidable due to the experimental context, it is worth remarking on as one of the potentially most relevant exogenous factors influencing the behavior of the models.

Following the data collection phase was a data processing phase in which features were winnowed, reconfigured, and rescaled before modeling. The winnowing and reconfiguration of features used a combination of manual inspection, statistical modeling, and common sense. Features that were initially considered for inclusion in the model but which upon data collection exhibited limited or no variability across devices or applications and thus limited or no information content for a model to use were eliminated. Other features were collapsed from numeric values to dummy values as appropriate, either to avoid having the model "misinterpret" the feature (as in the case of compute capability, described earlier), or to simplify the model in some fashion (e.g., collapsing several dynamic features related to stream management into a single dummy variable indicating usage of streams). After a first pass of manual winnowing and feature reconfiguration in this manner, features were rescaled in Python using a scaler robust to outliers from the Scikit-learn (*sklearn*) package [15] (which was also used for the modeling phase), while the target variable was left as time measured in seconds for interpretability.[6] A second pass of statistically-based winnowing was conducted at this point, using significance of the coefficients in an OLS model of the scaled data to manually select features for exclusion. Multiple variations on this manual feature elimination were conducted and tested, and ultimately a relatively

[4] This was primarily a matter of convenience, as the order in which the applications were compiled and run to extract the target variable led to the code being compiled last for this hardware; profiling features were extracted after target values had been collected. Spot checking against different hardware did not reveal meaningful differences in these values; however, a more comprehensive check might reveal discrepancies in some of these counts across hardware and thus these values should be considered subject to some possible inaccuracy when used as features for data records that incorporate hardware features from different GPUs.

[5] In general, the kernels were compiled using provided Makefiles included with the Rodinia applications. Several of these Makefiles specified target hardware architecture flags; when present, these were changed for each target device and recompiled for that device's architecture (obtained during hardware feature collection). In some cases, no such flag was specified, and compiler default settings were used instead. Regardless of whether the kernel was compiled for the specific target architecture or used default settings, the applications exhibited generally stable relative performance (in terms of average execution time) across the five target devices. Thus, while the comparison would be conceptually improved by performing customized builds for all kernels, it may not have material impact on modeling results. Also of note is that times were collected for the Rodinia applications running in their default configurations; as the author is not an expert in the various scientific domains illustrated by these applications (as listed in Table 3), they did not wish to attempt to reconfigure the applications and inadvertently devise an unrealistic test case. This does highlight an embedded assumption that the default configurations of the Rodinia applications are in fact reflective of reasonable scientific use cases.

[6] This robust scaler appeared to lead to better downstream model performance than using a standard normalizing scaler, indicating the presence of some outliers in the raw feature set.

conservative approach was used to eliminate a small number of features from the model.[7] The dataset prior to this final winnowing phase consisted of 32 features over 85 samples, and after winnowing consisted of 29 features.

During the final phase of the experimental setup, the model data was divided into training and testing sets in preparation for the modeling phase. Three different approaches were taken for this phase, and modeling was conducted for each downstream. The first approach was to hold out a single GPU's results from the training dataset for use during testing. This approach simulates the problem of wanting to estimate the performance of one or more applications with existing known performance on certain hardware devices on a new, untested device. From the perspective of considering a research organization potentially investing in new GPU hardware, this may be considered the most realistic scenario. Thus, for this approach, each individual GPU was held out as a candidate test set, leading to five train/test splits in total.[8] The second approach was to hold out one results from one GPU at random for each given application. This attempts to simulate a similar scenario as the first approach from the perspective of a single application, while "stretching" the (somewhat limited) data to cover more hardware cases. While the model is able to "see" results across all of the available hardware, it has no universal picture of any one application's performance across devices. The final approach simply holds out a random 20% sample across the entire dataset, allowing the model to see such a universal picture for at least some kernels. While all testing data remains for an unseen combination of application and hardware, the model generally speaking has the most information available to it in this scenario, which may be less realistic in real-world cases.

VI. Experiments and Analysis

After completing the feature and target data collection phases and the preprocessing of the data as described in Section V, model fitting and results analysis was conducted on the author's local machine. As mentioned in Section IV, the data were fitted on a selection of models spanning a variety of statistical learning techniques. Due to the continuous nature of the target variable (execution time), all models were varieties of regression models. Simple linear regression was used as a baseline, beyond which a mix of regularized models, alternative regressors, and Bayesian models were fitted for comparison. A list of the models used during this phase is in Table 6 below:

[7] All later modeling exercises were conducted against the feature sets both before and after this final statistical winnowing pass, and models were analyzed for stability of results across the two sets of features.

[8] One caveat regarding this approach is that two of the five GPUs on which model data were collected are fairly similar to one another (the GeForce GTX TITAN Black and GeForce GTX TITAN Z, both launched the same year using the same hardware architecture, as seen in Table 1). Thus, when holding out results from either of these devices but retaining results from the other in the training data, test performance might be expected to benefit somewhat from "correlation" between the hardware.

Model Name	Description
LinearRegression	Simple OLS regression
RidgeCV	L2 regularized regression
LassoCV	L1 regularized regression
ElasticNetCV	Mixed L1 / L2 regularized regression
LarsCV	Least angle regression
BayesianRidge	Bayesian L2 regularized regression
ARDRegression	Bayesian regression with alternative prior over coefficients

Table 6: Models fitted for results analysis.

All models were fitted on the feature data before and after the statistically-guided manual winnowing pass described in Section V. The modeling pipeline was implemented using the *sklearn* package for Python, during which time manual parameter tuning was conducted for certain models. Accuracy and error results from were collected and aggregated from all seven candidate models for analysis, and for meta-analysis of stability across models and sampling contexts. These results are presented in Tables 7 and 8 below, followed by a discussion of the results.

		Scenario (train / test split methodology)						
		cuda1_exclude	cuda2_exclude	cuda3_exclude	cuda4_exclude	cuda5_exclude	par_exclude	rand_exclude
LinearRegression	Score (R)	0.8799	-0.2029	0.7418	0.6781	0.8649	0.7323	0.9030
	Score (W)	0.8801	-1.0995	0.7418	0.7243	0.8650	0.7323	0.9030
RidgeCV	Score (R)	0.8808	0.6609	0.8543	0.9465	0.8635	0.7255	0.9033
	Score (W)	0.8810	0.6599	0.8349	0.9465	0.8635	0.7255	0.9033
LassoCV	Score (R)	0.7594	0.4565	0.7942	0.8572	0.7739	0.4235	0.8443
	Score (W)	0.7594	0.4565	0.7942	0.8572	0.7739	0.4235	0.8443
ElasticNetCV	Score (R)	0.7594	0.4565	0.7942	0.8572	0.7739	0.4235	0.8443
	Score (W)	0.7594	0.4565	0.7942	0.8572	0.7739	0.4235	0.8443
LarsCV	Score (R)	0.7948	0.5551	0.8037	0.8530	0.7530	0.5077	0.8591
	Score (W)	0.7948	0.5479	0.7989	0.8641	0.7542	0.5077	0.8568
BayesianRidge	Score (R)	0.7575	0.4353	0.8016	0.8604	0.8604	0.7093	0.9033
	Score (W)	0.7575	0.1598	0.8017	0.8604	0.8606	0.7103	0.9034
ARDRegression	Score (R)	0.8971	0.7321	0.9275	0.9738	0.8699	0.7109	0.8978
	Score (W)	0.8966	0.6991	0.9277	0.9732	0.8697	0.7095	0.8980

Table 7: Accuracy scores by model, data preparation scenario, and fitted feature set.

		Scenario (train / test split methodology)						
		cuda1_exclude	cuda2_exclude	cuda3_exclude	cuda4_exclude	cuda5_exclude	par_exclude	rand_exclude
Training data sample statistics	Target mean	0.9453	0.9427	0.8575	0.9207	0.8666	0.9318	0.7862
	Target SD	1.2720	1.3085	1.2344	1.2239	1.1338	1.3060	0.8485
LinearRegression	MAE (R)	0.3143	0.9517	0.4709	0.7024	0.3161	0.1830	0.3251
	MAE (W)	0.3140	1.2484	0.4710	0.6468	0.3158	0.1830	0.3251
	MAE SD (R)	0.2471	0.7273	0.3814	0.5739	0.2788	0.1401	0.3831
	MAE SD (W)	0.2468	0.9541	0.3815	0.5285	0.2786	0.1401	0.3831
RidgeCV	MAE (R)	0.3142	0.3152	0.3975	0.2429	0.3160	0.2032	0.3347
	MAE (W)	0.3139	0.3215	0.4235	0.2414	0.3158	0.2033	0.3341
	MAE SD (R)	0.2470	0.2409	0.3220	0.1985	0.2787	0.1556	0.3944
	MAE SD (W)	0.2468	0.2457	0.3431	0.1972	0.2786	0.1556	0.3938
LassoCV	MAE (R)	0.4773	0.4914	0.3783	0.4191	0.4772	0.5108	0.6066
	MAE (W)	0.4773	0.4914	0.3783	0.4191	0.4772	0.5108	0.6066
	MAE SD (R)	0.3753	0.3755	0.3065	0.3424	0.4209	0.3911	0.7150
	MAE SD (W)	0.3753	0.3755	0.3065	0.3424	0.4209	0.3911	0.7150
ElasticNetCV	MAE (R)	0.4773	0.4914	0.3783	0.4191	0.4772	0.5108	0.6067
	MAE (W)	0.4773	0.4914	0.3783	0.4191	0.4772	0.5108	0.6067
	MAE SD (R)	0.3753	0.3755	0.3065	0.3424	0.4209	0.3911	0.7151
	MAE SD (W)	0.3753	0.3755	0.3065	0.3424	0.4209	0.3911	0.7151
LarsCV	MAE (R)	0.4370	0.4562	0.3821	0.4243	0.4872	0.4722	0.5327
	MAE (W)	0.4370	0.4601	0.3839	0.4126	0.4861	0.4722	0.5528
	MAE SD (R)	0.3436	0.3486	0.3095	0.3466	0.4297	0.3616	0.6278
	MAE SD (W)	0.3436	0.3516	0.3110	0.3371	0.4288	0.3616	0.6515
BayesianRidge	MAE (R)	0.4751	0.6252	0.3791	0.4138	0.3162	0.2348	0.3282
	MAE (W)	0.4750	0.7885	0.3791	0.4137	0.3158	0.2330	0.3274
	MAE SD (R)	0.3735	0.4778	0.3071	0.3381	0.2789	0.1798	0.3868
	MAE SD (W)	0.3735	0.6026	0.3071	0.3380	0.2785	0.1784	0.3858
ARDRegression	MAE (R)	0.2814	0.2311	0.2941	0.1802	0.2896	0.2386	0.3257
	MAE (W)	0.2812	0.1938	0.2930	0.1828	0.2891	0.2408	0.3311
	MAE SD (R)	0.2212	0.1766	0.2382	0.1472	0.2555	0.1827	0.3839
	MAE SD (W)	0.2210	0.1481	0.2374	0.1494	0.2550	0.1844	0.3902

Table 8: Mean average errors by model, data preparation scenario, and fitted feature set.

In both Tables 7 and 8, the seven data preparation scenarios correspond to the train/test split methodologies described in Section V. The first five columns of both tables correspond to leaving out data collected on the five GPUs listed in Table 1, in the same order as given in the table. The models themselves are as described in Table 6. Accuracy scores in Table 7 are R^2 values for the predictions against the held-out test data for each scenario. Table 8 reports mean absolute errors for the predictions in each scenario, as outliers were deemed important to capture for purposes of the prediction exercise. To give a sense of the relative scale of the reported mean absolute errors, they are also computed in terms of the standard deviation of the target value for the training data in each scenario (listed as MAE SD in Table 8). In both tables, a flag of (R) next to a metric indicates results from the "raw" modeling data before manually winnowing, while (W) indicates the data as winnowed by examining significance of OLS coefficients.

Several interesting observations can be made from the results. Regarding the models themselves, the baseline linear regression model tends to underperform the other six. It was included specifically because in the author's experience, such relative performance is not a foregone conclusion, and a simpler model fit on more parsimonious data can sometimes achieve superior results when compared with throwing a complex machine learning model against a dataset, though this proved not to be the case here. The regularized models and least-angle regression tend to perform relatively similarly to one another, though L2 regularization appears to outperform this group in general. It is worth noting that this particular outperforming model (RidgeCV) was the one which proved most amenable to model tuning: a more aggressive set of regularization parameters increased performance here more so than for other models in this group. The Bayesian models tended to perform well in general, with the automatic relevance detection model (ARDRegression) performing "best" among the seven models overall: its accuracy was consistently high, its errors were consistently small, and it exhibited the most consistent stability across train/test split scenarios and raw vs. winnowed input data sets.

Additional observations of note can be made across the train/test split scenarios. As noted in the footnotes for Section V, for the first and fifth scenarios listed in Tables 7 and 8, the held-out hardware used for testing was relatively similar (having the same architecture and year of release), and as expected, this led to results across these two scenarios being relatively close to one another. Interestingly, holding out the data from the NVIDIA TITAN V which is three years newer than the "correlated" hardware just mentioned also led to fairly consistent results with these other two scenarios. Among the first five scenarios, holding out the data for the GeForce RTX 2080 Ti (the newest GPU among those tested) proved most difficult across all of the models, and holding out data from the GeForce GTX TITAN X led to the most consistently accurate results for all models save OLS. In general, these first five scenarios indicate that making cross-architecture performance predictions against new hardware may be a difficult exercise with inconsistent results, though fairly impressive accuracy against unseen devices can still be achieved as most strongly evidenced by the fourth scenario in the tables. Excluding one device per application at random in the sixth scenario led to less varied (and generally lower) model performance compared to other scenarios, as the models effectively blended results from the first five here. As expected, the seventh scenario using pure random sampling across all available data led to the best overall model performance; however, as described earlier, this is arguably the "least realistic" of all the scenarios, and the first five are most representative of real-world cases where an accurate performance prediction model would be practically useful (e.g., when determining new accelerator hardware to purchase).

Model stability was examined through three different lenses: cross-scenario accuracy, cross-scenario errors, and sensitivity to input dataset alterations. The most strongly consistent model by these measures was the Bayesian ARDRegression model, though the hand-tuned RidgeCV model also performed relatively consistently. It is surmised the additional flexibility of the ARDRegression model to independently model the distribution of each parameter coefficient was helpful during the fitting process, as even the scaled data exhibit different properties depending on the nature of the raw features. Linear regression performed worst by the three stability measures, with the remaining models mostly affected by the exclusion of results from the GeForce RTX 2080 Ti hardware. Regularized models were minimally affected by data winnowing (as regularization tended to remove a superset of the hand-winnowed features, though not in all cases), though again with the exception of effects from holding out the 2080 Ti data, all models were fairly consistent across the scenarios with regard to excluding the winnowed features or not.

While the results overall demonstrate the possibility of a well-chosen model performing prediction fairly accurately and consistently, various limitations of the modeling environment are worth noting. The most obvious of these is the limited size of the dataset: only 17 parallel applications were included, across only 5 hardware devices, all of which were from the same manufacturer spanning a 4-year release window.[9] While the applications were relatively diverse, the extensibility of results to broader application contexts and different classes of GPUs not available in the experimental environment is questionable. In addition, all tests were conducted against unmodified applications available in the Rodinia benchmark suite (as described in the footnotes to Section V). It is possible that the kernel code for these applications could be modified to take advantage of newer GPU features since the initial development of the code, which could affect the model performance (for better or worse).[10] It is interesting to note that thread count is not an explicit feature used in the models, though it is effectively an implicit feature: because the benchmark applications were presumably optimized for some GPU hardware that they were initially developed and tested on, thread count is essentially factored out of the model (one can imagine it being replaced with a dummy variable for "optimized thread count" which is set to 1 across all applications and thus provides no information to any model, for instance). Combining domain expertise with GPU computing expertise might allow this and other thread-related variables to become more important features in the models, which could lead to different model performance. However, at this juncture the question starts to cross the boundary from cross-architecture performance estimation to application-specific optimization, which is a separate (though also important) problem.

In light of these limitations and considerations related to the somewhat orthogonal question of application optimization, do the model results offer any real-world value? The answer likely depends on the context in which such a model may be used. In a situation where an entity wishes to find the best-performing combination of hardware and software for a particular application at any (reasonable) cost, such a model may be of limited use: acquiring and testing a variety of new hardware and hand-tuning kernels for each may be worth the investment, and using a model such as those presented in this paper to help guide an acquisition decision may be of limited or no use. However, not all situations might demand this level of performance, as time and capital to invest in high-performance applications are finite in practice. In a rough analogy to the appeal of parallel programming frameworks like OpenACC, which favor ease of deployment and use over maximal theoretical performance, the models presented in this paper may prove primarily useful as a guide to the approximate performance benefit of new hardware in cases where some model errors are tolerable as long as the results are directionally informative. In any case, the general modeling framework discussed here might be extended and refined through further research to offer greater practical benefit, even if the limitations of the experimental context here preclude the specifically fitted models from offering broad utility.

[9] Another item of note as relates to the hardware: the non-GPU hardware was not completely uniform across the tested environments, though all used Intel Xeon 64-bit CPUs with advertised clocks between 2.3 and 2.6 GHz. To the extent that the execution times were meaningfully affected by differences in this hardware, the model results may be called into question.

[10] Modification of the kernels might also lead to generally lengthier application execution times, which could potentially make error measurements more informative: while mean absolute errors are reported here, execution times for the applications are all on a fairly small scale, and one application in particular represents times that are something of an outlier. Thus, average errors across all applications may be of limited use, which is why accuracy metrics and also MAEs in standard deviation terms are included in the results.

VII. Conclusions

The modeling framework described in this paper offers the potential for (necessarily limited) real-world use in contexts where maximal performance optimization is not required. The framework itself consisting of a mixture of hardware and software features for cross-architecture performance prediction may be extended in future research across additional applications and more modern (and varied) GPUs, and may use a modified set of features to improve accuracy and reduce errors when predicting execution times on unseen hardware. Such extensions of the framework may be necessary to make use of results, due to limitations in the modeling environment used for the experiments described in this paper. However, these limitations notwithstanding, by comparing several modeling approaches against a baseline and testing for stability in multiple dimensions, this research has demonstrated the capability to fit a performance prediction model with reasonably stable results across a wide range of parallel application contexts and a moderate range of hardware.

References

[1] Q. Wang, C. Liu, and X. Chu, "GPGPU Performance Estimation for Frequency Scaling Using Cross-Benchmarking," in *GPGPU*, 2020, pp. 31-40.

[2] G. Alavani, K. Varma, and S. Sarkar, "Predicting execution time of CUDA kernel using static analysis," in *IEEE International Conference on Parallel & Distributed Processing with Applications, Ubiquitous Computing & Communications, Big Data & Cloud Computing, Social Computing & Networking, Sustainable Computing & Communications*, 2018, pp. 948-955.

[3] U. Gupta, J. Campbell, U. Ogras, R. Ayoub, M. Kishinevsky, F. Paterna, and S. Gumussoy, "Adaptive performance prediction for integrated GPUs," in *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2016, pp. 1-8.

[4] Y. Zamora, B. Lusch, V. Vishwanath, I. Foster, and H. Hoffman, "Cross Architecture Performance Prediction," 2020. Available: https://newtraell.cs.uchicago.edu/files/tr_authentic/TR-2020-12.pdf.

[5] G. Chapuis, S. Eidenbenz, and N. Santhi, "GPU Performance Prediction Through Parallel Discrete Event Simulation and Common Sense," in *VALUETOOLS'15: Proceedings of the 9th EAI International Conference on Performance Evaluation Methodologies and Tools*, 2016, pp. 204-211.

[6] N. Ardalani, U. Thakker, A. Albarghouthi, and K. Sankaralingam, "A Static Analysis-based Cross-Architecture Performance Prediction Using Machine Learning," in the 2nd International Workshop on AI-Assisted Design for Architecture (AIDArc), held in conjunction with the International Symposium on Computer Architecture (ISCA), 2019.

[7] N. Ardalani, C. Lestourgeon, K. Sankaralingam, and X. Zhu, "Cross-Architecture Performance Prediction (XAPP) Using CPU Code to Predict GPU Performance," in *MICRO-48: Proceedings of the 48th International Symposium on Microarchitecture*, 2015, pp. 725-737.

[8] N. Agarwal, T. Jain, and M. Zahran, "Performance Prediction for Multi-threaded Applications," in the 2nd International Workshop on AI-Assisted Design for Architecture (AIDArc), held in conjunction with the International Symposium on Computer Architecture (ISCA), 2019.

[9] K. O'Neal, P. Brisk, E. Shriver, and M. Kishinevsky, "Hardware-Assisted Cross-Generation Prediction of GPUs Under Design," in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2019, pp. 1133-1146.

[10] M. Sadeghi, "Improving the GPU Performance Prediction Models to Design Space Exploration," in *International Journal of Science and Engineering Investigations*, 2016, pp. 24-31.

[11] Rodinia: Accelerating Compute-Intensive Applications with Accelerators, http://rodinia.cs.virginia.edu/doku.php.

[12] S. Memeti, L. Li, S. Pllana, J. Kołodziej, and C. Kessler, "Benchmarking OpenCL, OpenACC, OpenMP, and CUDA: Programming Productivity, Performance, and Energy Consumption," in *ARMS-CC '17: Proceedings of the 2017 Workshop on Adaptive Resource Management and Scheduling for Cloud Computing*, 2017, pp. 1-6.

[13] NVIDIA Profiler User's Guide: nvprof, https://docs.nvidia.com/cuda/profiler-users-guide/index.html#nvprof-overview.

[14] Lines of Code (linecounter) tool. Available: https://marketplace.visualstudio.com/items?itemName=lyzerk.linecounter.

[15] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, É. Duchesnay, "Scikit-learn: Machine Learning in Python," in *JMLR 12*, 2011, pp. 2825-2830.