UNIVERSITY OF CALIFORNIA
RIVERSIDE

# Performance and Power Prediction of Compute Accelerators Using Machine Learning

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Kenneth Norman Lee O'Neal

June 2018

Dissertation Committee:

- Dr. Philip Brisk, Chairperson
- Dr. Jiasi Chen
- Dr. Rajiv Gupta
- Dr. Daniel Wong

The Dissertation of Kenneth Norman Lee O'Neal is approved:

_____

_____

_____

_____

Committee Chairperson

University of California, Riverside

# ACKNOWLEDGEMENTS

I would like to thank my graduate advisor Philip Brisk for recognizing my potential as an undergraduate and taking what would normally be considered a high-risk student in under his wing. Further, he has been a true mentor to me, helping me to critically evaluate, understand and integrate cutting edge approaches into my daily tasks and research efforts, even when those research efforts had no clear alignment with his own interests. He has been instrumental in facilitating my academic and professional career. I would also like to thank him for his financial support via his many NSF grants and awards. I would also like to thank my committee members Dr. Jiasi Chen, Dr. Rajiv Gupta, and Dr. Daniel Wong for their patience, guidance and evaluation of my Ph.D. work.

Further, I would also like to thank the many co-workers and friends I've met and worked with throughout my internships, who gave me opportunities and trusted me with ambitious projects. These opportunities, their advice and training, and their professionalism have had profound impacts on my research interests and helped to mature my approach to performing high-quality research in a competitive and demanding field. These opportunities have been instrumental in shaping my approach to research, my desire to continue performing research, and set me on a career path that has the potential to be truly life changing. I would also like to specifically thank Strategical CAD Labs (SCL) of Intel, namely Michael Kishinevsky and Emily Shriver for their early financial support and for exposure to research fields that have inspired me greatly, as well as my first manager Gary Snyder who was often my greatest supporter and a devoted friend in the work place. Finally, I would also like to thank the reviewers, conferences and journals that acknowledged and accepted for publication the research that this dissertation is based on. Lastly, I would like to thank my parents and grandparents who've supported and encouraged me in all ways. Thank you for always standing behind me and helping push me up the proverbial hill.

Riverside, CA May 2018

# DEDICATION

I would like to dedicate this dissertation to my loving mother, Laura. Although you are unable to witness me completing my Ph.D. and attend my commencement, I know you will be watching on in spirit, and that you had the greatest confidence that this day would come. Thank you for being my fiercest champion, my caretaker, and for convincing me that I always had more to offer the world. Without a doubt your encouragement, your relentless belief in me, and your continual reinforcement of the value of education during my early years of development, none of this would have been possible.

ABSTRACT OF THE DISSERTATION

Performance and Power Prediction of Compute Accelerators Using Machine Learning

by

Kenneth Norman Lee O'Neal

Doctor of Philosophy, Bourns College of Engineering in Computer Science
University of California, Riverside, June 2018
Dr. Philip Brisk, Chairperson

CPUs and dedicated accelerators (namely GPUs and FPGAs) continue to grow increasingly large and complex to support todays demanding power and performance requirements. Designers are tasked with evaluating the performance and power of or increasingly large design spaces during pre-silicon design. Validating and evaluating the performance during the pre-silicon stage catches performance and device issues in advance of fabrication. This reduces time-to-market by reducing the bugs that must be found and fixed after manufacturing, after synthesizing an FPGA accelerator using HLS tools.

CASs are integral to architectural design and are often the only tool at computer architects' disposal to evaluate workloads on architectural design points in advance of manufacturing. To enable optimization of the architecture without the exorbitant cost of repeatedly fabricating designs, a high-level of simulator precision is required. As such, the simulators are compute intensive, limiting the number of simulations, and thus workloads or design points that can be evaluated before manufacturing. Historically the computational cost of simulation is absorbed, which slows down time to market and increases the cost of development.

Machine Learning and statistical prediction models have emerged as viable tools to avoid repeated cycle accurate simulations by accurately predicting the metrics generated by CASs. After one-time model training, the predictive models can then be used in lieu of simulation. Here I will present my research into the development of machine learning frameworks for GPU and FPGA architectures, which leverage cross-architecture predictive statistical modeling to significantly reduce the time required to evaluate workloads and architectural design points.

# CONTENTS

# LIST OF FIGURES

# LIST OF TABLES

# LIST OF EQUATIONS

# Chapter 1 Introduction

As computer architecture grows increasingly complex, Cycle-Accurarte Simulators (CAS) are rapidly becoming a bottleneck when designing and testing Central Processing Units (CPUs) and Graphical Processing Units (GPUs). CPUs and GPUs must be designed and tested before manufacturing, however in both academic and industrial CASs, the necessary run times are untenable when attempting to evaluate numerous architecture design points, or workloads for a chosen design point. CPU and GPU CAS [1] are used to estimate performance and power consumption, to perform *design space exploration* (DSE), and to prototype and functionally verify new architectures. CPU and GPU simulators also support co-optimization of hardware and software (e.g., application programming interface -- API, firmware, and driver development). The industrial counterparts of academic simulators must also perform *Register Transfer Level* (RTL) performance validation, necessitating greater detail and higher accuracy, exacerbating the problem. In both cases, designers must meet *quality of service* (QoS) requirements which dictate the requisite performance and power consumption of the design under test. Typical software simulators execute orders of magnitude slower than native execution on commercial hardware [2], too slow to meet modern design productivity demands, despite efforts to raise the abstraction level [3], parallelize the simulations [4], and leverage hardware assistance [5] to improve simulation speed.

The design and implementation of *Field Programmable Gate Array* (FPGA) accelerators is difficult and time consuming, in no small part due to the complexity and tedium inherent to RTL design. In addition to the difficulty of RTL design, the synthesis tools themselves have long run times that are prohibitive for design tasks. *High-Level Synthesis (HLS)* tools [6, 7], can significantly boost developer productivity compared to writing RTL. Software simulators are used by commercial FPGA synthesis tools to obtain early estimates of design performance. Although FPGA accelerators aren’t manufactured, design efforts are impeded by long simulation and synthesis

times. The developer must appropriately set design pragmas (pipelining, array partitioning, loop unrolling), to optimize performance and throughput. Finding the right HLS parameter settings is a complex problem, which entails some mechanism to evaluate the most promising design points that have been uncovered. Although synthesizing, placing, and routing each design point to measure the requisite metrics (cycle count, power usage, and utilization) to characterize the design points performance by direct execution would be ideal, the amount of time required is infeasible.

Predictive modeling, and specifically cross-architecture predictive modeling has emerged as one potential solution to this conundrum. Although less precise than cycle-accurate simulation, predictive models can be trained, evaluated and applied faster, thereby increasing both the number of design points that can be explored and the number of workloads evaluated per design point. This shares some similarities to the introduction of statistical sampling into CASs [8, 9] where simulation taime is reduced by identifying program phases and choosing phase representatives to eliminate redundant simulation. in both cases, the productivity benefits that accrue from modifying existing methodologies outweigh the resulting loss in accuracy. The goal of these predictive models is to obtain performance estimates more rapidly than with CAS by extending fast executing host platforms with predictive models, as show in Fig. 1:1.

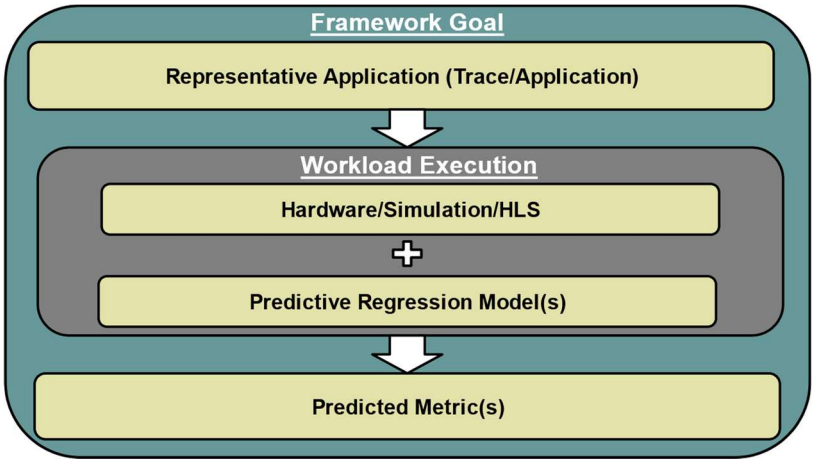


Figure 1:1 Cross-Architecture Predictive Modeling Goal

## 1.1 Cross-Architecture Predictive Modeling

Generally, cross-architecture predictive modeling is the task of using one architecture, represented as either a post-silicon commercially available device or as a simulator configured to represent a device, as a platform to execute representative workloads and collect execution statistics. These statistics will be used as features to train models that predict the metrics typically output by the CAS and used to evaluate the design. Cross-architecture models, which exist for CPUs, GPUs, and FPGAs can be further categorized into 3 groups.

- **Cross-generation models**: direct execution of workloads on an older generation of the target design provides features used to predict the performance and/or power consumption of the newer target within the same family, possibly at the pre-silicon stage of development, where only a CAS is available as a performance validation mechanism.
- **Cross-abstraction models**: an abstracted, functional simulator host is configured to represent the target design and used to provide features used to predict the performance and/or power of the detailed CAS representation of the target design.
- **Cross-platform models**: features obtained from executing workloads on a host architecture that significantly differs from the target design (e.g. a CPU) are used to predict the performance of the target design (e.g. , a FPGA [10] or GPU [11, 12] accelerator).

In the remainder of this manuscript we present three frameworks, corresponding to these three categories of cross-architecture modeling. We present the general form of the predictive modeling framework below. Each framework will be similar, but the host and target platforms, the workloads used to evaluate the approach, the feature collection techniques used, and the employed ensemble of models trained and applied, will vary. We present the general form of the prediction framework in Figure 1:2. We also present the categories of machine learning models leveraged by each framework to train an ensemble of models below in Figure 1:3. A variant of each figure will

be re-presented in the following chapters, with the specifics of each framework detailed. Models are from the following categories: Linear, regularized, and non-linear regression. The following section presents the model training, evaluation and application methods used in all our frameworks.

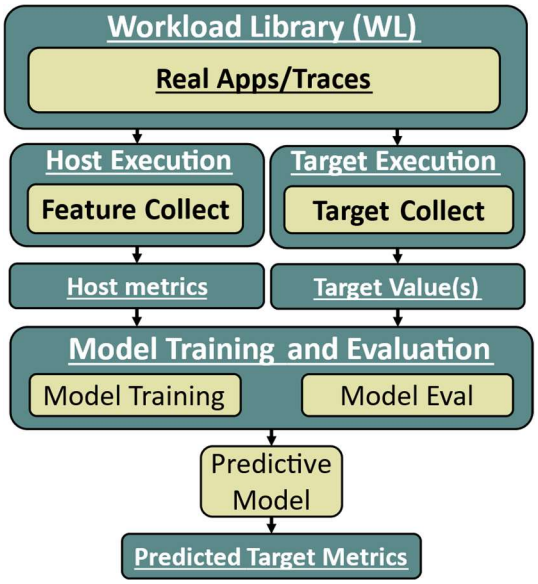


Figure 1:2 The Generalized Framework Form

The general framework, where a host is used to collect metrics, a target used to provide example power/performance values, and models trained and validated to perform the prediction.

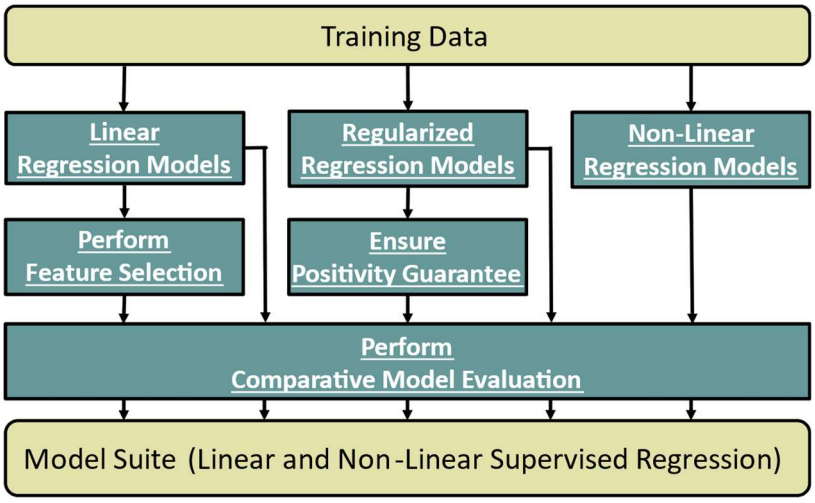


Figure 1:3 The General Model Ensemble Form

This depicts the categories of models used in each iteration of our cross-architecture predictive modelling framework. We construct linear, regularized and non-linear regression models employing feature selection and non-negative guarantees to improve model accuracy and generalizability.

## 1.2 Training, Validation and Application of Predictive Models

In all instances, we are predicting one or more metrics related to performance and power of architectural designs under test. The primary problem we are tasked with is learning from data by constructing a model, known as model training, which in the case of supervised regression, has learned by example. Training supervised regression models requires several key ingredients:

- A set of representative workloads (data points).
- A set of features, which can be reasonably collected from the representative data points and contain meaningful information detailing the data points' behavior or attributes.
- A corresponding target metric or set of target metrics that you wish to predict. You should have at least one target metric for each data point.

For a model to be meaningful and work well on data not present in your training set, the features collected should correlate well with the value attempting to be predicted. The collection of all three ingredients constitutes what is commonly referred to as a training set. This training set will be used by a set of statistical or machine learning techniques to learn relationships between the features and target values for each data point to form a model. After assessing model accuracy, the model cannot yet be used on additional data points, it must first be verified to work well in general, on data not contained in the representative training set. This process is known as model validation.

Model validation can be performed in a variety of ways depending on the amount of available data points, and target values for those data points available for model training. The primary purpose of model validation is to verify that the model can be applied accurately to data points not in your training sample. This is referred to as model generalizability. The average error of the model when applied to the validation set is typically coined out-of-sample error ($E_{out}$). We will use these terms extensively throughout the remaining chapters.

In a data rich scenario, validation is performed by creating a model validation set. This validation data set consists of the same three primary ingredients as the model training data set, but the data points are not those contained in the training set. This is typically selected by choosing at random a percentage of data from all available data to withhold from model training. Although all three ingredients are still required, the model validation step does not alter or perform additional learning. Instead, the model is simply applied, and the collected target values are compared to the model predictions to ensure the model is accurate.

In a data sparse scenario, wherein too few available data points are available to create both a sufficiently representative training set and a holdout validation set, statistical methods are instead applied to accurately estimate how the model would perform on unseen data. One of the most popular and effective techniques employed is cross-validation (CV). Due to the relatively small amount of target metrics and data points available for architectural modeling (due to slow simulation times) we exclusively apply CV to evaluate our models error on unseen data. Specifically, we perform 10-fold CV which has been shown to produce accurate estimates [13] of $E_{out}$. After properly evaluating the performance of the model during model validation, the model can then be applied.

Model application is where the benefit of model training and validation is realized. The model can be used to provide estimates of the target for which it was trained with very little overhead. In contrast to model training and validation, only two ingredients are now required: 1) Additional workloads or data points that need to be evaluated and 2) features for those data points. Example target metrics are no longer needed, as the model has already been verified, and the model can be used in lieu of the target device that would normally generate those values, i.e. a CAS. Model application can provide large speedups with very little impact on accuracy as we will demonstrate. Figure 1:4

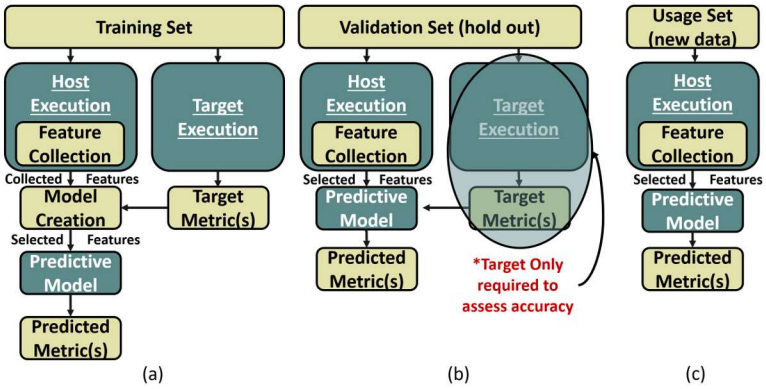


Figure 1:4 Generalized Model Training, Validation and Application Flow

This depicts the general form of the (a) model training, (b) model validation and (c) model application flow for each model training in our framework iterations. Each model in the ensemble follows a similar procedure. In our case our validation sets are created via 10-fold cross validation procedure.

While the preceding paragraphs present the generic form of model training, validation and application little is presented in the way of how the models are formed. In lieu of defining and presenting the relevant equations for model construction and model validation in the introduction, we define and present the models where first used, and the refer to them accordingly in the remainder of the manuscript.

The remainder of the manuscript is organized as follows: In chapter 2, we present our cross-generation modeling approach for integrated GPUs executing 3D rendering DirectX workloads to predict GPU performance. In Chapter 3 we present our cross-abstraction modeling approach, which also predicts the performance of integrated GPUs executing DirectX workloads. In Chapter 4, we present our cross-platform modeling approach which leverages CPU execution to predict the performance and power of several FPGA accelerators designed using HLS. Chapter 5 presents a consolidated related works, largely based on our previously published cross-architecture predictive modeling survey [14], omitting related works in each section. Chapter 6 concludes.

# Chapter 2 Cross-Generation GPU Prediction

CAS times for highly-threaded processors, such as GPUs, are rapidly becoming untenable. The situation is exacerbated in industry, where simulators serve a dual-purpose of performance simulation and RTL performance validation. Cross-architecture predictive modeling and specifically hardware-assisted cross-generation predictive statistical modeling can help to overcome this conundrum when designing subsequent GPU devices in a larger family. What is needed is a commercially available GPU, representing a current- or past-generation member of the family, a simulator representing a future-generation GPU family member under development, and a set of representative rendering workloads.

This chapter is based on a journal article "Hardware-Assisted Cross-Generation Prediction of GPUs Under Design" [15] which extends "HALWPE: Hardware-Assisted Light Weight Performance Estimation" [16], a methodology that uses fabricated silicon (host) GPUs to predict the performance of future GPUs under development (target). Our experiments focus on GPUs and treat the 7.5$^{th}$ generation integrated HD4600 GPU of the Intel Core i7-4790 processor as a current-generation GPU (fabricated silicon) and predict the performance of three future-generation GPUs: two 8$^{th}$ generation Broadwell GPUs and one 9$^{th}$ generation Skylake GPU. The prediction is performed by configuring a CAS to model the newer generation GPUs, executing a set of 3D render workloads on both the host GPU, to collect performance counter and software metrics, and the CAS to collect target performance, cycles-per-frame— CPF, simultaneously. We then train an ensemble of regression models to predict the CPF of each frame using the host metrics; the models are then applied to new workloads. The goal is to speedup pre-silicon development tasks like functional and performance verification of prototype architectures, perform hardware/software co-optimization and more. Fig. 2:1 depicts the goal of our framework, and Fig. 2:2 depicts an example pre-silicon hardware/software co-optimization task. HALWPE has several contributions:

- HALWPE's novelty is accurately predicting GPU performance across three device generations, spanning micro-architectural, software, parallelism and process improvements. HALWPE achieves 7.45 %, 7.47 % and 8.91 % average *out-of-sample-error* respectively.
- HALWPE uses hardware assistance to run ~30,000-45,000x faster than a GPU CAS.
- HALWPE predicts performance impacts from changes to vendor-provided drivers and APIs (Fig. 2:2) on current and future generations of GPU.
- HALWPE predicts performance impacts caused by increasing available GPU parallelism on the current and future generations of GPU.
- HALWPE ranks features to improve model inference and guide designers toward prime microarchitectural and software candidates that warrant additional study.

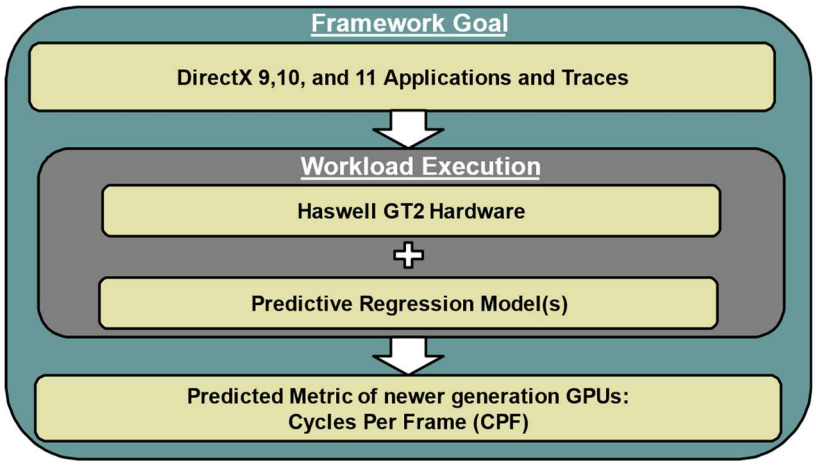


Figure 2:1 HALWPE Model Goal

Modeling framework: a host Haswell GT2 GPU executes 3D DirectX9, 10, and 11 workloads. Performance counter measurements obtained from the host GPU are used by predictive models to predict the performance of the GPU CAS configured as newer GPU generation devices.

The remaining text is organized as follows: Section 2 details the modeling framework, model building and application, and workload execution. Section 3 describes the Intel GPU generations modeled and their relative differences. Section 4 details the regression model ensemble

employed by HALWPE. Section 5 details simulation-based model results when modifying the driver and increasing available device parallelism. Section 6 details hardware-assisted, cross-generational model results. Section 7 presents feature ranking and discussion, and finally Sections 8 and 9 present related works and our concluding remarks, respectively.

## 2.1 HALWPE Modeling Framework

We assume that at least one current-generation GPU is available in silicon, and that a high-accuracy next-generation GPU simulator is available, along with representative workloads. Fig. 2:3 illustrates the HALWPE model development flow. The *Graphics Workload Library (GWL)* refers to our collection of benchmarks, listed in Table 2:1.

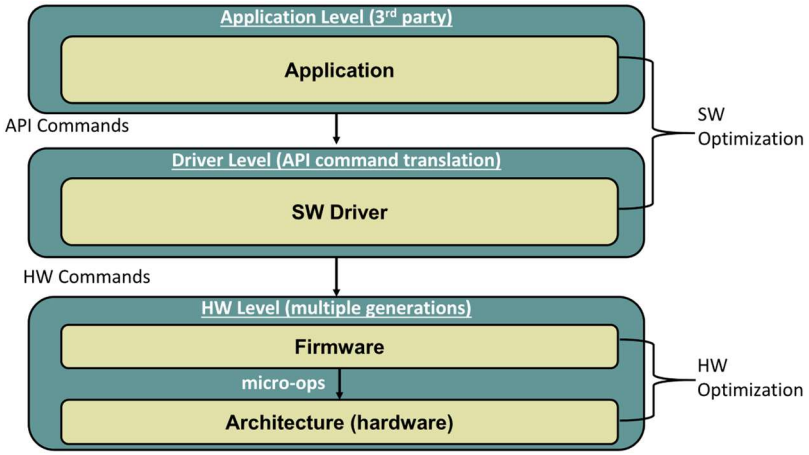


Figure 2:2 HW/SW Co-Optimization
GPU performance depends on the application, driver/API commands, and architecture.

The GWL contains rendering frames from 43 *DirectX* games and GPU benchmarking tools spanning the version 9, 10, and 11 APIs. We collected multiple frames per application and treat each as one workload. The one-frame-per-workload constraint is imposed by the GPU simulator's execution overhead, but longer traces can be executed as well. The GPU Simulator models the GPU microarchitecture, memory subsystems, and a representation of *Dynamic Random-Access Memory* (DRAM), all validated internally

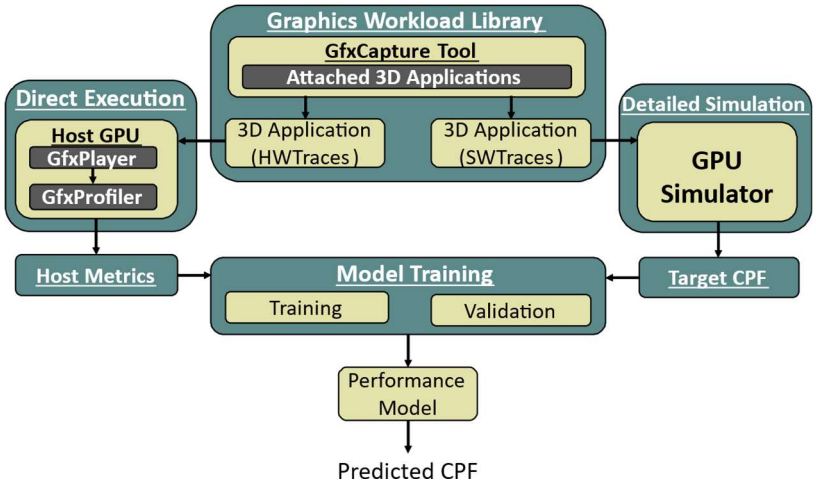


Figure 2:3 HALWPE Model Development Flow

(1) Traces are collected and stored in the GWL. (2) Workloads execute on the current-generation GPU host, and next-generation simulator. (3) Performance counter measurements and DirectX program metrics obtained from the host are used to train a model to predict the CPF that the GPU simulator would report.

when configured to model post-silicon GPUs. No explicit memory model beyond the memory subsystem and DRAM already present on the integrated GPU, leading to few related counters. The GWL applications are assembled into a single training set; we apply 10-fold cross validation to estimate out of sample error. We use three proprietary trace tools to collect single-frame traces in two formats (*GfxCapture*), replay isolated traces (*GfxPlayer*), and collect performance counters (*GfxProfiler*) [17], hardware queries [18], and DirectX program metrics [4–11] on the Haswell host GPU.

The trace formats are *HWTraces* (DirectX commands) executed on our Haswell host GPU [17], and *SWTraces* (native GPU commands) executed on our GPU simulator. To reduce profiling overhead, we collect performance counters that can be read in one pass as detailed in the table on page 13 of Ref. 1. Table 2:2 summarizes the software tools required to implement the HALWPE framework.

| Workload | Frame Count | DX Version |
|---|---|---|
| Diablo 3 | 2 | 9 |
| Dragon Age: Origins | 11 | 9 |
| Dragon Age 2 | 10 | 9 |
| Left For Dead 2 | 10 | 9 |
| Portal 2 | 12 | 9 |
| Skyrim | 9 | 9 |
| Starcraft 2 | 1 | 9 |
| StarWars The Old Republic | 10 | 9 |
| Witcher2 | 10 | 9 |
| 3D Mark Vantage | 30 | 10 |
| Hawx | 5 | 10 |
| Cut The Rope | 3 | 10 |
| Fishie | 2 | 10 |
| Resident Evil 5 | 10 | 10 |
| Winsat Alpha Blend | 3 | 10 |
| Winsat ALU Perf | 3 | 10 |
| Winsat Batch Perf | 3 | 10 |
| Winsat Constant Buffer | 3 | 10 |
| Winsat Geometry Perf1 | 3 | 10 |
| Winsat Geometry Perf2 | 3 | 10 |
| Winsat Texture Load Perf | 3 | 10 |
| 3DMark 11 entry | 19 | 11 |
| 3DMark 11 perf | 20 | 11 |
| 3DMark Cloud Gate | 9 | 11 |
| 3DMark Ice Storm | 9 | 11 |
| Assassins Creed 3 | 5 | 11 |
| AVP: Evolution | 7 | 11 |
| Batman Arkham City | 8 | 11 |
| Battlefield 3 | 5 | 11 |
| Bioshock Infinite | 7 | 11 |
| Crisis 3 | 10 | 11 |
| Dirt2 | 10 | 11 |
| F1 2012 | 15 | 11 |
| Farcry 3 | 10 | 11 |
| Heaven | 10 | 11 |
| Lost Planet 2 | 26 | 11 |
| MaxPayne 3 | 5 | 11 |
| Metro Last Light | 10 | 11 |
| Saints Row 3 | 8 | 11 |
| Saints Row 4 | 7 | 11 |
| Sleeping Dogs | 6 | 11 |
| Stone Giant | 11 | 11 |
| Tomb Raider | 1 | 11 |

Table 2:1 Graphics Workload Library

GWL: 26 DirectX Applications consisting of 364 rendered frames

The predictive models are programmed using R and other commercially available tools to estimate GPU CPF. Power per frame and other metrics can also be predicted given the target simulator provides a reference value, though feature rankings and selection may change. Model training time is not included in the runtime comparison of HALWPE to cycle-accurate simulation, as training time is amortized over repeated model usage.

| Tool | Acronym | Purpose |
|---|---|---|
| Graphics Workload Library | GWL | Collect frame traces |
| GPU Simulator | - | Collect golden reference CPF |
| Graphics Capture Tool | GfxCapture | GPU frame trace capture tool |
| GPU Hardware Trace | HWTrace | Capture traces executed on the host GPU |
| GPU Simulator Trace | SWTrace | Capture traces executed on the GPU simulator |
| HWTrace Replay Tool | GfxPlayer | Replay HWTraces on the host GPU |
| HWTrace Profiling Tool | GfxProfiler | Collect model features from execution on the host GPU |

Table 2:2 HALWPE's Software Tool and Libraries

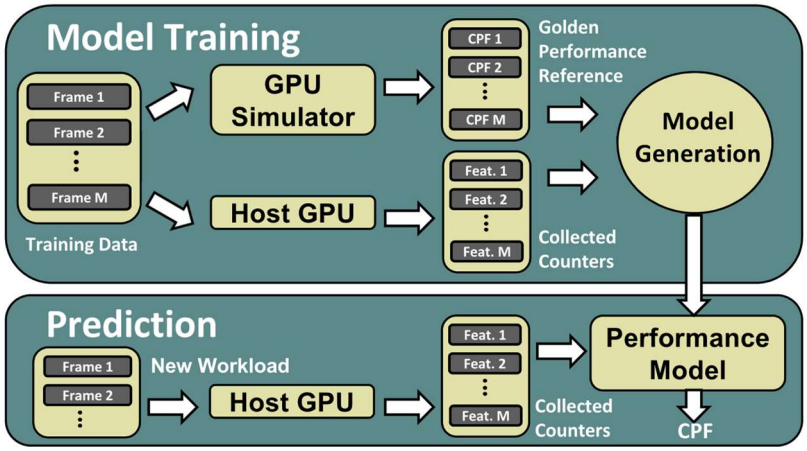


Figure 2:4 HALWPE Model Training and Application

Training (Top): Performance counter measurements and DirectX program metrics obtained from direct execution on the GPU host are used to train a model to predict the performance of a GPU simulator configuration. Prediction (Bottom): An application runs on the GPU host; the collected performance counter measurements and DirectX program metrics are input to the performance model, which predicts the CPF that the GPU simulator would report.

In practice, models are trained on one set of workloads, and deployed on a disjoint second set of workloads. Once a model has been trained it can be applied to any 3D rendering workload of any length. However, validation of that prediction is limited to workload lengths that can be reasonably executed on the CAS. Fig. 2:4 illustrates the model training and deployment (prediction) workflows.

## 2.2 Intel GPU Architecture Description

HALPWE is validated using three generations of Intel integrated GPUs (see Table 2:3). The host desktop *personal computer* (PC) has a 4-core, 8-thread Intel Core i7-4790k, 16 GB of DDR3 @ 1666 MHz, an Intel HD 4600 Haswell GT2 GPU running at 1155 MHz, and a 2TB 7200 RPM hard disk. The Broadwell GT2, Broadwell GT3 and Skylake GT3 are later versions of this GPU for which simulators are available. We include the performance differences between each generation of GPU to highlight that HALWPE's model suite can accurately generate cross-generation performance estimates, even when the relative performance difference of the two generations is large. In principle this method can be used on any GPU supporting the DirectX API to produce the same metrics, and the devices hardware counters. Further, GPUs leveraging other APIs, such as OpenGL can produce a set of metrics like the DirectX.

To create hardware-assisted model scenarios, we use simulator configurations that execute a driver reflective of the GPU generation: version 1 (Haswell GT2), version 2 (Broadwell GT2), and version 3 (Broadwell GT3, which we also use for Skylake GT3). In some situations, compatibility issues between the architecture and driver caused trace execution to fail on the GPU host and simulator. In Table 2:3, the Haswell GPU host can execute 300 of the available traces, while simulators for the Broadwell GT2, GT3 and Skylake GT3 can execute 282, 364, and 364 traces respectively.

While Skylake and Broadwell GPUs are commercially available, we validate accuracy using only CAS. Our goal is to mimic the GPU design process while employing commercially to maximal data can be disclosed publicly, e.g. model features. This ensure validated simulator configurations exist, while avoiding implementation work to target in-flight designs. For any host-target prediction scenario, the number of traces that we use to build and evaluate the model is the minimum number that both host and target have the capability to execute.

| GPU Product | Available As | # Slices | # EUs | Driver Gen. | # Executable Traces |
|---|---|---|---|---|---|
| Haswell GT2 | Hardware | 1 | 20 | 1 | 300 |
| Broadwell GT2 | Simulator | 1 | 24 | 2 | 282 |
| Broadwell GT3 | Simulator | 2 | 48 | 3 | 364 |
| Skylake GT3 | Simulator | 2 | 48 | 3 | 364 |

Table 2:3 HALWPE's Evaluated Intel GPU Device Legend

### 2.2.1 Integrated GPU Architecture Differences

HALWPE's novelty is accurately predicting GPU performance across multiple device generations spanning micro-architectural, software, parallelism and process improvements. Intel's GPU render pipeline is organized into two logical groups: (1) the Unslice and (2) the Slice. The *Slice* count of a GPU is a measure of available parallelism and contains three sub-elements. The slice common holds fixed function (FF) caches, global slice units, the sub-slice, and L3 cache. Sub-slices are organized into parallel groups each containing *Execution Unit (EU)* clusters and their supporting thread dispatch (TD) units, samplers, instruction cache (IC), and peripherals. Section 7 provides relative performance comparisons between the host and target GPU representations. [12–14].

We utilize three generations of Intel integrated GPU device; a Haswell GT2 single-slice host (previous generation hardware), with 20 EUs per slice, two variants of Broadwell (single slice GT2, and dual-slice GT3), with 24 EUs per slice and a dual-slice Skylake GT3 containing 24 EUs per slice. The following section omits detailing individual units and their purpose, instead focusing

only on the differences between generations. Readers interested in a more detailed description of the architectures should consult [12–14].

#### 2.2.1.1 Haswell vs. Broadwell Architecture

Broadwell generation GPUs optimize the microarchitecture. Below we highlight key areas where the Broadwell device has improved over the Haswell implementation.

**Unslice:** The CPU and GPU communication unit, the Graphics Translation Interface (GTI) to lower level cache (LLC) bandwidth has improved, allowing 64-bit read and write rather than 32-bit as in Haswell. The FF render pipeline has also been optimized on a per-unit basis, resulting in improved pixel back end fill rate and improved Z/Hi-Z test performance.

**Slice:** Most notably, Broadwell has doubled 32-bit integer computational throughput, and has added 16-bit floating point support. An increase in computational throughput derives from more efficient global resource sharing (LLC) amongst slices, changing the total number of EUs per slice and changing the number of sub-slices.

**Slice Common:** The L1 cache has an increased in overall size by increasing allocation per slice, and the L3 cache has increased 33.33 % from 385 Kbytes to 576 Kbytes.

**Sub-slice:** By increasing the number of sub-slices to 3 and allocating 8 EUs per sub-slice in Broadwell rather than maintaining two subs-slices with 10 EUs each, two sources of additional throughput were added. First, the total number of EUs increased, and sampler contention has decreased. In total Broadwell contains 120 % more EUs, and 150 % more sampler throughput than a Haswell counterpart containing the same number of slices.

Comparing the performance of the Haswell and Broadwell generation devices we measure a median performance difference of 66.43 %, an average of 317.74 % and a maximum improvement >1000 %. The comparisons are between an actual GPU, the host Haswell generation device, and a near CAS, our Broadwell generation target, the same GPUs used in Scenarios$_{4\text{-}5}$.

#### 2.2.1.2 Broadwell vs. Skylake Architecture

The recently released Intel Skylake generation GPUs make significant architectural improvements to the Broadwell GPU architecture.

**Unslice:** At the platform level, the GTI latency has been reduced, and the command issue ring buffer has also been improved. Further, DDR speed has increased from 1868 MT/s to 2133 MT/s. In total platform compute has improved 50 % from 768 GFLOPS to 1152 GFLOPS. Notable improvements have also been made to the Fixed Function render pipeline, including to the Vertex Shader (VS), the Geometry Shader (GS), the Hull Shader (HS), and the DS. Additional geometry features have also been added such as Auto Strip detection and their employment in the Tessellation stages of the FF pipeline. This serves to improve both bandwidth and cull rates by reducing the number of redundant computations performed.

**Slice:** pixel back end fill rate has been further increased between 33 % and 100 %, workload dependent. In addition, a new Multi-Sampling anti-aliasing (MSAA) mode has been added, allowing for 16x MSAA, and performance improvements in the existing 2, 4 and 8x MSAA modes.

**Slice Common:** Cache has been increased to 768 Kbytes, an additional 25 %, a total of 200 % over the Broadwell size cache. In addition to the increased memory, memory management is optimized by performing render target compression, compressing memory before send to increase bandwidth at each cache line. This results in 11 % increased cache line bandwidth.

**Sub-slice:** has been improved by adding explicit 16-bit and 32-bit floating point support. EU/Sampler throughput and Z/Stencil and Pixel operation speed has increased 200 % by performing individual pixel hashes on different slices. Shared virtual memory and cache coherency have also been improved, resulting in better 3D computation. Atomic operations for 32-bit floats, min, max, compare and exchange have also been added. TD has been further improved by allowing

smaller thread groups, providing finer granularity pre-emption to increase 3D compute responsiveness. These architectural changes result in an additional increase of 24 % performance on average, and 14.3 % median when comparing the Broadwell and Skylake generations.

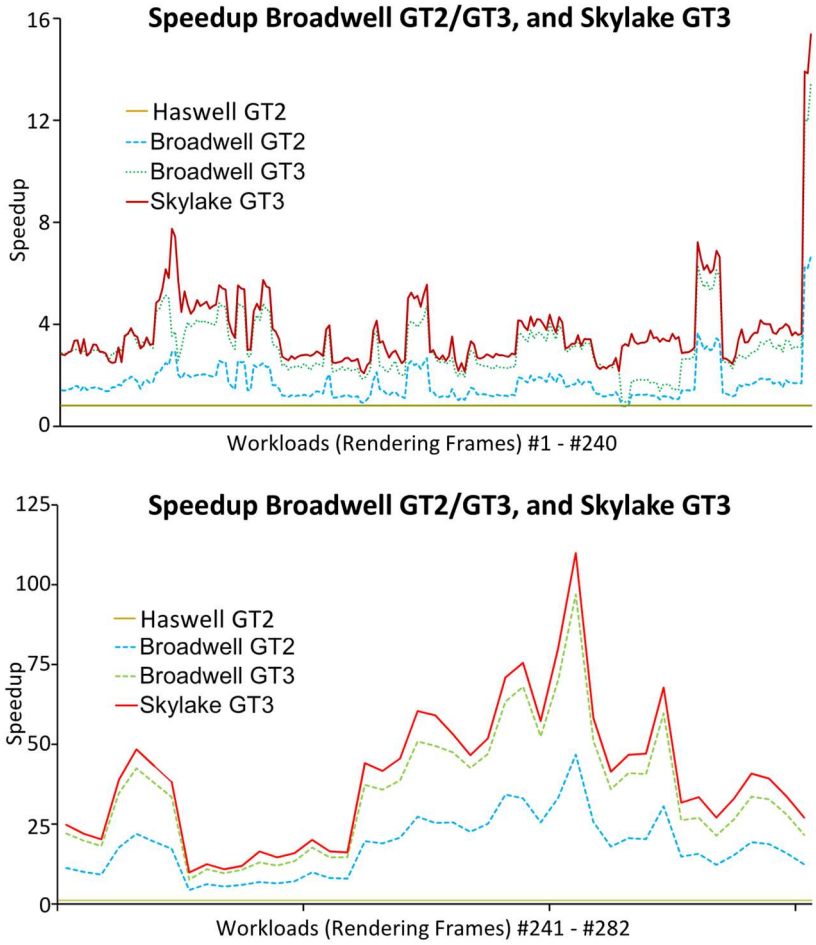


Figure 2:5 HALWPE Speedup between Broadwell and Skylake

The speedup computed from CPF obtained on the Broadwell GT2/GT3 and Skylake GT3 (normalized to the CPF attained by Haswell GT2) for 282 rendering frames. Frames 1-240 (top) and frames 241-282 (bottom) are shown on two separate graphs due to the stark difference in CPF ranges.

Fig. 2:5 reports the speedup of the three target GPU architectures we predict (Broadwell GT2/GT3, and Skylake GT3) normalized to the CPF attained by the Haswell GT2 host for 280 frames. The CPF difference between Skylake GT3 and Haswell GT2 varies from 3x to 112x. This

large variation in CPF speedup (and at times a decrease in the Broadwell GT2/GT3 cases) as compared to the Haswell GT2 baseline cannot be captured by a constant multiplier to the baseline performance; more complex predictive models are required.

## 2.3 HALWPE Regression Models

HALWPE includes twelve linear and one non-linear regression models, which are presented in Fig. 2:6 We produce 10 least-squares variants; two the standard OLS and NNLS approach, the remaining 8 are the combinations of employing 4 feature selection variants with each. Feature selection is performed using a combination of forward stepwise selection, backward stepwise selection and evaluation of the selected features using the Akaike Information Criterion (AIC) [30] and the Bayesian Information Criterion (BIC) [31]. We also leverage the Lasso regularization model, a non-negative Lasso and the non-linear Random Forest (RF) Model. We choose the model that yields the smallest $E_{out}$ as the most accurate.

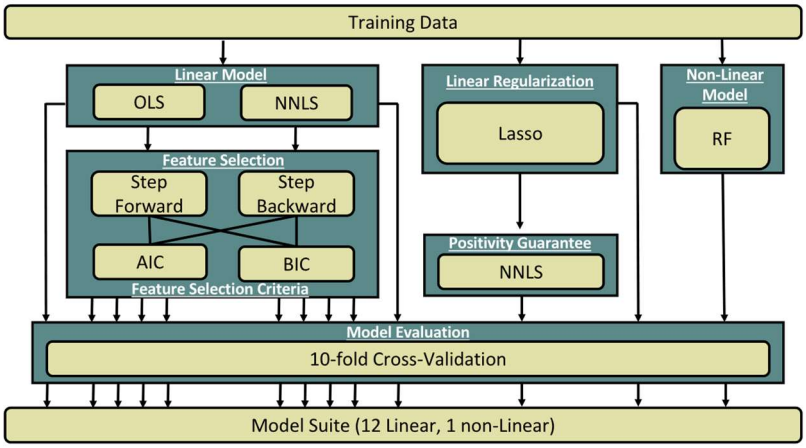


Figure 2:6 HALWPE Regression Model Suite
The generated suite of 13 regression models. 10 OLS and NNLS variants, 2 Lasso regularization variants and one non-linear RF model.

The choice to use multiple models is driven by the fact that the best model is data dependent, and that each scenario exhibits different relationships between host features and target CPF. The OLS and NNLS models are useful when the relationship is linear, and features are non-

correlated; using the AIC and BIC criteria to remove features can simplify the model and help to avoid overfitting.

Linear regularization, shown in the middle, selects features during model construction, by moving their coefficient closer to zero, helping reduce variance and noise on the prediction curve and can improve model accuracy. RF models non-linear behavior, prevalent as the generation gap between host and target grows. A larger gap may necessitate using new models such as Neural Networks to maintain accuracy.

### 2.3.1 Linear Regression Overview

Let $M$ be the number of workloads and $X = [x_1, x_2, \ldots, x_N]$ be the set of features, i.e., the values of the performance counters that we measure for each workload. A *model* is a function $f$ that computes a scalar predicted performance value, $\hat{y} = f(X)$. Under a linear model, $f$ has the form:

$$f(X) = \sum_{j=j}^{N} x_j \beta_j + \beta_0$$

Equation 2:1 – Linear Regression Model Form

where $\beta = \{\beta_1, \beta_2, \ldots, \beta_N\}$ is a coefficient vector that corresponds to the features, and $\beta_0$ is a bias term called the *intercept*, which serves as a model adjustment factor. The *error* associated with the $i^{th}$ workload is $y_i - f(X_i)$, where $y_i$ is the empirically obtained CPF, and $f(X_i)$ the predicted CPF. Given training data, the generation of a coefficient vector is formulated as a constrained optimization problem [32]. The model generation techniques employed by HALPWE differ in terms of the optimization problem formulation and how it is refined by post-processing steps (Fig. 2:6).

### 2.3.2 Ordinary Least Squares

Given a coefficient vector $\beta$, the *aggregate error* of the training data set is the *Residual Sum of Squares (RSS)*. *Ordinary Least Squares (OLS)* computes the coefficient vector $\beta$ and intercept $\beta_0$ that minimizes $RSS(\beta)$ [32].

$$RSS(\beta) = \frac{1}{N}\sum_{i=1}^{M} ||(y_i - f(X_i))^2.$$

Equation 2:2 – Ordinary Least Squares RSS

### 2.3.3 Non-Negative Least Squares

OLS may produce models that estimate negative CPF values for certain data sets, which is physically impossible. *Non-Negative Least Squares (NNLS)* [33] can be applied to ensure that model estimates cannot be negative. NNLS implicitly removes certain features from model by setting negative-valued coefficients to zero and distributing their impact amongst the remaining positive values. NNLS may degrade model accuracy as it no longer minimizes *RSS(β)*. NNLS is equivalent to the quadratic programming problem of the form below:

$$RSS(\beta^{NNLS}) = \ argmin_{\beta,\beta 0 \geq 0} \frac{1}{N} \sum_{i=1}^{M} || \left(y_i - f(X_i)\right) ||_2^2$$

Equation 2:3 – Non-Negative Least Squares RSS

### 2.3.4 Feature Selection and Ranking

OLS and NNLS are *full regression* models that may use all input features. *Feature selection*, which removes feature $x_j$ from the model by setting coefficient $\beta_j$ to zero, can improve *prediction accuracy* by sacrificing bias to reduce variance, as well as *interpretation*: identifying a subset of features that exhibits the strongest effect on model accuracy enhances understanding of the underlying mechanisms [34].

We employ two stepwise feature selection methods [34]. *Forward Stepwise Selection* iteratively and greedily build the feature subset by selecting coefficient pairs that achieve the maximal incremental improvement to the model; the process terminates when adding more features is no longer beneficial to model prediction accuracy. *Backward Stepwise Selection* is similar but starts with a full regression model and iteratively removes one feature at a time. We apply AIC and

BIC as feature ranking criteria during stepwise selection. This provides us with four feature selection methods: {Forward, Backward}×{AIC, BIC}, which can be applied to either OLS or NNLS models.

For each model, we *rank* the selected features via *p-value hypothesis testing* [32] using a threshold of 0.05 to quantify their impact on model accuracy (a smaller p-value indicates greater significance). We report p-values for models that perform feature selection, omitting full regression models. We do not rank features for NNLS models, because the NNLS process discards features that break the assumption that model residuals follow a normal distribution.

### 2.3.5 Linear Regularization via Lasso

*Lasso* [35] is a *linear regularization model* that constructs a model while simultaneously selecting features using an RSS penalty term [32]. Lasso penalizes features in a blanket fashion, unlike step-wise selection, which is iterative. Lasso selects features via shrinkage, which reduces "small enough" coefficients to zero, depending on the value of the regularization term coefficient. We produce two variants of a Lasso, with and without the NNLS criterion. Below we present the RSS computation for the Lasso Regularization model, which is closely related to the NNLS model.

$$RSS(\beta^{Lasso}) = \ argmin_{\beta,\beta 0 \geq 0} \frac{1}{N} \sum_{i=1}^{M} || \left(y_i - f(X_i)\right) ||_2^2 + \alpha_1^T \beta$$

$$\text{Subject to } \alpha > 0$$

Equation 2:4 – Lasso Regularization RSS

### 2.3.6 Model Evaluation

We use 10-fold CV [32] to estimate model *generalizability*, i.e., the models predictive capability when applied to unseen data. An example is presented in Fig. 2:7. 10-fold CV randomly partitions the training data (size *M*) into 10 sets of size *M/10*. One partition is retained as a validation set; the remaining 9 train the model. This process repeats 10 times, with each partition used once

as the validation set. We compute the *Mean Absolute relative Percentage Error (MAPE)* for each CV fold, and take the average to produce the $E_{out}$ [13]:

Let $\hat{f}$ be the fitted module under evaluation. We define a function $k: \{1, \dots, M\} \rightarrow \{1, \dots, K\}$, $K = 10$, to associate the index of feature $X_i$ with its CV fold. We then define $\hat{f}^{-k(i)}$ to be the fitted function computed with the $k^{th}$ CV fold removed. $E_{out}$ is then computed as follows:

$$E_{out}(\hat{f}) = \frac{100}{M} \sum_{i=1}^{M} \left| \frac{y_i - \hat{f}^{-k(i)}(X_i)}{y_i} \right|$$

Equation 2:5 – Out-Of-Sample Error Calculation

The *Absolute Relative Percentage Error* (APE) of a feature vector (trace) $X_i$ is

$$APE(X_i) = 100 \left| \frac{y_i - f(X_i)}{y_i} \right|$$

Equation 2:6 – Average Relative Absolute Percentage Error

We also evaluate models in terms of their *inlier ratios* (IR). Given a percentage threshold $T$, a trace $X_i$ is called an inlier if $APE(X_i) \leq T$, and an outlier otherwise. Given $T$, the $I_R$ is the percentage of traces that are inliers, i.e.:

$$I_R(f, T) = \frac{100}{M} |\{X_i | APE(X_i) \leq T, 1 \leq i \leq M\}|$$

Equation 2:7 – Inlier Ratio Calculation

Intuitively, the $I_R$ can be interpreted as a measure of variance in the model error. At a given threshold, a model with a higher $I_R$ would seem less likely to produce an anomalous prediction (outlier) on a new trace than a model with a lower $I_R$, even if the latter model has a lower $E_{out}$.

We report 10 % and 20 % IRs for each model we produce and compare IRs across varying thresholds for comparative analysis of the prediction scenarios. An example of $I_R$ at T=20% is presented in Fig. 2:8.

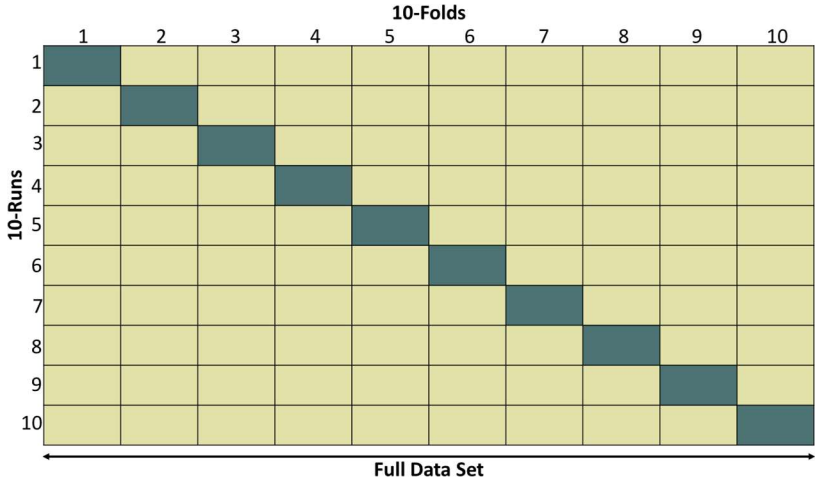


Figure 2:7 10-Fold CV Example

Training data is partitioned into 10 equal sized folds, one-fold is treated as the validation set and the other 9 as a training set. The model is trained repeatedly, selecting a new fold as the validation set each time. The error on the validation sets is computed and then averaged to estimate the generalizability of the model onto unseen data.

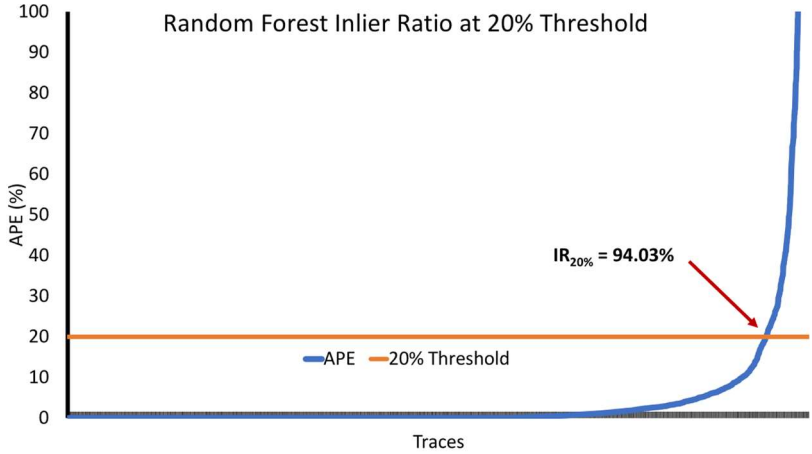


Figure 2:8 $IR_{20\%}$ Example

The $I_R$ is the percentage of data points that have less than T% error. T is 20% in this example.

### 2.3.7 Random Forest

RF regression is a non-linear supervised learning model where prediction is an aggregate of individual predictions made by a set of regression trees. Due to space limitations, we omit describing RF in detail; interested readers may consult Ref. [36] for detail. We construct our RF using *bootstrap aggregation (bagging)*, applying *feature bagging* to reduce correlation among trees. We compute $E_{out}$ using 10-fold CV, by averaging the *out-of-bag error* for each fold.

Regression trees and forests include all features by design. Although feature ranking via RSS error is performed, we do not report any feature rankings produced by RF models. Chapter 3, wherein RF regression is the most accurate provides further detail on the RF model building and evaluation process and presents feature ranking results.

## 2.4 HALWPE Simulator Based Models

Although the focus of this work is hardware-assisted modeling, we first create simulation-based models. This allowed us to build confidence in the cross-generational modeling approach by evaluating the broader capability of the hardware-assisted technique.

Using simulation-based modeling is easier, as it removes the well-known difficulties inherent to hardware-assisted modeling, such as limited architectural visibility, run-to-run noise cycle count and performance counter variations. Simulation modeling also provides greater degree control of the degree of difference between the generations of devices configured as host and target during model building, allowing us to isolate and evaluate well known design time tests.

### 2.4.1 HALWPE Framework Validation Scenarios

We created 3 simulation-based prediction scenarios which were designed to be easy, thereby enabling us to validate our modeling suite. We used the GPU simulator to collect performance counters *and* to model the prediction target CPF. The simulator eliminates sources of non-determinism that can affect hardware-assisted models (see Section 6).

**Scenario$_1$ (364 traces)** configures the simulator as a 2-slice Broadwell GT3 and builds a model to predict its own CPF.

**Scenario$_2$ (364 traces)** configures the simulator as a 2-slice Skylake GT3 and builds a model to predict its own CPF.

**Scenario$_3$ (364 traces)** configures the GPU simulator as a 2-slice Broadwell GT3 running a Broadwell-generation driver and builds a model to predict the performance a 2-slice Skylake GT3

running the same driver. Although both GPUs have 48 EUs, the evolution from Broadwell to Skylake does include microarchitecture changes not reported in Table 2:2.

We generated 13 models for each scenario. For each model, we report the $E_{out}$, 10 % and 20 % IR, the number of selected features, and the number of available features; we also report the APE for each workload.

Tables 2:4 and 2:5 respectively report the best-performing non-NNLS and NNLS models that minimized the $E_{out}$ for each of three scenarios listed above; Figs. 2:8 and 2:9 depict the observed CPF, predicted CPF, and APE for each workload for the three models listed in Tables 4 and 5. For all three scenarios, the best non-NNLS models produced lower $E_{outs}$ than the best NNLS models.

### 2.4.2 HALWPE Non-NNLS Models

The three models reported in Table 2:4 exhibit very low $E_{outs}$; the RF model for $Scenario_3$ had a slightly higher $E_{out}$ than the OLS/Backward/AIC models for $Scenario_1$ and $Scenario_2$, which is to be expected because it is a cross-generation prediction scenario, whereas $Scenario_1$ and $Scenario_2$ are same-generation. All three models obtained 10 % IRs of more than 80 % and varied slightly in terms of the number of selected features.

In Fig. 2:9, it is near-impossible to discern the difference between predicted and observed CPF for most workloads with the naked eye, which reinforces the accuracy of these models.

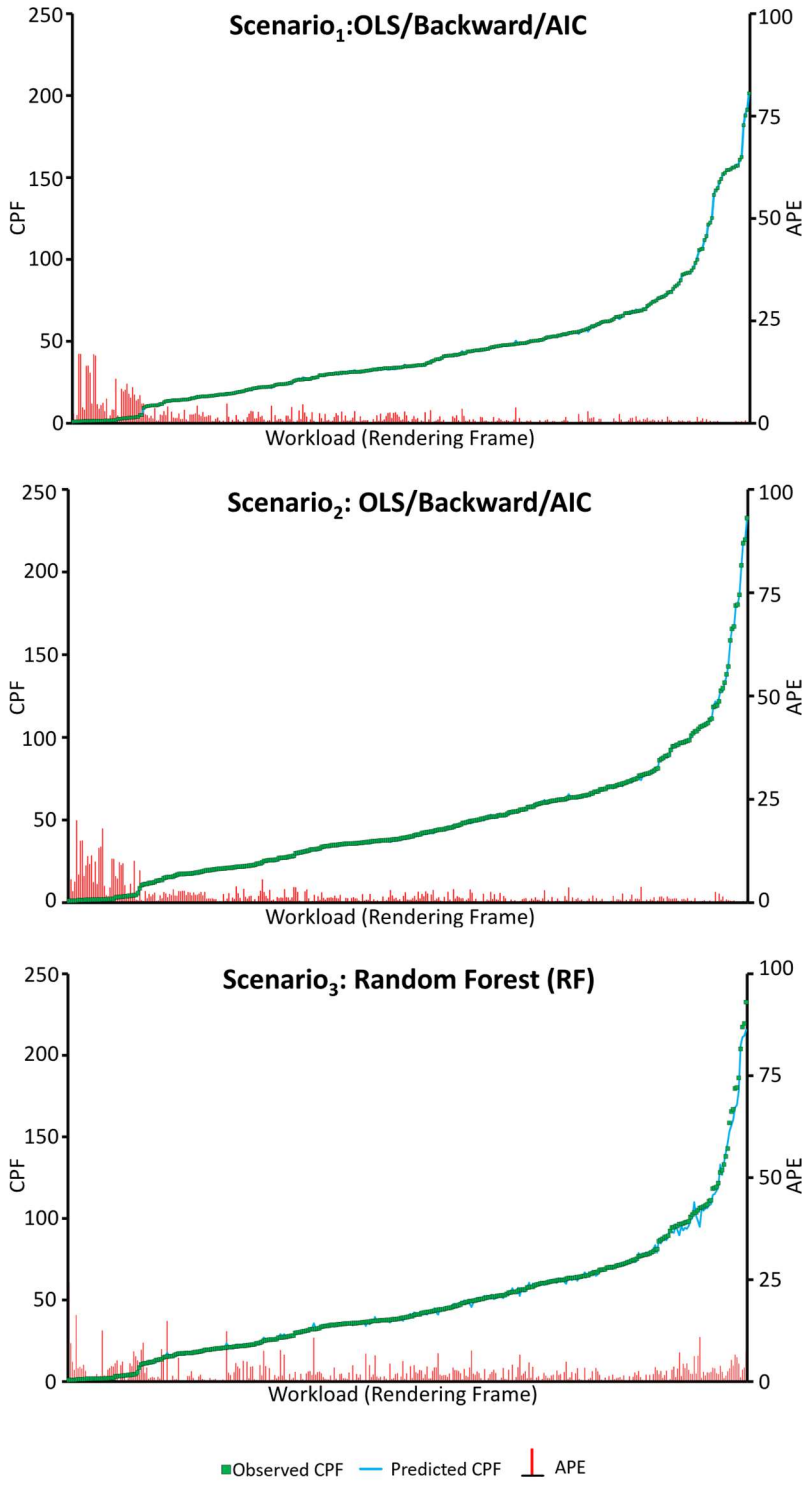


Figure 2:9 Scenario3 Non-NNLS Model Errors

The observed CPF, predicted CPF, and per-workload APE for the best performing non-NNLS models of Scenario1 (a), Scenario2 (b), and Scenario3 (c). Workloads are ordered from left-to-right in non-decreasing order of observed CPF.

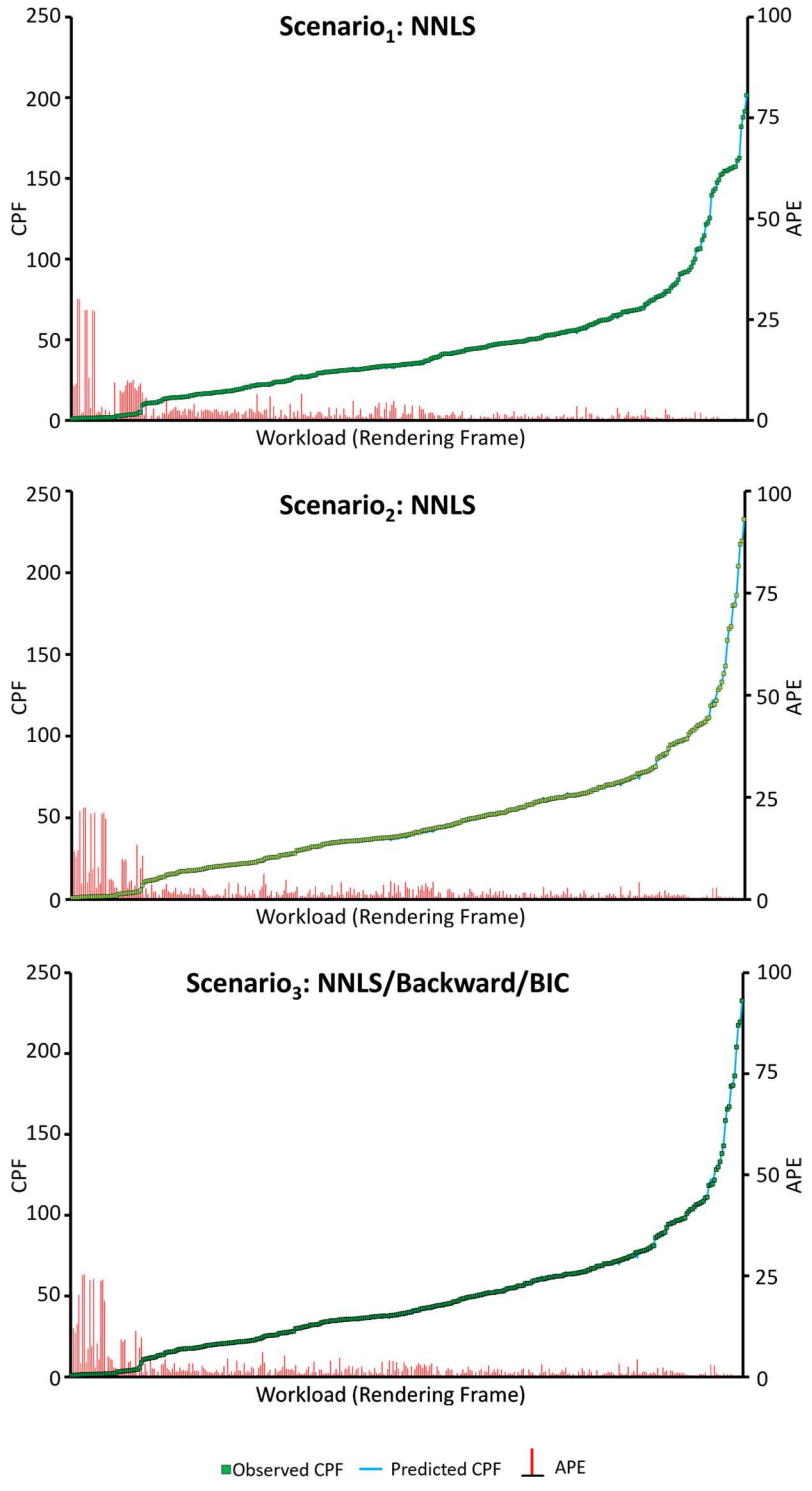


Figure 2:10 Scenario$_3$ NNLS Model Errors

The observed CPF, predicted CPF, and per-workload APE for the best performing NNLS models of Scenario$_1$ (a), Scenario$_2$ (b), and Scenario$_3$ (c). Workloads are ordered from left-to-right in non-decreasing order of observed CPF.

| Scenario | Best Performing NNLS Model | $E_{out}$ | $I_R$ (20%) | $I_R$ (10%) | Features Selected | Available Features |
|---|---|---|---|---|---|---|
| $Scenario_1$ | NNLS | 2.22% | 98.36% | 97.27% | 19 | 42 |
| $Scenario_2$ | NNLS | 2.25% | 97.54% | 95.9% | 15 | 42 |
| $Scenario_3$ | NNLS/Backward/BIC | 8.95% | 88.25% | 74.04% | 8 | 42 |

Table 2:4 $Scenario_{1\text{-}3}$ NNLS Model Results

| Scenario | Best Performing Model | $E_{out}$ | $I_R$ (20%) | $I_R$ (10%) | Features Selected | Available Features |
|---|---|---|---|---|---|---|
| $Scenario_1$ | OLS/Backward/AIC | 1.75% | 100% | 97.26% | 34 | 42 |
| $Scenario_2$ | OLS/Backward/AIC | 1.89% | 91.77% | 82.79% | 31 | 42 |
| $Scenario_3$ | Random Forest (RF) | 3.20% | 98.91% | 87.95% | 38 | 42 |

Table 2:5 $Scenario_{1\text{-}3}$ Non-NNLS Model Results

The highest APEs are observed for workloads with the smallest CPFs on the left-hand side of the graphs, which suggests that the three models are stable; slightly higher APEs for workloads with large CPFs are observed for Scenario3's RF model, which we attribute to the fact that Scenario3 entails cross-generation prediction.

### 2.4.3 HALWPE NNLS Models

Comparing the non-NNLS models of Table 2:4 and Fig. 2:9, the NNLS models reported in Table 2:5 and Fig. 2:10 have a higher $E_{out}$ while selecting fewer counters as features; however, when looking at Fig. 2:10 in detail, virtually all the visible increase in per-workload APE occurs for the workloads with the smallest CPFs. The gap in predictive accuracy between NNLS and non-NNLS models may not be as pronounced as one might interpret by considering $E_{out}$ in isolation.

$Scenario_3$'s NNLS/Backward/BIC model has lower APEs for large-CPF workloads than $Scenario_3$'s non-NNLS RF model, which has a lower overall out-of-sample error. Likewise, $Scenario_3$'s NNLS/Backward/BIC model has a lower 10 % $I_R$ than the non-NNLS RF model; however, the outliers are clustered among workloads with the smallest CPFs. Similar, observations hold for $Scenario_1$ and $Scenario_2$ as well.

### 2.4.4 HALWPE Driver Scalability Scenario

**Scenario$_{3D}$ (364 traces)** re-runs Scenario$_3$, modifying the target simulator to produce new validation data obtained used the same Skylake GT3 device, only now running the applications with the Skylake GT3 driver instead of the Broadwell GT3 driver.

Updating the driver to increase application performance is a common optimization made by GPU designers, and it is imperative that predictive models can accurately predict generational changes in both the hardware and software stack. Scenario$_{3D}$ shows that the HALWPE regression suite selects features and produces models that account for the performance difference caused by updating the target platforms driver, producing accurate $E_{out}$ estimates.

Table 2:6's last row presents the best performing non-NNLS model for Scenario3D and the first-row repeats Scenario3's best model result from Table 2:4. Table 2:7's last row presents the best performing NNLS model, and the first row repeats the last row of Table 2:5. Most notably, the out of sample error of the best performing NNLS model in Table 2:7 has decreased from 8.95 % to 7.40 % using the same features without retraining. The increased accuracy is a byproduct of using a small number of traces, 32, for validation. The reduced frame count is unavoidable as both driver versions were not compatible with all frames. This study demonstrates HALWPE's robustness when modeling driver generation updates.

| Scenario | Best Performing Model | $E_{out}$ | $I_R$ (20%) | $I_R$ (10%) | Features Selected | Available Features |
|---|---|---|---|---|---|---|
| Scenario$_3$ | OLS/Backward/AIC | 1.75% | 100% | 97.26% | 34 | 42 |
| Scenario$_{3D}$ | Lasso | 6.41% | 93.55% | 74.19% | 23 | 42 |

Table 2:6 Highest Accuracy Non-NNLS Models Scenario$_{3D}$.

| Scenario | Best Performing NNLS Model | $E_{out}$ | $I_R$ (20%) | $I_R$ (10%) | Features Selected | Available Features |
|---|---|---|---|---|---|---|
| Scenario$_3$ | NNLS/Backward/BIC | 8.95% | 88.25% | 74.04% | 8 | 42 |
| Scenario$_{3D}$ | Lasso/NNLS | 7.40% | 93.55% | 77.42% | 11 | 42 |

Table 2:7 Highest Accuracy NNLS Models Scenario$_{3D}$.

### 2.4.5 HALWPE Slice Scalability Scenario

**Scenario$_{3S}$ (364 traces)** re-runs Scenario$_3$, modifying the training data to use a Broadwell GT2 device to predict the original Skylake GT3 device. Modifying the training data approximates a scenario in which the host platform is both a generation older and has half the available parallelism of its target. It is likely that as new generations of GPU are developed and tested, their slice count will continue to increase.

The model is applied without retraining to a validation set with 60 single frame workloads, demonstrating HALWPE's ability to identify features that accommodate CPF changes due to slice scaling, obtaining high cross-generation prediction when the host has 200 % less parallelism than the target.

Row 1 of Tables 2:8 and 2:9 re-report Scenario$_3$ results, and Row 2 reports the results of Scenario$_{3S}$'s slice scaling study. Comparing Scenario$_3$ and Scenario$_{3S,}$ the error increases 2.3 %, resulting in an 11.5 % average error, but remains usable for pre-silicon performance estimation. The increase in error due to slice scaling is also present in our hardware-assisted scenarios.

| Scenario | Best Performing Model | $E_{out}$ | $I_R$ (20%) | $I_R$ (10%) | Features Selected | Available Features |
|---|---|---|---|---|---|---|
| Scenario$_3$ | Random Forest (RF) | 3.20% | 98.91% | 87.95% | 38 | 42 |
| Scenario$_{3S}$ | Random Forest (RF) | 4.81% | 96.67% | 93.33% | 38 | 42 |

Table 2:8 Highest Accuracy Non-NNLS Models Scenario$_{3S.}$

| Scenario | Best Performing NNLS Model | $E_{out}$ | $I_R$ (20%) | $I_R$ (10%) | Features Selected | Available Features |
|---|---|---|---|---|---|---|
| Scenario$_3$ | NNLS/Backward/BIC | 8.95% | 88.25% | 74.04% | 8 | 42 |
| Scenario$_{3S}$ | Lasso/NNLS | 9.75% | 83.33% | 66.67% | 11 | 42 |

Table 2:9 Highest Accuracy NNLS Models Scenario$_{3S.}$

## 2.5 HALWPE Hardware-Assisted Models

When profiling an application on commodity hardware, certain sources of non-determinism may arise that simulators either do not model or can suppress. We discuss strategies to mitigate these issues in detail before moving on to present the results of our hardware-assisted models.

### 2.5.1 GPU Profiling and Mitigating Variation

Referring to Fig. 2:3, *HWTraces* refers to a stream of DirectX *GfxAPI* commands that we collect without firmware or driver modification using *GfxCapture*. *HWTraces* are repeatable, platform-independent, and allow instrumentation of the host GPU and API. We attach *GfxProfiler* directly to the *device context* [20], which is created along with its *device* when the GPU renders a frame. The device creates resources and queries the GPU's rendering capabilities, while the device context comprises the GPU's pipeline and resource states, which generate actual rendering commands.

*GfxProfiler* collects three classes of features: performance counter measurements (via HWTraces), profiled DirectX API commands (via HWTraces), and *hardware queries* (via the device context) which leverage exposed parts of the API. An exemplary hardware query is *PSInvocations*, the number of times the pixel shader (PS) invoked an EU while rendering.

Workload execution is performed using an unmodified operating system (OS; Windows 7) and driver. To reduce variability introduced by the OS, we suppress non-OS background processes and run traces in full-screen mode. By leaving the OS and driver unmodified, we eschew control of sleep states. By adjusting BIOS settings, we can disable deep sleep state RC6 and suppress dynamic frequency scaling and Turbo Boost. The sources of variation that remain are competing background tasks, which affect CPU-GPU communication latency, and access to shared resources, and the sleep states that we cannot control.

We perform outlier detection and elimination to mitigate variation. We apply the *Median Absolute Deviation (MAD)* test [37] to identify runs that exhibit abnormal behavior. We empirically determined a threshold of ±7 MADs using 10 representative frames, executing each frame 100 times. During model construction and evaluation, we execute each frame 100 times on the host GPU using *GfxProfiler* to collect features. We remove outliers, i.e., all runs whose CPF values are outside of the ±7 MAD threshold. The CPF and feature values reported for the frame are averaged across the inliers.

Fig. 2:11 reports the CPF of 100 executions of Witcher 2 Frame 769 normalized to the smallest CPF that we observed. To avoid cold-start issues, we insert a generic "warmup" frame that is executed but not profiled. Most of executions are within the MAD window, although some non-negligible variation in CPF is clearly visible.

### 2.5.2 Prediction Scenarios

We present three hardware-assisted predictive models based on performance counter measurements taken from a Haswell GT2 GPU, which provides 577 features. The results show that HALWPE can perform accurate cross-generation CPF prediction.

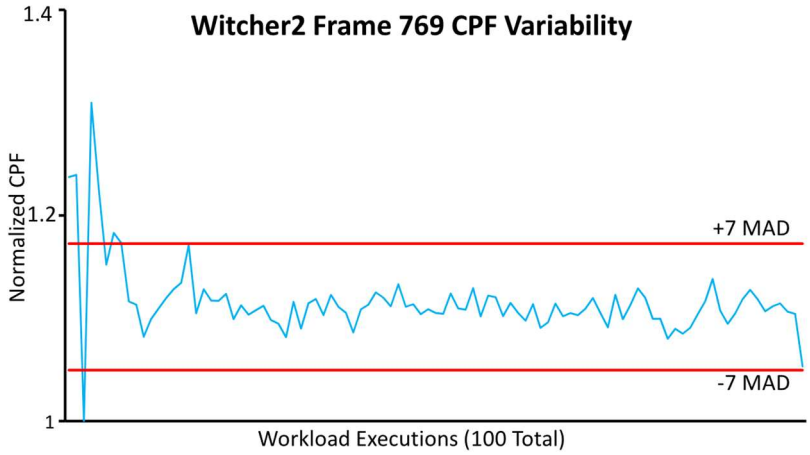


Figure 2:11 CPF Variability for one frame of Witcher 2.
Witcher 2 Frame when executed 100 times; the first execution is removed due to cold start issues. Of the remaining 99 runs, 7 frames were identified as outliers and removed using the ±7 MAD approach.

**Scenario$_4$ (282 traces)** uses a Haswell GT2 GPU host to predict the CPF of a simulated Broadwell GT2 GPU.

**Scenario$_5$ (300 traces)** uses the host to predict the CPF of a simulated Broadwell GT3 GPU.

**Scenario$_6$ (300 traces)** uses the to predict the CPF of a simulated Skylake GT3 GPU.

Tables 2:10 and 2:11 respectively report the best-performing non-NNLS and NNLS models that minimized the $E_{out}$ for each of three scenarios listed above; Figs. 2:12 and 2:13 depict the observed CPF, predicted CPF, and APE for each workload for the three models listed in Tables 2:10 and 2:11. For Scenario$_4$ and Scenario$_5$, OLS/Forward/BIC produced the lowest $E_{outs}$; for Scenario$_6$, the NNLS produced the lowest $E_{out}$.

### 2.5.3 Non-NNLS Models

In Fig. 2:12, slight differences between predicted and observed CPF for the OLS/Forward/BIC model for Scenario$_4$ and Scenario$_5$ can be seen by the naked eye; the differences are more pronounced for Scenario$_6$'s RF model, especially for workloads with higher CPFs. The degradation in model quality is clear between scenarios.

The OLS/Forward/BIC models generated for Scenario$_4$ and Scenario$_5$, exhibited the largest APEs are at the low-CPF end up the spectrum; in contrast, the RF model generated for Scenario$_6$ has a more uniform distribution of high APEs across the CPF spectrum. This is like the distribution of APEs reported for the RF model in Fig. 2:9 for Scenario$_3$.

| Scenario | Best-Performing Model | $E_{out}$ | $I_R$ (20%) | $I_R$ (10%) | Features Selected | Available Features |
|---|---|---|---|---|---|---|
| Scenario$_4$ | OLS/Forward/BIC | 7.45% | 92.76% | 85.13% | 40 | 577 |
| Scenario$_5$ | OLS/Forward/BIC | 7.47% | 91.33% | 83.67% | 30 | 577 |
| Scenario$_6$ | Random Forest (RF) | 10.28% | 95.85% | 62.85% | 577 | 577 |

Table 2:10 Scenario$_{4\text{-}6}$ Non-NNLS Model Results

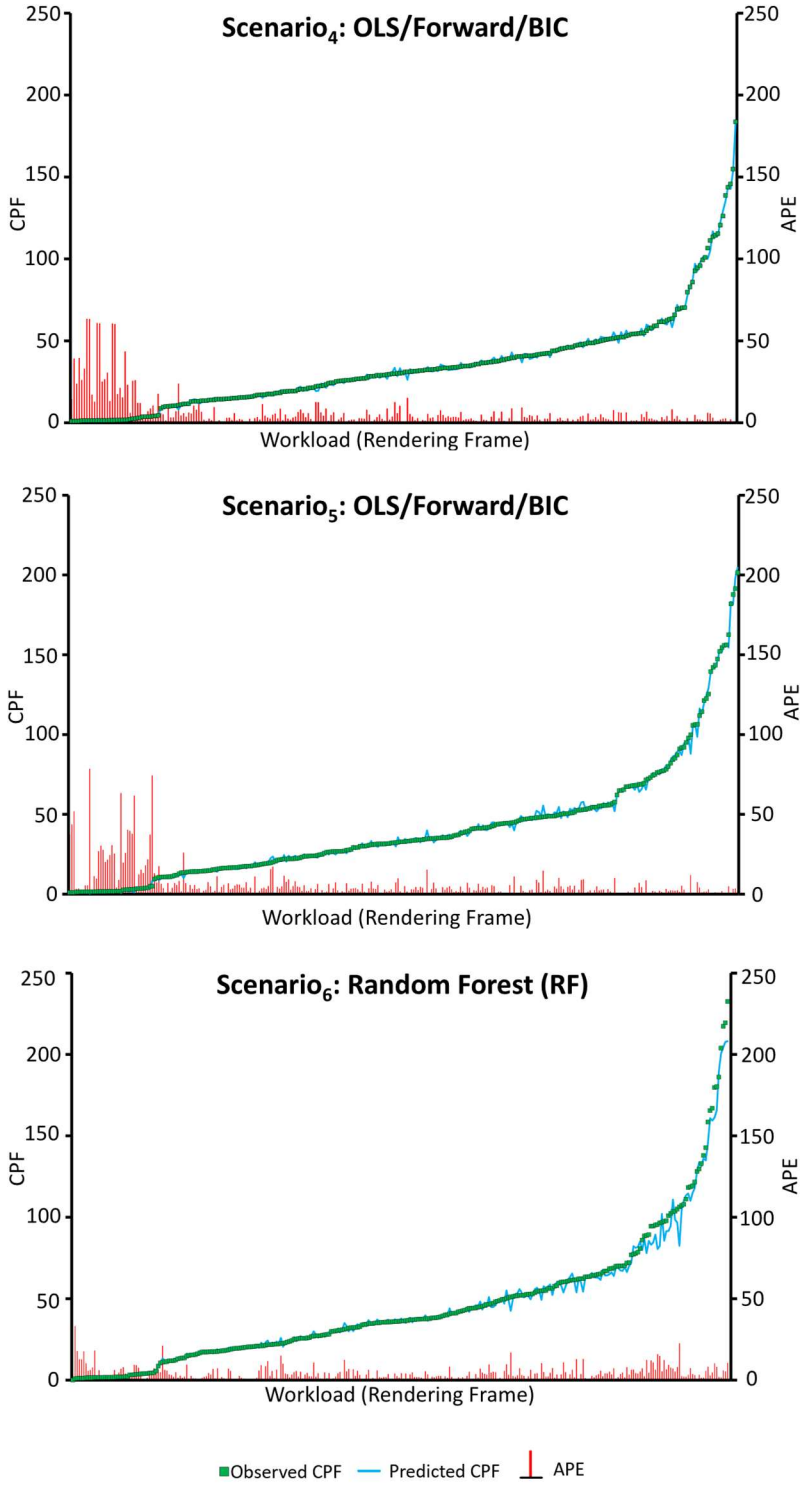


Figure 2:12 Scenario$_6$ Non-NNLS Model Errors

The observed CPF, predicted CPF, and per-workload APE for the non-NNLS models summarized in Table 2:6. Workloads are ordered from left-to-right in non-decreasing order of observed CPF.

### 2.5.4 NNLS Models

The NNLS models produced for $Scenario_4$ and $Scenario_5$ in Table 2:11 nearly double the $E_{out}$ produced by the non-NNLS models in Table 2:10, with large reductions in the 10 % IRs in both cases. In the case of $Scenario_6$, the NNLS model yielded an $E_{out}$ of 8.91 %, which is slightly worse than the 7.45 % and 7.47 % produced by the best non-NNLS models for $Scenario_4$ and $Scenario_5$ in Table 2:10, but respectable given the challenges associated

with CPF prediction across two GPU generations; it's 10 % $I_R$ was respectively 14.23 % and 12.77 % lower, which can be explained similarly.

| Scenario | Best-Performing NNLS Model | $E_{out}$ | $I_R$ (20%) | $I_R$ (10%) | Features Selected | Available Features |
|---|---|---|---|---|---|---|
| $Scenario_4$ | Lasso/NNLS | 13.87% | 75.06% | 58.95% | 23 | 577 |
| $Scenario_5$ | NNLS/Backward/AIC | 13.68% | 84.00% | 73.67% | 83 | 577 |
| $Scenario_6$ | NNLS | 8.91% | 92.63% | 77.89% | 59 | 577 |

Table 2:11 $Scenario_{4\text{-}6}$ NNLS Model Results

This level of accuracy should be sufficient for use in early-stage DSE; however, designers must understand that model accuracy will necessarily degrade as number of generations between the host and prediction target increases. Scenario6 investigates this issue further.

### 2.5.5 $Scenario_6$ Model Comparison

Table 2:12 reports the accuracy of all 13 HALWPE models for $Scenario_6$. This study serves to justify the need for an ensemble of models, by assessing the differences in model accuracy in our most ambitious CPF prediction scenario, across two GPU generations. The $E_{out}$ ranged from 8.91 % (NNLS) to more than 1000 % (four OLS variants); the four highly inaccurate OLS variants likely overfit the training data. Employing an ensemble of models increases the likelihood that at least one model does not overfit. Both RF (which is nonlinear) and Lasso (due to regularization) are less likely than OLS and NNLS variants to overfit the training data; including them in HALWPE's ensemble increases the likelihood that at least one model is accurate. RF, Lasso, and Lasso/NNLS did not overfit in any of our scenarios.

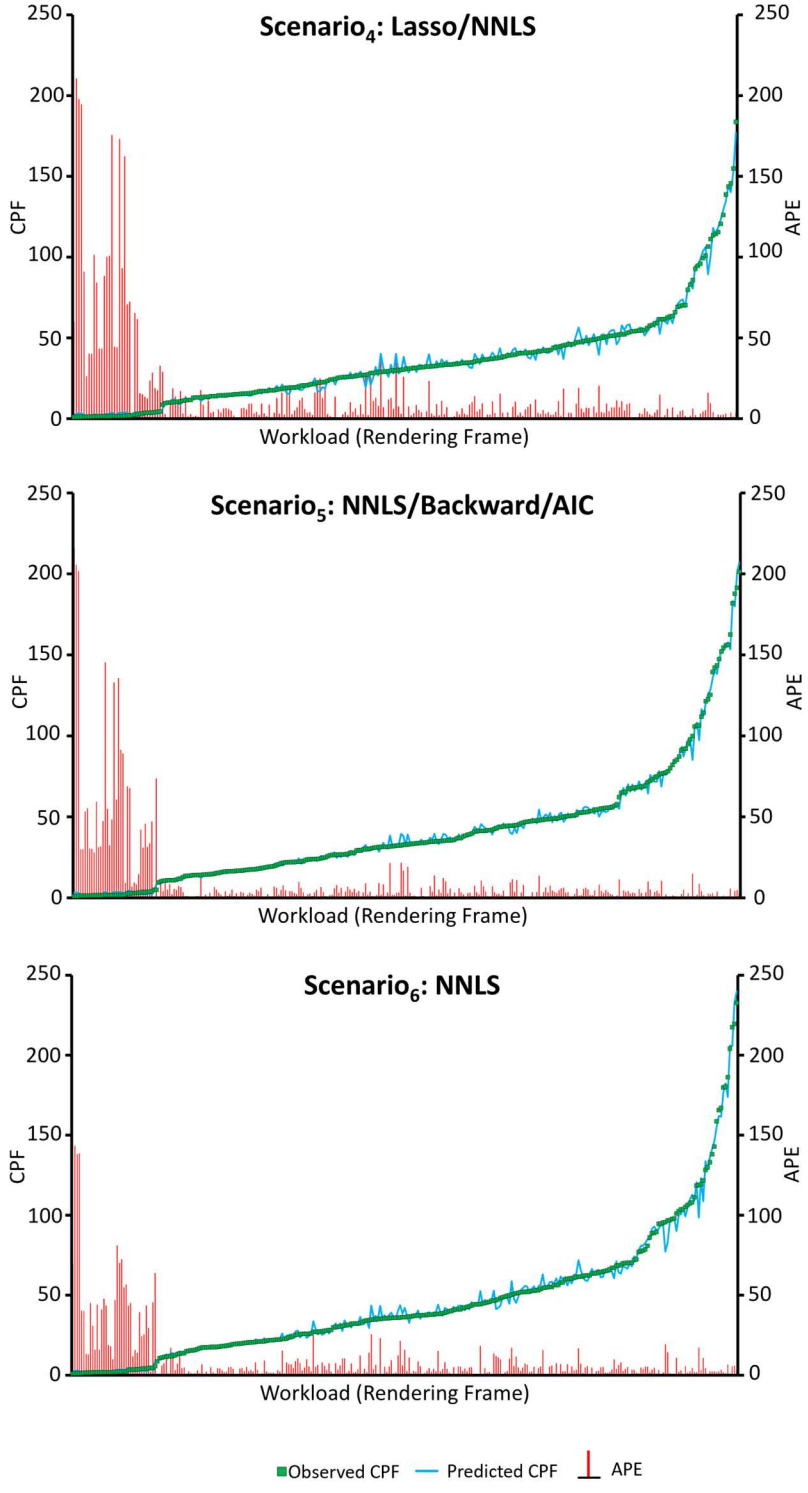


Figure 2:13 Scenario$_6$ NNLS Model Errors

The observed CPF, predicted CPF, and per-workload APE for the NNLS models summarized in Table 2:6. Workloads are ordered from left-to-right in non-decreasing order of observed CPF.

Among the remaining nine models in the highest $E_{out}$ was 19.69 % (NNLS/step-forward/BIC). As Tables 2:4-2:11 show, it is difficult to know a-priori which model (with or without the NNLS guarantee) will yield the highest overall accuracy, which justifies the ensemble approach. In Table 2:12, NNLS achieves the smallest $E_{out}$ and is tied for third highest 10 % IR; however, it selects the most features of the remaining nine linear models.

| Model | $E_{out}$ | $I_R$ (10%) | Features Selected | Available Features |
|---|---|---|---|---|
| OLS | >1000% | 4.02% | 299 | 577 |
| NNLS | 8.91% | 70.90% | 59 | 577 |
| OLS/Forward/AIC | >1000% | 9.43% | 299 | 577 |
| OLS/Forward/BIC | 12.51% | 77.89% | 29 | 577 |
| OLS/Backward/AIC | >1000% | 4.02% | 299 | 577 |
| OLS/Backward/BIC | >1000% | 4.02% | 299 | 577 |
| NNLS/Forward/AIC | 14.32% | 66.22% | 53 | 577 |
| NNLS/Forward/BIC | 19.69% | 44.15% | 14 | 577 |
| NNLS/Backward/AIC | 12.78% | 71.57% | 59 | 577 |
| NNLS/Backward/BIC | 12.59% | 70.90% | 59 | 577 |
| Lasso | 12.24% | 61.51% | 12 | 577 |
| Lasso/NNLS | 13.34% | 65.22% | 11 | 577 |
| RF | 10.28% | 62.85% | 577 | 577 |

Table 2:12 $Scenario_6$ All Model Results Comparison

### 2.5.6 Inlier Ratio vs. Out-of-sample Error

Scenarios exist in which a processor architect may prefer a predictive model that maintains a higher $I_R$ within a given threshold to a model that minimizes $E_{out}$. Treating the $I_R$ as a proxy for variance provides higher confidence in the fidelity of the model. Fig. 2:14 depicts the IRs at eight thresholds for five models generated for $Scenarios_{4\text{-}6}$. For each scenario, Fig. 2:14 includes HALWPE's OLS and NNLS variants that minimize $E_{out}$, its two regularization models (Lasso, and Lasso/NNLS), and its non-linear model (RF).

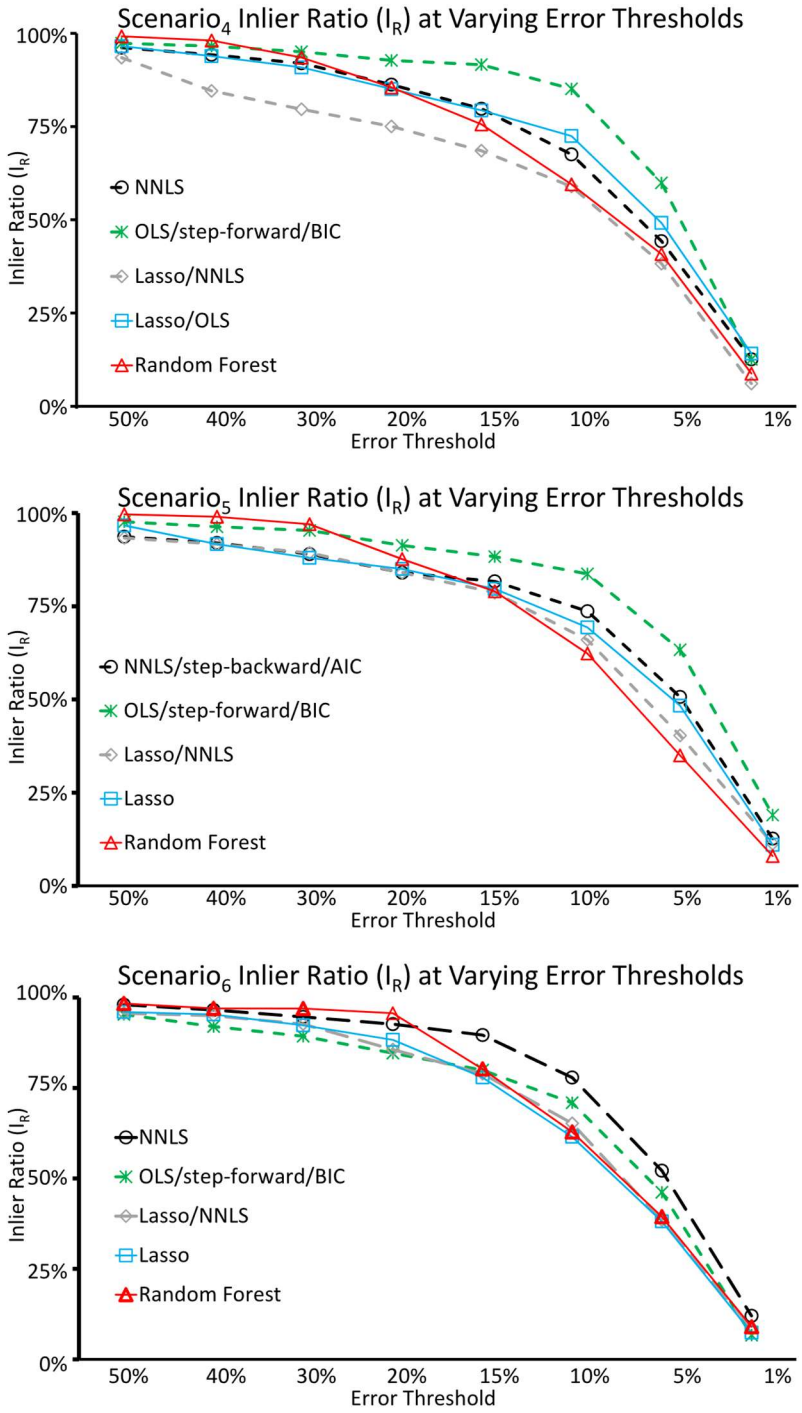


Figure 2:14 Scenarios$_{4\text{-}6}$ IRs at Various Thresholds

IRs at various error threshold for predictive models generated for Scenario$_6$, Scenario$_7$, and Scenario$_8$. For each scenario, the best-performing OLS and NNLS-variants are shown, along with Lasso, Lasso/NNLS, and RF.

For $Scenario_4$ and $Scenario_5$ OLS/Forward/BIC had the smallest $E_{out}$. For Scenario4 in Fig. 2:14, OLS/Forward/BIC has the highest $I_R$ at error thresholds of 30 % or lower, except for the 1 % threshold where NNLS and Lasso/OLS are marginally higher.

For Scenario5 it has the highest $I_R$ at error thresholds of 20 % or below; in $Scenario_6$, NNLS had the smallest $E_{out}$ and the highest $I_R$ for thresholds of 15 % and lower. These observations reinforce the benefits of these three models: they exhibit low $E_{outs}$ and high fidelity, so they can be used with high confidence.

### 2.5.7 Speedup

Compared to CASs that are properly tuned, predictive models sacrifice accuracy to provide computer architects with a rapid result. In the case of HALWPE, the execution time of the model on a given workload entails executing the workload trace on the Haswell host GPU and then applying the linear regression or RF model to the obtained features; in most cases, the latter is negligible. Fig. 2:15 compares the execution time of HALWPE to the simulator configured as a Broadwell GT2, Broadwell GT3, and Skylake GT3 GPU for the 282 common executable traces (Table 2:3). On average, HALWPE achieved a speedup of 29,481x over the Broadwell GT2 simulator, 43,643x over the Broadwell GT3 simulator, and 44,214x over the Skylake GT3 simulator. From workload to workload, the speedups reported in Fig. 2:15 vary considerably; in a few cases, the simulator executed a trace faster than the host GPU. Compared to CASs, predictive models sacrifice accuracy to provide a rapid result.

.

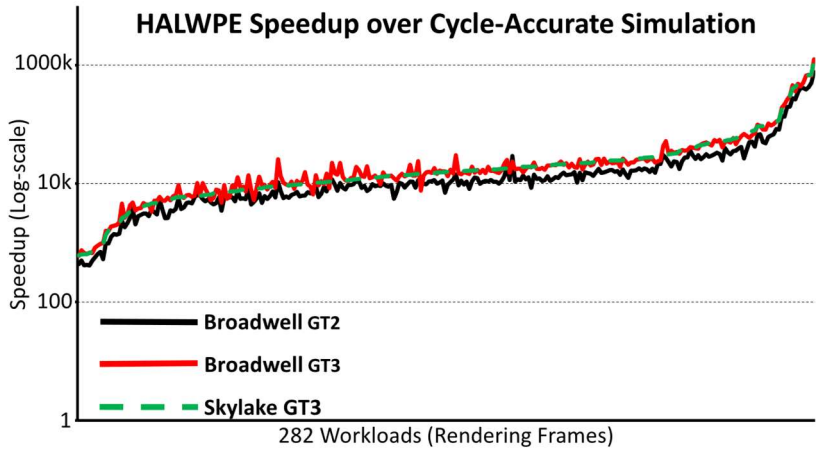


Figure 2:15 Scenarios$_{4\text{-}6}$ HALWPE Computed Speedup

HALWPE speedup over simulators for Broadwell GT2/GT3 and Skylake GT3 GPUs using 280 workloads common to all three simulators (Table 2:3). Frames are ordered by increasing Skylake GT3 speedup. Average speedups were 29481x for Broadwell GT2, 43643x for Broadwell GT3, and 44214x for Skylake GT3.

The execution time of a model on a frame entails running the frame trace on the host GPU and then generating model using the obtained features; in most cases, the latter is negligible.

Fig. 2:16 reports the time to train all 13 HALWPE models --excluding target simulation time and host GPU execution time to render each frame once. In addition to rendering, host GPU execution time includes overhead associated with loading the application, profiling, and streaming API commands from the trace player. The longest model training and host GPU execution time was ~2.5 hours for Scenario$_3$. We rendered each frame 100 times, thus execution time is dominated by the host GPU, not model training.

When comparing Haswell to Skylake the total performance increase is 820.79 % on average, and 242 % on median. HALWPE can provide accurate models, which utilize a Haswell generation GPU to predict the performance of a Skylake generation GPU, as demonstrated in Scenario$_6$.

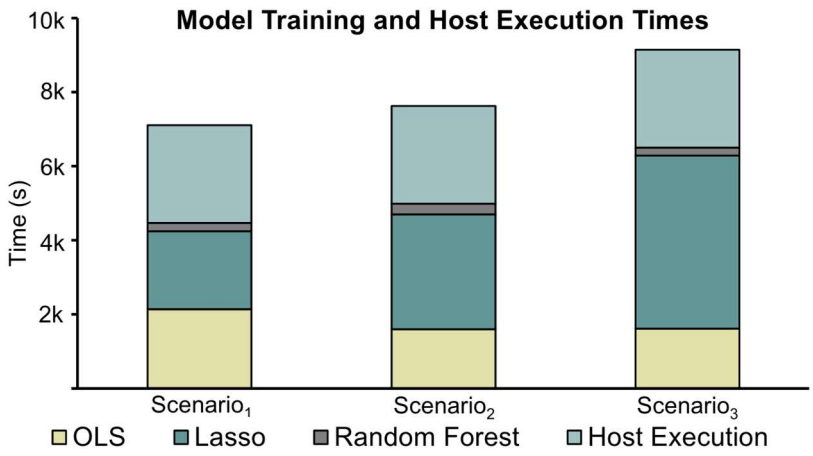


Figure 2:16 Model Training and Host Execution Time
Model Training and Host Execution Times for Scenarios$_{4\text{-}6}$.

## 2.6 Feature Ranking

This section reports the 10 highest ranked features from the models generated for Scenario$_4$, Scenario$_5$, and Scenario$_6$. We rank the features using the OLS/Forward/BIC model, which was the best performing model for Scenario$_4$ and Scenario$_5$, and the third best for Scenario$_6$. In our system, 577 features are available (see Section 6). In the feature list tables presented in the next two subsections, performance counters are yellow rows, DirectX metrics are blue rows, and hardware queries are grey rows. All the features reported in these subsections have been publicly disclosed [2–14].

### 2.6.1 Broadwell GT2/GT3

Tables 2:13 and 2:14 report the top-ten most influential features for the OLS/Forward/BIC models produced for Scenario$_4$ and Scenario$_5$, ranked by p-values (Section 4.D). These scenarios respectively target the Broadwell GT2 and GT3 GPU, where the latter has twice as many slices/EUs.

In Table 2:13, the top-5 highest ranked features provide information about EU activity relating to front-end render pipeline units. The top-2 features are EU active and stall times, which

holds consistent with the observation that the limited parallelism in a single-slice Broadwell GT2 GPU can impede performance. The 3rd through 5th highest ranking features report compute and DS EU activity, which suggests that interactions between the two shaders and their EU occupations should be analyzed. Notably, the presence of A19_CSEUStallTime suggests that limiting the amount of time that the compute shader (CS) stalls the EU array could potentially improve overall GPU throughput.

In Table 2:14, EU busy/stall cycles no longer fall within the top 10 ranked counters, which reflect the increase in parallelism provided by GT3 GPUs. The two counters representing the DS are ranked 8th and 10th, suggesting that they still influence performance, but that other subsystems with higher-ranking features should be given priority for analysis. Again, A19_CSEUStallTime is within the top-5 ranked features, which suggests that the process by which the CS stalls the EU array still influences performance significantly.

| Performance Counter Name, Category, and Description | P-value |
|---|---|
| A00_EUBusyTime (clocks; HW Counter)<br>Number of cycles that all cores spent actively executing instructions. | 1.78 E-125 |
| A01_EUStallTime (clocks; HW Counter)<br>Number of cycles on all cores where the EU was not idle or active. | 2.35 E-83 |
| A12_DSThreadBusyTime (clocks; HW Counter)<br>Number of cycles the domain shader was active on all cores. | 2.02 E-49 |
| A19_CSEUStallTime (clocks; HW Counter)<br>Number of cycles compute shader stalled on cores; all cores also stalled | 2.75 E-49 |
| A14_DSEUStallTime (clocks; HW Counter)<br>Number of cycles domain shader stalled on cores; all cores also stalled | 6.22 E-39 |
| IDirect3DDevice9_DrawPrimitiveUP (count; DirectX Metric)<br>Number of times that data specified by a user memory pointer as a sequence of primitives was rendered. | 4.26 E-21 |
| IDirect3DQuery9_OcclusionQuery_GetData (count; DirectX Metric)<br>Number of all queries for occlusion data. | 3.04 E-18 |
| ID3D11DeviceContext_DrawInstanced (count; DirectX Metric)<br>Number of times non-indexed, instanced primitives were drawn. | 1.72 E-16 |
| A39_OMZTestFail (count; HW Counter)<br>Number of pixels/samples that failed the Z test after pixelshader execution. | 9.81 E-16 |
| ID3D11DeviceContext_ExecuteCommandList (count; DirectX Metric)<br>Number of queue commands issued from a command list onto a device. | 4.56 E-15 |

Table 2:13 Scenario$_4$ Feature Ranking
Top-10 Model Features Ranked by P-Value for the OLS/Forward/BIC Model of Scenario$_4$.

The geometry shader now appears in the top-10 highest ranked features with the inclusion of A22_GSThreadBusyTime, which suggests that as parallelism increases doubles Broadwell GT2 to GT3, additional front-end units start to influence performance.

### 2.6.2 Skylake GT3

Table 2:15 reports the 10 most influential features for the OLS/Forward/BIC model for Scenario6 ranked by p-values. The top two features from Scenario5 (Table 2:14) remain in the top ten for Scenario6. The highest-ranking feature in Table 2:15, the number of cycles not idle, is the second feature in Table 2:14.

This suggests that the ability to provide the render engine with a steady supply of data remains a critical indicator of performance; potential optimizations ensure that the GPU can consume enough data to avoid idle states. In Fig. 2:5, we see that Broadwell GT3 and Skylake GT3 had similar CPF profiles, and that the gap in favor of Skylake 3 was small.

| Performance Counter Name, Category, and Description | P-value |
|---|---|
| ID3D11DeviceContext_ClearUnorderedAccessViewFloat (clocks; DirectX Metric)<br>Number of cycles spent accessing memory through a resource. | 4.95 E-172 |
| A41_GpuBusy (clocks; HW Counter)<br>Number of cycles the render engine was not idle. | 2.48 E-91 |
| IDirect3DDevice9_DrawIndexedPrimitive (clocks, DirectX Metric)<br>Number of cycles spent rendering a primitive into an array of vertices via indexing. | 7.62 E-46 |
| A17_CSThreadBusyTime (clocks; HW Counter)<br>Number of cycles the compute shader was active on all cores. | 1.25 E-31 |
| A22_GSThreadBusyTime (clocks; HW Counter)<br>Number of cycles the geometry shader was active on all cores. | 2.68 E-29 |
| IDirect3DDevice9_DrawPrimitiveUP (clocks; DirectX Metric)<br>Number of cycles spent rendering data specified by a user memory pointer as primitives. | 8.21 E-23 |
| ID3D11DeviceContext_DrawInstanced (count; DirectX Metric)<br>Number of times non-indexed, instanced primitives were drawn. | 9.17 E-19 |
| A14_DSEUStallTime (clocks; HW Counter)<br>Number of cycles domain shader stalled on cores; all cores also stalled. | 1.57 E-18 |
| IDirect3DQuery9_OcclusionQuery_GetData (count; DirectX Metric)<br>Number of queries for occlusion data. | 1.79 E-18 |
| A12_DSThreadBusyTime (clocks; HW Counter)<br>Number of cycles the domain shader was active on all cores. | 3.68 E-18 |

Table 2:14 $Scenario_5$ Feature Ranking
Top-10 Model Features Ranked by P-Value for the OLS/Forward/BIC Model of $Scenario_5$.

Table 2:15 suggests that Skylake GT3 CPF can be predicted foremost by back-end pixel-based metrics corresponding to PS memory activity, PS invocations, and the render engine's ability to pre-emptively avoid pixel processing by killing pixels (A36_RSKillPixelCount); A40, which counts render target writes also corresponds to the back-end. Compared to Table 2:14, the absence of geometry and CS metrics suggests that changes to the Skylake GT3 front end may have removed performance bottlenecks present in Broadwell GT3.

Table 2:15 also includes domain and HS metrics, which correspond to the first and last stages of the DirectX tessellation pipeline. This suggests that tessellation has emerged as a prime indicator for CPF for Skylake, and that it may be a suitable candidate for further optimization.

### 2.6.3 Discussion

This analysis shows how to interpret predictive models to obtain insights regarding which features have the greatest predictive impact on CPF. A highly-ranked feature may hint that a subsystem that could benefit from further architectural improvement; however, models are inexact, and, for a given scenario, feature rankings may vary from model to model. Thus, feature ranking can provide "hints" to understanding performance, not solutions These hints are symptoms, and should not be misconstrued as having diagnostic abilities to validate the existence of the bottlenecks or to identify root causes. Any hint provided by a model should be validated by further simulation before being accepted as fact. Architects should use predictive models judiciously and conservatively and should not misconstrue them as automated substitutes for existing methodologies or human intelligence.

| Performance Counter Name, Category, and Description | P-value |
|---|---|
| A41_GpuBusy (clocks; HW Counter)<br>Number of cycles the render engine was not idle. | 2.15E-84 |
| ID3D11Device_CreateHullShader (count; DirectX Metric)<br>Number of times a hull shader was created. | 4.67E-33 |
| A15_DSThreadsLoadedCount (count; HW Counter)<br>Number of domain shader threads issued to EUs for execution. | 3.82E-18 |
| A36_RSKillPixelCount (count; HW Counter)<br>Number of pixels/samples killed in the pixelshader. | 6.29E-16 |
| IDirect3DDevice10_PSSetConstantBuffers (count; DirectX Metric)<br>Number of calls to allocate constant buffers used by the pixelshader pipeline stage. | 3.44E-15 |
| ID3D11DeviceContext_ClearUnorderedAccessViewFloat (clocks; DirectX Metric)<br>Number of cycles spent accessing memory through a resource. | 1.24E-14 |
| PSInvocations (count; HW Query)<br>Number of pixel shader invocations. | 2.00E-12 |
| IDirect3DDevice9_SetRenderTarget (count; DirectX Metric)<br>Number of allocations of memory regions to declarecolor information for the render target. | 5.87E-12 |
| IDirect3DIndexBuffer9_Unlock (clocks, DirectX Metric)<br>Number of cycles spent attempting to unlock index buffers used by the vertex shader and render engine. | 5.13E-11 |
| A40_OMSamplesWritten (count; HW Counter)<br>Number of 3D render target writes. | 2.04E-10 |

Table 2:15 Scenario$_6$ Feature Ranking

Top-10 Model Features Ranked by P-Value for the OLS/Forward/BIC Model of Scenario$_6$.

# Chapter 3 Cross-Abstraction GPU Performance Estimation

This chapter is based primarily on the contents of the "GPU Performance Estimation using Software Rasterization and Machine Learning" [38]. The underlying observation of this work is that GPU performance depends primarily on architectural innovations and advances in process technology, both of which increase complexity and cost. This necessitates hardware-software co-design, co-development, and co-validation prior to manufacturing. During the design and development stages, GPU architects use pre-silicon detailed performance CASs to explore the architectural design space. Functional simulators, which are faster, can aid development but cannot provide detailed timing information and cannot characterize application performance. To reduce simulation times and the time required to perform early-stage architectural DSE for GPUs, this paper presents a cross-abstraction predictive statistical modeling framework that predicts the performance of a pre-silicon GPU CAS using a functional GPU simulator. GPU architects can use these predictions to explore the architectural design space while rapidly characterizing the performance of far more workloads than would be possible using CAS alone.

The primary differences between this work and HALWPE as described in Chapter 2 are that 1) This work is not cross-generational, and instead focuses on the same, possibly pre-silicon GPU generation under design that the target CAS is configured to model. 2) This work does not leverage previous generation hardware, or hardware assistance, instead replacing the host with a highly-abstracted functional simulator, RastSim. 3) The functional simulator sacrifices speed when compared to HALWPE but allows the explicit modeling and instrumentation of feature additions exclusive to the new generation GPU, which HALWPE is incapable of modeling in its present form.

This work focuses on Intel integrated GPUs, which are customized to accelerate graphics and gaming workloads. The performance overhead of Intel's proprietary simulator is prohibitive for pre-silicon DSE, software performance validation, and analysis of architectural optimizations. Hence, Intel's GPU architects require a faster alternative, or must otherwise forego traditional early-stage (pre-silicon) DSE that accounts for hardware enhancements in conjunction with software evolution. Our proposed solution to this conundrum is a framework that trains predictive regression models using a functional simulator that we modified to execute DirectX 3D rendering workloads and extend with a *workload characterization framework* (WCF) to produce model features. The models are trained to predict the performance of the GPU CAS that architects would prefer to use during pre-silicon design. Using a predictive model is several orders of magnitude faster than cycle-accurate simulation, while incurring an acceptable loss of accuracy. This increases the rate at which automated design tools can evaluate new points in the GPU architectural design space and increases the number of workloads that can tractably be used during for evaluation.

In addition to pre-silicon DSE, GPU hardware-software co-design tasks include: pre-silicon driver conformance and performance validation, evaluation of new microarchitectural units designed to accelerate latest generation API features, and performance evaluation of system level integration of the GPUs. The predictive modeling framework introduced in this paper can accelerate performance evaluation of many of these tasks as well. We further utilize the trained models to rank the metrics produced by the functional simulation to determine their relative impact on GPU performance, providing designers with intuition as to which micro-architectural subsystems are likely performance bottlenecks. These counters provide architectural information in the form of malleable and generic execution counts of API supported rendering tasks. This information is different than can be obtained from counters in commercial GPUs.

We evaluate the models' accuracy using a representative workload sample consisting of 369 frames collected from 24 DirectX 11 games and GPU benchmarking tools. Once a regression model has been trained, it can be more generally applied to a larger set of workloads used for DSE. Our best performing model, RF regression, achieves a respectable 14.34% average *out-of-sample-error*, while running a minimum 40.7x, maximum 1197.7x and average 327.8x faster than the pre-silicon CAS.

## 3.1 Rasterization-Based Modeling Framework

Fig. 3:1 depicts our pre-silicon predictive modeling framework goal; our evaluation focuses on Intel GPU architectures (specifically, the Skylake generation) using 3D DirectX 11 rendering workloads. The software rasterizer (RastSim) is a functional simulator configured to model the Skylake generation GPU architecture, to execute the workloads and is augmented to provide program counter measurements. These measurements are input into a model that predicts the performance that would be reported if we executed the workload on an internally validated GPU CAS, titled GPUSim, configured to model the same architecture.

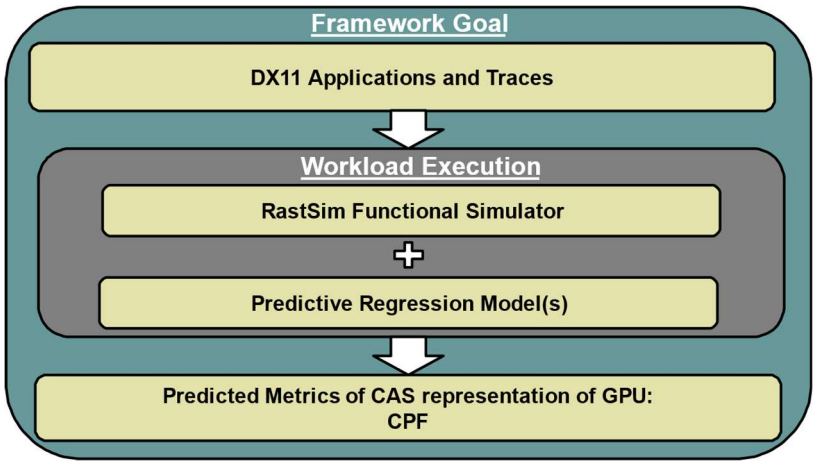


Figure 3:1 Rasterization-Based Model Goal
Modeling framework: a functional simulator executes 3D DirectX11 workloads. Performance counter measurements obtained from the RastSim simulator are used by predictive models to predict the performance of the cycle-accurate GPU simulator configured as newer GPU generation devices.

Both RastSim and GPUSim use vendor drivers to execute the rendering workloads. A new model is trained for each point in the GPU architectural design space. GPUSim is only used to collect the golden reference performance, which is used to train the model. Once the model has been trained, the design point can be characterized on a much larger set of evaluation workloads. Each evaluation workload executes on the functional simulator to collect performance counter measurements, which are then input to the model, which predicts the execution time of the workload at the current design point. Our results show that this is much faster than cycle accurate simulation and provides performance estimates that the functional simulator cannot provide on its own.

Fig. 3:2 shows that GPU performance ultimately depends on co-optimized hardware and software. Predictive modeling enables designers to perform co-optimization in earlier stages of the design process, allowing many more design points to be explored.

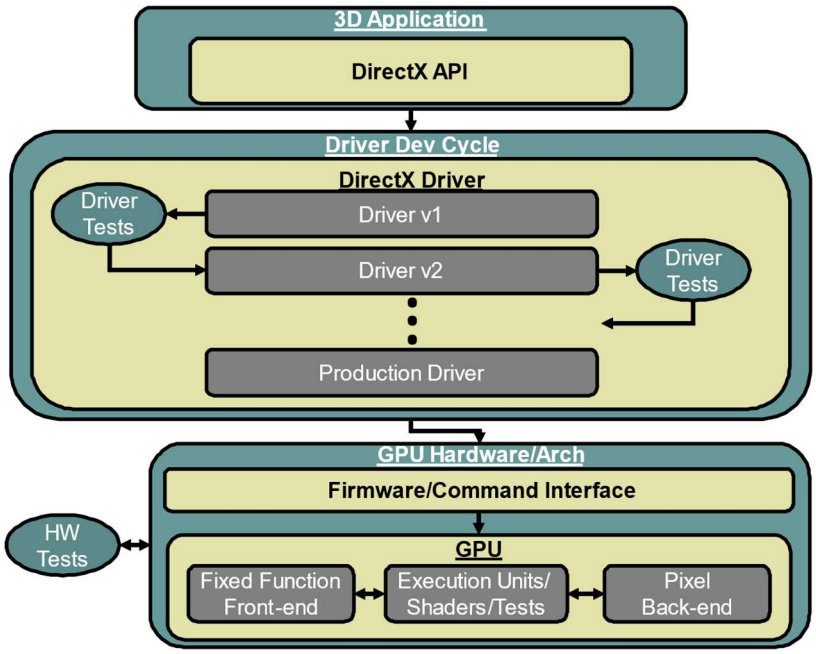


Figure 3:2 Example Design Usage of Rasterization Method

GPU Hardware and software co-optimization process being targeted for rapid performance estimation. Hardware and software are co-optimized in lock-step fashion, requiring repeated simulation in a traditional design environment.

### 3.1.1 The Graphics Workload Library

The GWL contains 369 frames collected from 24 DirectX 11 games and GPU benchmarking tools, as listed in Table 3:1. Although we collect multiple frames from each application, we treat each frame as a single workload due to long per-frame cycle-accurate simulation times. The GWL applications are input to the model training and validation process, which uses 10-fold CV as discussed in Section 4.3.

| Workload | Frame Count |
|---|---|
| 3DMark Fire Strike | 16 |
| 3DMark Sky Diver | 29 |
| 3DMark Fire Strike | 11 |
| 3DMark Ice Storm | 6 |
| Assassins Creed IV Black Flag | 7 |
| Assassins Creed Unity | 6 |
| Batman Arkham City | 41 |
| Batman Arkham Origins | 25 |
| Bioshock Infinite | 26 |
| Civilization 5 | 2 |
| Civilization Beyond Earth | 10 |
| Call of Duty Advanced Warfare | 9 |
| Crysis 3 | 15 |
| F1 2013 | 2 |
| Far Cry 4 | 6 |
| Metro Last Light | 19 |
| Middle-earth: Shadow of Mordor | 18 |
| Tom Clancy's Splinter Cell: Blacklist | 13 |
| Unigine Heaven Benchmark 4.0 | 56 |
| Unigine Valley | 23 |
| Volume Rendering | 8 |
| Voxel Based Global Illumination | 5 |
| The Witcher 3: Wild Hunt | 10 |
| World of Warcraft: Warlords of Draenor | 7 |

Table 3:1 Graphical Workload Library
24 3D DirectX 11 workloads, and the number of frames used from each workload.

### 3.1.2 Model Training and Validation Flow

Fig. 3:3 illustrates our model training and prediction flow using GWL workloads. A proprietary tool (*GfxCapture*) collects single-frame traces in two formats: (1) *SWTraces*, which consist of DirectX API commands collected pre-driver, which execute on RastSim; and (2) *HWTraces*, which consist of native GPU commands collected post-driver to execute on GPUSim. A subsequent proprietary application, *GfxPlayer*, streams the traces to RastSim, which collects and provides a set of performance counter measurements.

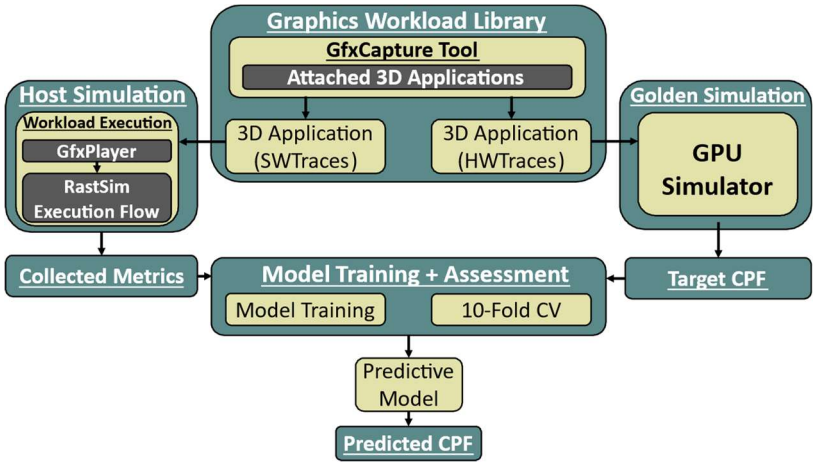


Figure 3:3 Rasterization Model Framework Details
Model training and validation requires workload execution on both RastSim and GPUSim.

Fig. 3:4 illustrates the modeling training and deployment (prediction) phases of Fig. 3:3. GWL applications are assembled to form a training set. Performance counter measurements provided by RastSim are used for model training. GPUSim executes the training workloads to provide performance measurements in terms of CPF; these golden reference values are used to train the model. We again use *10-fold CV* [13] to estimate *out-of-sample-error* ($E_{out}$) to validate the model. The model is used to predict the CPF of previously unseen workloads.

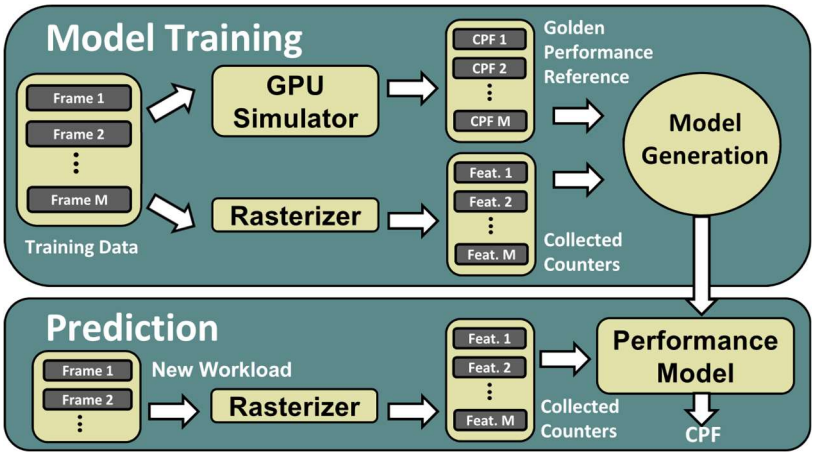


Figure 3:4 Rasterization-Based Model Train and Application
Model training (top) and prediction flow (bottom).

### 3.1.3 RastSim

RastSim is a proprietary extension to the OpenSWR [39] rasterizer, which is fully integrated into the Mesa 3D Graphics Library and normally targets the OpenGL API. As shown in Fig. 3:5, RastSim consists of two primary subsystems:

- the *RastSim Command Interface* and state tracker; and
- the *Rasterization Core*.

The Command Interface and state tracker are modified to ensure Intel GPU and DirectX API conformance, and are implemented as the external interface and internal control of the Rasterization Core, which executes functional simulation. The wrapper intercepts and issues commands from the API and drivers, providing the same interface to the software execution stack as the GPU hardware it replaces. It also maintains the necessary data structures to track render pipeline activity between architectural units and maintains GPU state during workload execution. RastSim has been extended with a WCF that has been integrated into Mesa3D as “archrast,” which instruments the Rasterization Core and Command Interface to track render pipeline behavior, instruction counts, and workload execution state.

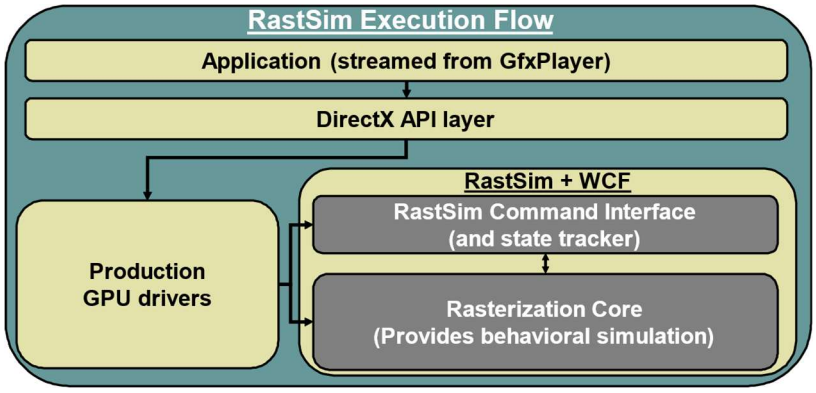


Figure 3:5 Rasterizer Execution Flow

RastSim utilizes GPU drivers, GfxPlayer and the DirectX API to provide a behavioral simulation of the Intel GPU DirectX pipeline to produce performance counters.

### 3.1.4 GPUSim

GPUSim is a proprietary CAS used for pre-silicon design studies. GPUSim models the GPU microarchitecture, memory subsystems, and DRAM, and has been validated internally when configured to model post-silicon GPUs. We use GPUSim to produce golden reference performance estimates for model training. To avoid disclosure of propriety information, CPF estimates produced by GPUSim are reported in normalized form.

## 3.2 Rasterization-based GPU Model

We configured RastSim and GPUSim to model a 2-slice Intel Skylake GPU (Fig. 3:6). While Skylake GT3 GPUs are commercially available, we do not predict the performance of the post-silicon device, because our objective is to mimic the GPU design process. Employing commercially available silicon mitigates confidentiality concerns, as pre-silicon GPUs may include features that cannot be disclosed publicly. Validated Skylake GT3 architectural models are available, which eliminates the need to tune the functional simulator and WCF to match an ever-evolving in-flight design.

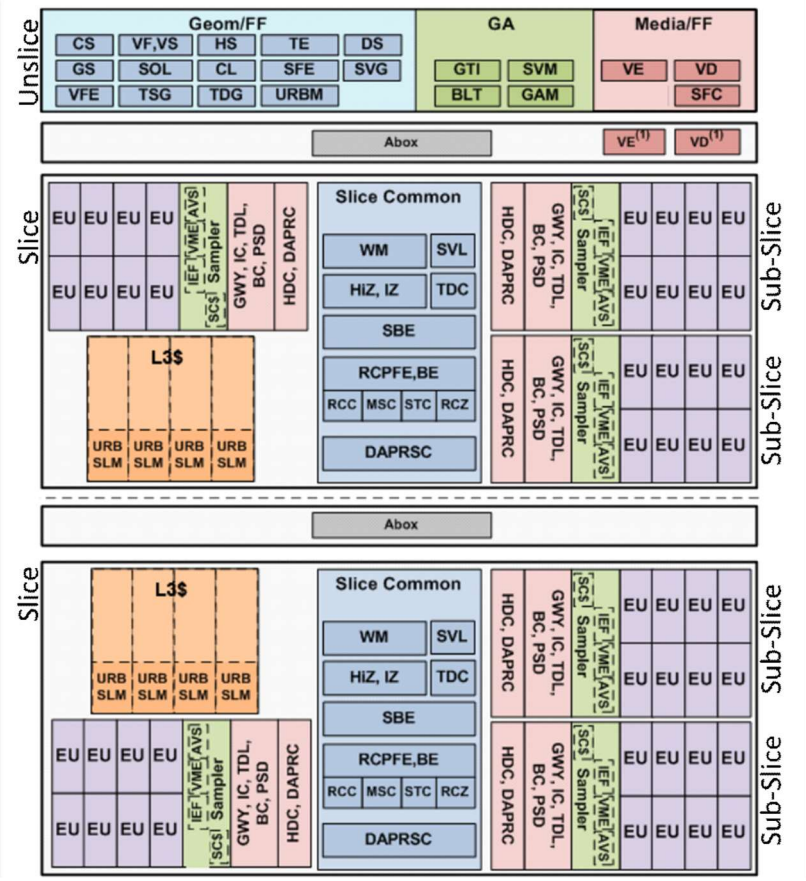


Figure 3:6 Intel Skylake GT3 Architecture
The Intel Skylake GT3 Architecture uses two GPU slices.

### 3.2.1 Unslice Architecture

The Unslice is the GPU front-end, consisting of Global Asset (GA) and dedicated render (Geom/FF) units. The GA units contain the GTI, which performs I/O, and the State Variable Manager (SVM), which holds execution state variables. RastSim does not simulate the GA units, replacing them with the Command Interface and State Tracker, which also provide a JIT-compiled Blitter (BLT) for speed, and Graphics Arbiter (GAM) model. The Command Interface and State Tracking Layer provide 8 counters, which measure draw, synchronization, and vertex count metrics. RastSim natively produces 77 3D state tracking counters (SVM) and 27 pipeline control-related counters (GAM).

The Geom/FF units accelerate DirectX features and programmable shader requirements. The FF units interface with the Slice, utilizing caches and EU clusters to accelerate programmable features and to create and dispatch threads. RastSim models only those units that directly execute on geometry: the Input Assembler (IA), the CS, the Vertex Fetch (VF) and VS units. It also models the three stages of DirectX 11 tessellation: the HS, Tesselator (TE), and DS units, along with the GS, Clipper (CL) and the Stream Output Logic (SOL). The WCF provides 15 counters to track these Unslice behaviors, as shown in Fig. 3:7.

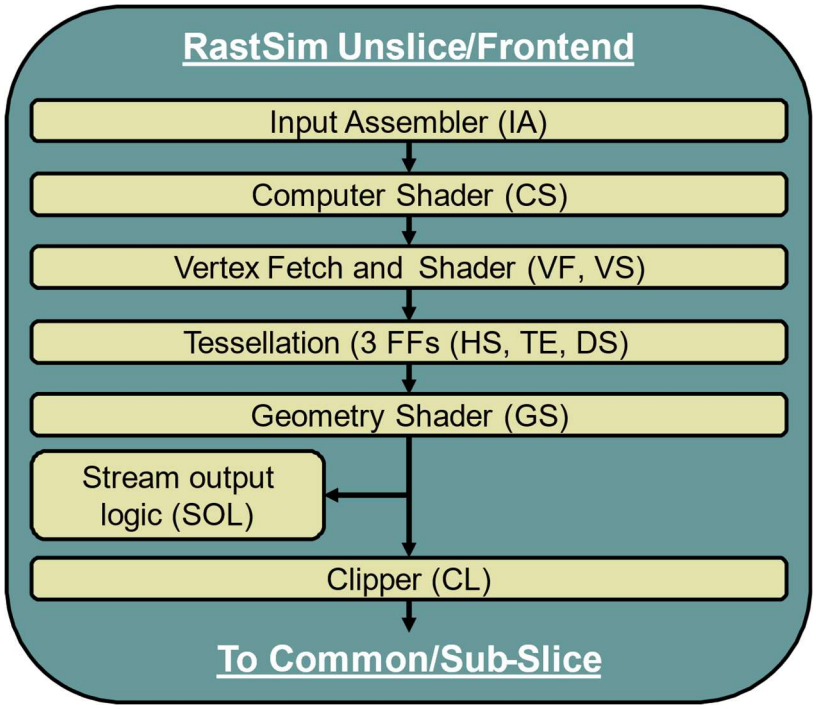


Figure 3:7 RastSim Unslice/Frontend
RastSim Models a limited number of the front end and unsliced units present on the GPU. This depicts the main components modeled.

### 3.2.2 Slice Architecture

Each slice is decomposed into three subgroups: (1) the *Slice Common* (Fig. 3:8) which provides additional FF architectural units; (2) the *Sub-Slice* (Fig. 3:9) which contains 24 EUs and supporting execution hardware; and (3) an L3 cache. RastSim models only the portions of the Slice

Common and Sub-Slice that are needed to provide functionally correct rendering. We target a 2-slice Skylake GT3 GPU.

#### 3.2.2.1 Slice Common and L3 Cache.

The Slice Common FF units support the front-end and Sub-Slice units; these include the Windower (WM) which performs rasterization, the Hierarchical-Z (HIZ), and Intermediate-Z (IZ) units, which perform Depth (Z) and stencil testing, and a host of caches used for differing portions of the pipeline. As shown in Fig. 3:8, RastSim models only those components necessary to provide functional equivalence at render output, omitting detailed modeling of the caches and HIZ and IZ units. The WCF produces 28 counters that capture metrics relating to Alpha, Early-Z, and Early Stencil tests.

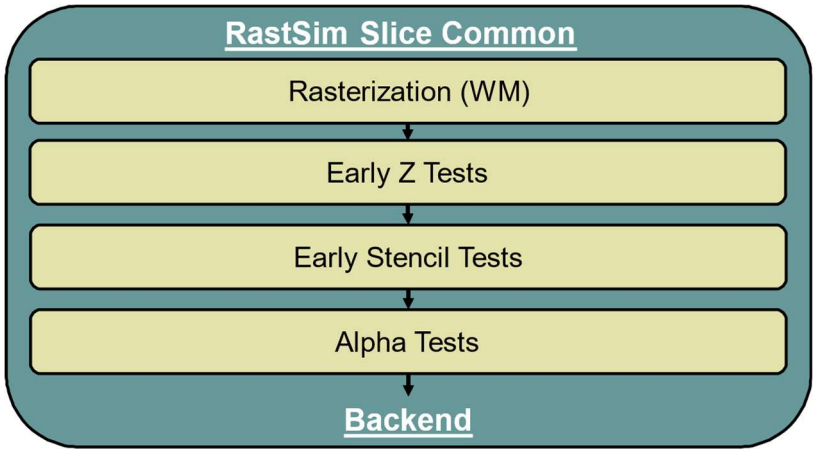


Figure 3:8 RastSim Slice Common
Slice Common units modeled by RastSim. RastSim does not model caches, TD logic, or dedicated media units.

#### 3.2.2.2 Sub-slice and EU clusters

The Sub-Slice and render pipeline back-end consist of an EU array, and supporting fixed and shared function units, such as the sampler, EU IC, Local EU TD logic, Data Cluster (HDC), render cache (DAPRC) a Pixel Shader dispatcher (PSD), and a Barycentric Calculator (BC). As shown in Fig. 3:9, RastSim models programmable units such as the PSD, PS, and the BC, along with late-stage Z- and stencil testing, blend shading, output results merging, and writes to the GPU

render target (Viewport); RastSim does not simulate TD and EU execution, nor model the IC, TDL, HDC, or DAPRC. The WCF provides 18 counters to track PS behavior, depth/stencil tests, and render target write metrics.

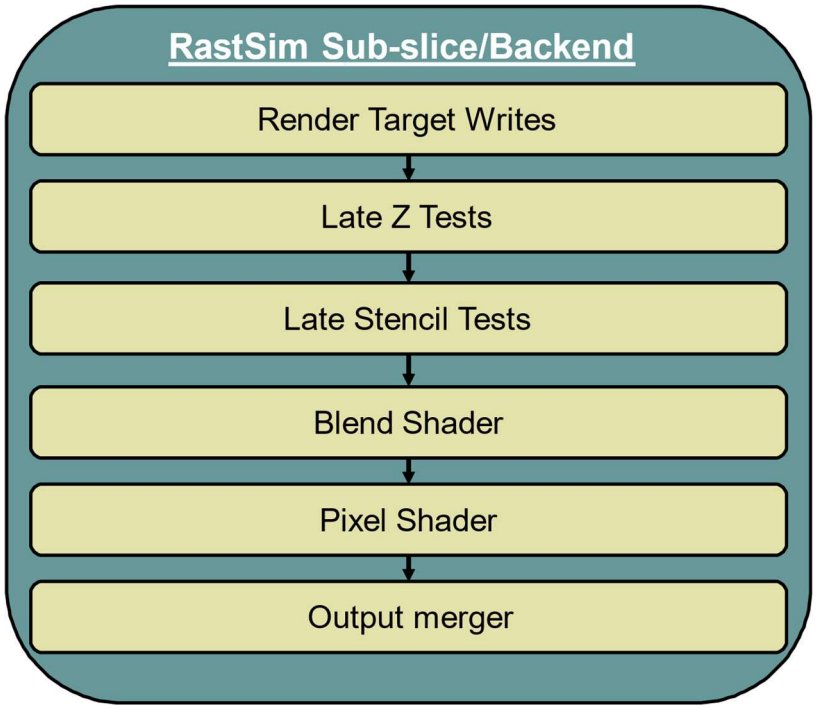


Figure 3:9 RastSim Sub-Slice and Backend
Subslice units and output stages modeled by the Rasterizer.

## 3.3 Rasterization-Based Regression Modeling Framework

We again employ a non-linear RF regression model [36] to estimate pre-silicon GPU performance, which proved to be most accurate for the Rasterization based modeling approach. Our model building procedure also produces and evaluates 14 linear regression models, which are used as a baseline for comparison. When compared to the HALWPE estimation-based approach, we have further added the Elastic-Net and Elastic-Net with NNLS models to account for L2 regularization effects in addition to the L1 regularization provided by Lasso. The choice to train an ensemble of models is motivated by the fact that both the correlation between CPF and model

features and the degree of linearity between program counters and target CPF are unknown in advance; moreover, it was not initially clear that RF would emerge as the most accurate model. Fig. 3:10 depicts the ensemble of models that were trained; readers unfamiliar with the underlying statistical concepts described are encouraged to consult Ref. [32].

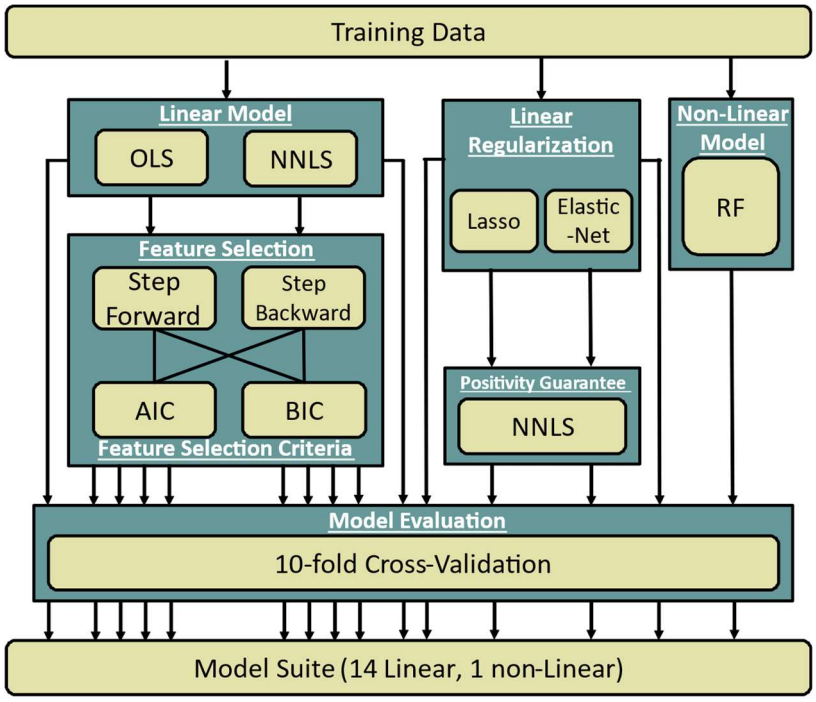


Figure 3:10 RastSim Model Creation Flow
14 linear models and one non-linear model is created for each model building task. Elastic-net is newly added.

### 3.3.1 Regression Model Categories

We generate 14 linear regression models, and one non-linear model that are placed into 5 modeling categories: **Category 1**, OLS, and **Category 2**, NNLS contain 10 models, 5 from each category. Category 1 includes the full regression OLS and category 2 the full NNLS model. The remaining 8 models perform feature selection utilizing *forward* (*Fwd*) and *backward* (*Bwd*) *stepwise selection*. We apply the AIC [1] and the BIC [26] to the stepwise methods, yielding 4 models: {Fwd, Bwd}×{AIC, BIC}, which are applied to OLS and NNLS. **Category 3,** Regularization and Category 4, Regularization-NNLS each contain 2 models. Category 3 contains

the Lasso [35] and Elastic-Net [40], which perform feature selection during model building. **Category 4** augments the Lasso and Elastic-net models with the NNLS requirement. **Category 5** contains our one non-linear model, RF [7], which turned out to be the most accurate model that we generated. For this reason, the discussion that follows emphasizes RF.

In the RastSim based iteration of the modelling framework, the input to each model is a set of program counters (a feature vector) collected from RastSim, $X = [x_1, x_2, \ldots, x_N]$. We produce $M$ feature vectors, from $M$ workloads. Each model produces a set of $M$ outputs, the responses, in the form of CPF, one for each workload. Each of the models' input data sets, consists of the same 369 workloads, containing 105 independent variables provided by RastSim natively, and an additional 69 program counters delivered by the RastSim workload characterizer.

### 3.3.2 Elastic-Net Regression Model

Of those models presented and discussed in Chapter 2 section 3 only the Elastic-Net is newly added. Lasso and OLS are not optimized to identify correlated features. We introduce Elastic-net to perform correlated feature selection. Elastic-net chooses a representative feature to include from a larger group of correlated features, removing the others to improve model accuracy and avoid redundant features. Elastic-Net is a hybrid method that bridges traditional Ridge Regression [41] and Lasso to perform pairwise feature correlation aware feature selection [40].

$$\text{RSS}(\beta^{Elastic-Net}) = argmin_{\beta,\beta 0 \geq 0} \frac{1}{N} \sum_{i=1}^{M} || \left(y_i - f(X_i)\right) ||_2^2 + \alpha_1^T \beta$$

Subject to:

$$\sum_{j=1}^{N} \left(\alpha \beta_j^2 + (1-\alpha)|\beta_j|\right) < T$$

Equation 3:1 – Elastic-Net Regularization RSS

*T* is the penalty term in (4), and the parameter $\alpha \in [0,1]$ serves as a correlation sensitivity knob. Setting $\alpha = 1$ assumes maximum correlation (ridge), and $\alpha = 0$ assumes none (lasso). We use $\alpha=0.5$.

### 3.3.3 Random Forest Regression Model

As previously discussed in Chapter 2 section 3, RF is a non-linear model ensemble regression method. In this section, we discuss our most accurate model, the RF model building process, and how the final prediction is arrived at from the underlying forest of decision trees. RF is consistently the most accurate model for RF is an ensemble method, which aggregates the predictions of a collection of regression trees [42]. RF is based on the observation that regression trees exhibit high-variance and low bias when grown sufficiently deep. Prior work has shown that bootstrap sampling [32] of training data can effectively minimize correlation between the regression trees comprising the forest; averaging the predicted CPF produced by the trees further reduces variance while maintaining low bias.

An RF model is created by creating *n* trees, each of which is grown on a bootstrap sampled data set *D*. Tree growth is achieved by a recursive process which randomly selects *M* variables from the original set of features, with replacement. Sampling with replacement is the process of replacing the originally sampled variables with the new variables chosen in subsequent sampling steps. This means that variables are not held out of subsequent rounds, ensuring that: (1) each variable is equally likely to be chosen during each round; and (2) the covariance between sets of sampled variables is 0, i.e. each sample is independent of the others. Utilizing the M variables chosen by sampling with replacement as candidates, we select the prime candidate to perform a split.

A split is performed by observing each variable $m_i \in M$, and determining the range of observable values in *D*. For each variable and range, we then choose the best value within that

range and treat it as a binary splitting point S, which is represented by a node in a tree. After selecting S, two daughter nodes (*left*, *right*) are created and assigned the parent node S, whereby each data point in D that has value ≤ S is assigned to the *left* sub-tree, $D_{left}$, and the remaining assigned to the *right*, subtree, $D_{right}$. The value S chosen as the split point is chosen by computing the RSS Error for all response variables at all split values considered. The value chosen for splitting is the one that minimizes RSS error.

For RF regression, RSS is computed as follows [43]:

$$RSS(RFSplit) = \sum_{i=0}^{|D_{left}|} (\mathrm{y_i} - \mathrm{y_L})^2 \; + \sum_{i=0}^{|D_{right}|} (\mathrm{y_i} - \mathrm{y_R})^2$$

Equation 3:2 – RF Regression RSS

where $y_i$ is the current CPF prediction of workload $i \in \mathrm{D_{left/right}}$, $y_L$ is the average true CPF value for all workloads $i \in \mathrm{D_{left}}$, and $y_R$ is the average true CPF value for all workloads $i \in \mathrm{D_{right}}$

After growing all *n* trees, we form an ensemble $RF = \bigcup_{j=1}^{n} T_j$. The CPF prediction of workload $m_i$ can then be computed by computing the mean of each tree's CPF prediction for $m_i$ as follows:

$$RF(m_i) = \frac{1}{n} \sum_{j=1}^{n} T_j(m_i)$$

Equation 3:3 – RF Predictive Mean

where $T_j(m_i)$ is the predicted CPF of $T_j$ when applied to the RastSim performance counters obtained from simulation of workload $m_i$.

### 3.3.4 Model Evaluation

We again use 10-fold cross validation [13] to report the $\mathrm{E_{out}}$, the mean absolute percentage error averaged over all ten folds, as our primary measure for model accuracy. We also evaluate models in terms of their IRs. We report 10 % and 20 % IRs for each model we produce and compare IRs across varying thresholds for comparative analysis of the prediction scenarios.

### 3.3.5 RF Parameter Optimization

RF has several parameters that must be chosen to reduce prediction error. Typically, RF works well with relatively little tuning. We utilize the *RandomForest* package from CRAN [21], and the default parameter settings, excluding the number of trees ($n$). We repeatedly fit RF models with $n = 2^i,\ 1 \leq i \leq 10$, trees, and select the value of $n$ that minimizes $E_{out}$.

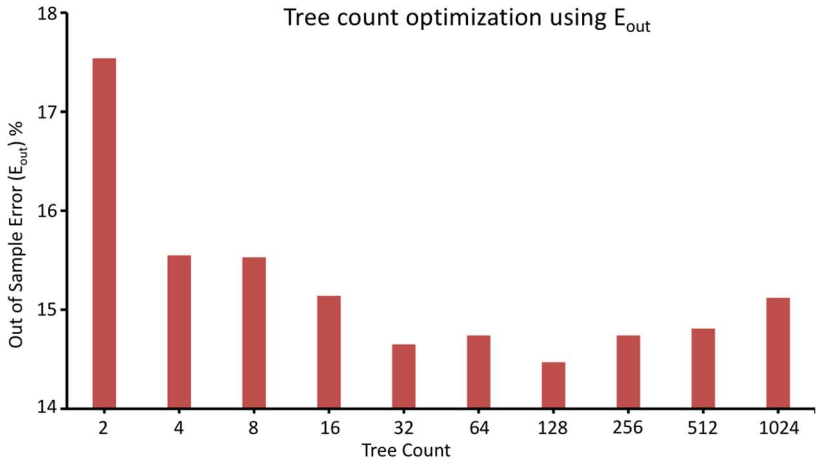


Figure 3:11 RF Tree Count Impact on $E_{out}$.
Quantifying the impact of the number of trees in the RF model on $E_{out}$; lower values are better.

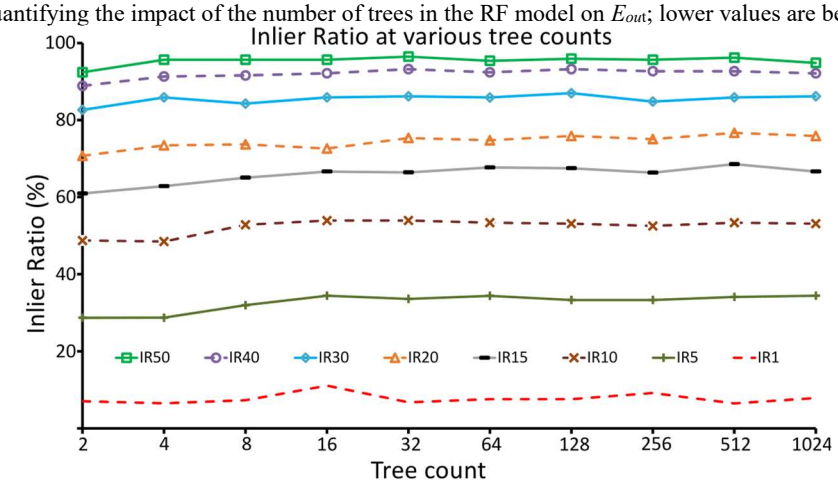


Figure 3:12 RF Tree Count Impact on $I_R$.
Quantifying the impact of the number of trees in the RF model on $I_R$ at varying thresholds; higher values are better.

## 3.4 Rasterization-Based Model Results

We configured RastSim and GPUSim to model a Skylake GT3 GPU operating at 1155 MHz (GPUSim). Performance counter readings for the GWL workloads (Table 3:1) produced by

RastSim were used to train and validate the 15 regression models, as discussed in the preceding section. We produce two sets of results:

The 5 best performing models obtained using the performance counters that we added to RastSim through the WCF (Table 3:2), including the $I_R$ at various thresholds (Fig. 3:14). The 5 performing models trained exclusively using performance counters originally available in RastSim (Table 3:3), including the $I_R$ at various thresholds (Fig. 3:15).

### 3.4.1 Predictive Model Results

Table 3:2 clearly indicates that RF is the best performing model, achieving an $E_{out}$ of 14.34%, a 5.75% improvement over the second-best model, the Elastic-net; this error rate is sufficiently low for use in early-stage DSE; GPUSim is still required for detailed performance characterization and post-silicon performance validation.

Fig. 3:13 reports the IRs at 8 different threshold values $\boldsymbol{T} \in \{\mathbf{50}\%, \mathbf{40}\%, \mathbf{30}\%, \mathbf{20}\%, \mathbf{15}\%, \mathbf{10}\%, \mathbf{5}\%, \mathbf{1}\%\}$. RF and Elastic-net achieve the highest IRs at each data point, with RF retaining a minor advantage at all threshold values other than 20%, where Elastic-net is 0.58% higher. These results indicate that RF is without question the best performing model.

| Model Category | Best Performing Model | #Wklds | #Feat | $E_{in}$ % | $E_{out}$ % |
|---|---|---|---|---|---|
| **Non-Linear** | **Random Forest** | **369** | **174** | **5.76** | **14.34** |
| OLS | OLS/FWD/AIC | 369 | 60 | 18.43 | 21.86 |
| NNLS | NNLS/BWD/AIC | 369 | 17 | 27.05 | 28.34 |
| Regularization | Elastic | 369 | 163 | 20.08 | 20.09 |
| Regularization NNLS | Lasso/NNLS | 369 | 12 | 35.32 | 34.35 |

Table 3:2 RastSim 5 Best Performing Models
Comparison of the best model errors from each category using the RastSim workload characterization extensions.

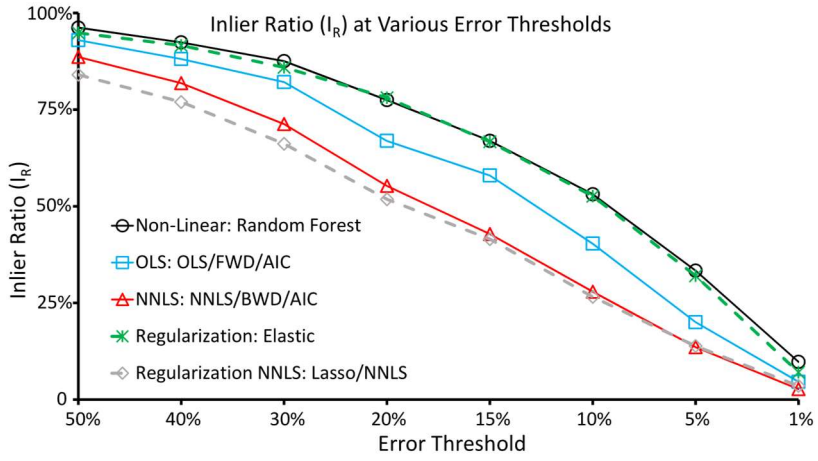


Figure 3:13 RastSim Best Model $I_R$ Percentages with WCF.
Skylake Inlier rates at various error thresholds using the RastSim workload characterization extensions.

### 3.4.2 WCF Impact in RastSim

Table 3:3 and Fig. 3:14 report the results of a similar experiment performed using only the performance counters available natively in RastSim, prior to the introduction of the WCF which introduced many additional counters. The best performing model, once again, is RF, although its $E_{out}$ jumps to 52.34% (Table 3:3); this justifies the introduction of the WCF and its additional performance counters for this RastSim use case.

Fig. 3:14 reports the IRs at the same threshold values as Fig. 3:14. Once again, RF and Elastic-net achieve the highest IRs, although they are much lower than the results reported in Fig. 3:14. For example, RF achieves a 29.03% $I_R$ at the 20% threshold, and a 15.47% $I_R$ at the 10% threshold, which once again testifies to the inaccuracy of our model without the additional performance counters provided by the WCF. This level of degradation in model accuracy indicates that the native RastSim counters do not correlate with GPU performance (CPF).

| Model Category | Best Performing Model | #Wklds | #Feat | $E_{in}$ % | $E_{out}$ % |
|---|---|---|---|---|---|
| **Non-Linear** | **Random Forest** | **369** | **105** | **18.29** | **52.34** |
| OLS | OLS/BWD/BIC | 369 | 22 | 74.27 | 78.41 |
| NNLS | Full | 369 | 13 | 106.3 | 108.32 |
| Regularization | Elastic | 369 | 105 | 75.39 | 75.41 |
| Regularization NNLS | Lasso/NNLS | 369 | 8 | 105.13 | 105.85 |

Table 3:3 RastSim 5 Best Performing Models
Comparison of the best model errors from each category without the RastSim workload characterization extensions.

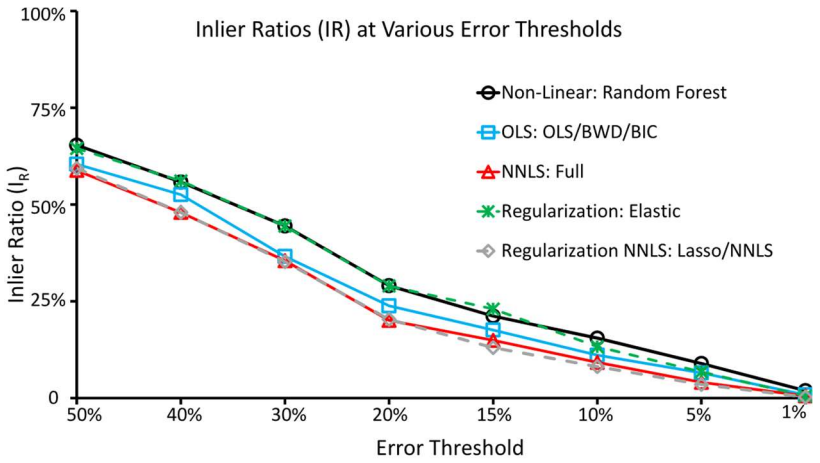


Figure 3:14 RastSim Best Model $I_R$ Percentages without WCF.
Skylake Inlier rates at various error thresholds without the RastSim workload characterization extensions.

### 3.4.3 Relative Accuracy Preservation

Fig. 3:1**5** reports the predicted and observed CPF (both normalized) for each trace, along with its APE, for the RF model built using WCF performance counters (Table 3:2). The observed CPF was obtained by cycle-accurate simulation (GPUSim), as was used as the golden reference model for predictive model training. With the data points reported in increasing order of observed CPF, we observe that the predicted CPF ordering is similar for most traces, with a handful of exceptions as observed CPF grows large. These disparities indicate that RastSim and WCF performance counters lack some key features that strongly correlate to CPF. A significant percentage of GPU execution time is spent on programmable shaders and threads in the Sub-Slice

EU clusters: GPUSim reported high EU active and stall times. These performance counter values were much higher for the workloads that exhibited large disparities between predicted and observed CPF.

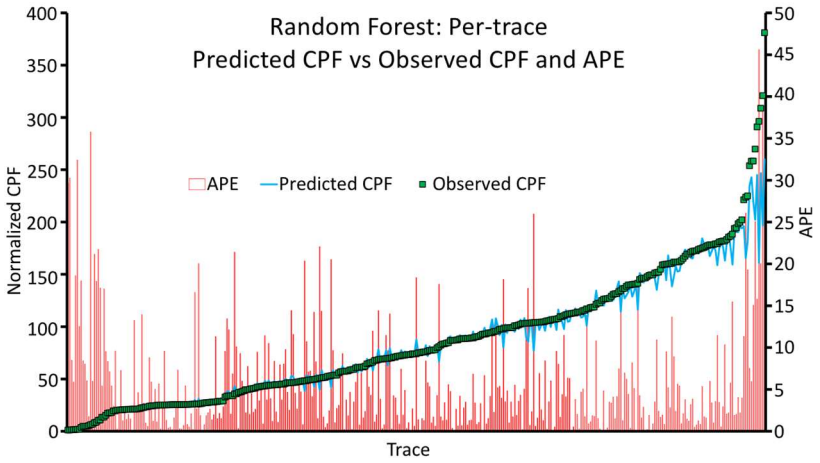


Figure 3:15 RastSim Relative Accuracy Preservation.
Predicted and observed CPF and APE for each GWL workload using the RF model (Table 3:2). Data points are sorted in non-decreasing order of observed CPF.

To capture this information, it is possible to extend RastSim to model TD and EU activity; however, this would introduce cycle-accurate simulation to RastSim, slowing it down significantly. It is clear talking to other internal RastSim users that increased execution time would degrade its other pre-silicon use cases. RastSim with the WCF, as presently constituted, strikes a good balance between preserving applicability to other uses cases and achieving accurate performance prediction.

### 3.4.4 Predictive Model Speedup

The motivation for predictive modeling is to obtain workload performance estimates faster than cycle-accurate simulation. Fig. 3:16 reports the speedup of RastSim functional simulation (including WCF overhead) and predictive model deployment compared to GPUSim cycle-accurate simulation for each workload; these results do not account for model training time, which is performed offline. Observed speedups ranged from 40.7x to 1197.7x, with an average of

327.8x and a standard deviation of 196.62. These speedups are sufficient to enable internal usage of RastSim for early-stage GPU architectural DSE. It is also worth noting that the WCF causes an average slowdown of 3x compared to native RastSim execution; the speedups included in Fig. 3:17 include the WCH overhead.

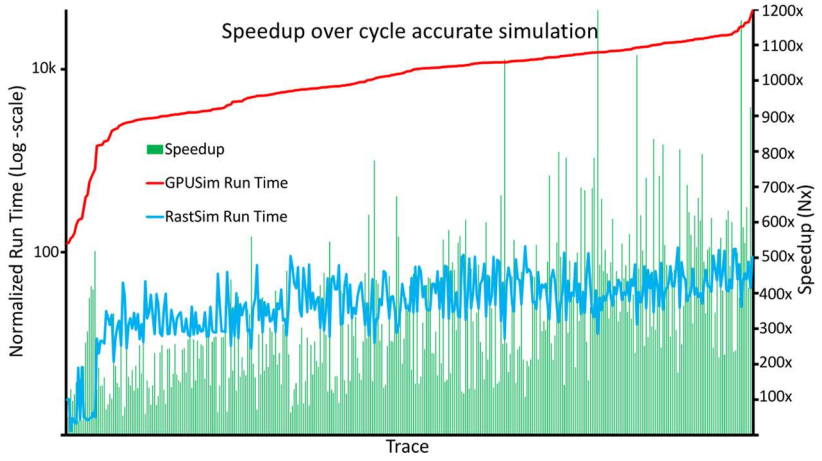


Figure 3:16 RastSim Based Modeling Speedup.
Normalized RastSim and GPUSim execution time and speedup for each GWL workload. Data points are ordered in non-decreasing order of GPUSim execution time.

### 3.4.5 RF Feature Ranking

The relative importance of counters in an RF regression model can be obtained by measuring the impact of each feature on the model's predictive capability. The impact is measured by summing each variable's RSS error, computed in Eq. (1), when it is used as a split point, for all trees in the forest for which it was selected [43]. Table 3:4 reports the 20 program counters ranked as being most important to the RF model using this approach; 18 of the counters are part of the WCF, while the remaining 2 are native to RastSim.

| Feature Name (Description) | Feature Category | RSS Ranking |
|---|---|---|
| **Pixel Written** (The number of pixels written to render target) | Pixel Backend | 20.39 |
| **Sub-spans Written** (The number of Sub-spans written to render target) | Pixel Backend | 19.73 |
| **Samples Written** (Number of samples written to render target) | Pixel Backend | 19.5 |
| **Passed Early Z Test** (The number of passed Early Z tests in output merger stage of render pipeline) | Vertex Testing | 15.73 |
| **Stencil Tested Subspans** (The number of stencils tests performed on sub-spans) | Vertex Testing | 14.61 |
| **Early Z Tests** (The total number of early Z tests performed) | Vertex Testing | 14.50 |
| **Passed Early Stencil Test** (The number of early stencil tests that passed in output merger stage) | Vertex Testing | 13.17 |
| **Early Stencil Tests** (The total number of early stencil tests performed) | Vertex Testing | 12.55 |
| **Passed Depth Tests** (The Number of pixels passing depth tests) | Pixel Testing | 12.01 |
| **Sub-span Z Tests** (The number of Z tests performed on sub-spans) | Vertex Testing | 11.87 |
| **Passing SubSpans** (The total number of SubSpans that pass all tests and reach the Pixel shader stage) | Vertex Testing | 11.2 |
| **Pixel Shader Invocations** (The number of times the Pixel Shader is invoked) | Pixel Shader | 9.74 |
| **Passed Early Z Single Sample** (The number of passing early Z tests performed on single sample configured vertices) | Vertex Testing | 5.53 |
| **Tested Early Z Single Sample** (The number of early Z tests performed on single sample configured vertices) | Vertex Testing | 5.31 |
| **Tested Earl Stencil Single Sample** (The number of early stencil tests performed on single sample configured vertices) | Vertex Testing | 3.90 |
| **Passed Early Stencil Single Sample** (The number of passing early stencil tests performed on single sample configured vertices) | Vertex Testing | 3.85 |
| **Samples Killed** (The number of samples killed by front end and backend tests) | Kill Count | 2.87 |
| **Stream Out Invocations** (Invocation count of Front End Fixed Function Stream out stage) | Fixed Function | 2.68 |
| **Raterizer Count** (Rasterizer Unit invocation count) | Common Core | 2.61 |
| **Vertex Fetch Instancing** (The number of Vertices that are fetched in geometry instancing mode) | Fixed Function | 2.59 |

Table 3:4 RastSim Top 20 RF Ranked Features

The 20 highest ranked performance counters in the RF model based on RSS ranking.

The 3 highest-ranked counters track the number of times pixels are written to the render target, each indicating a different method of pixel grouping. The most important measure was the total pixel write count, whose lower bound (for one frame) is the monitor resolution. In contrast, the number of pixels written to the render target also includes pixel writes which are later overwritten by objects that share the same space; the final viewable object is determined by depth and stencil testing.

The next 5 highest-ranked performance counters relate to depth and stencil tests performed on vertices in the GPU front-end. They closely track the number of vertices that are tested and passed, which approximates the number of vertices that survive the HIZ and IZ tests and are then rasterized by the WM.

The top-8 ranked counters suggest that performance is dominated by the GPU compute activity that determines the final number of pixels, including identification of the number vertices that pass early Z and stencil tests, and are subsequently converted to pixel space via rasterization in the WM, and ultimately pass the late depth and stencil tests.

The 9$^{th}$ most important performance counter also tracks depth tests, this time the number of pixels tested in the RastSim back-end. Six of the remaining counters (Sub-Span Z Tests, Passing Sub-Spans, Passed Early Z Single Sample, Tested Early Z Single Sample, Tested Early Stencil Single Sample, and Passed Early Stencil Single Sample) track additional depth and stencil tests. Each of these counters indicates a different number of vertices tested, as indicated by the grouping into single-samples, sub-samples, and sub-spans. Only the 17$^{th}$ ranked feature in Table 3:4 (Samples Killed) is indicative of work *avoided* in late pipeline stages.

Two front end FF unit counters (Stream Out Invocations and VF Instancing), ranked 18$^{th}$ and 20$^{th}$, track the number of times the stream out FF unit is used. Stream Out Invocations tracks the last stage in the Geom/FF units in the GPU unsliced pipeline, while VF Instancing refers to the

first stage in the render pipeline during instancing mode. Rasterizer Counter, ranked 19$^{th}$, counts the number of times vertices were converted from vector graphics to raster/pixel format by the WM, tracking the work flowing from the GPU front- to back-end.

In summary, Table 3:4 indicates that the most important performance counters for CPF prediction were chosen from all portions of the render pipeline, and many of them measure the number of vertices that were tested and passed front-end depth and stencil tests. This indicates that these subsystems have the greatest impact on GPU performance, and should be slated for further study and optimization by architects.

# Chapter 4 Cross-Platform Prediction for FPGA High-Level Synthesis

The design and implementation of FPGA accelerators is difficult and time consuming. HLS tools [6, 7], can significantly boost developer productivity compared to writing RTL, though developers must still properly optimize performance and throughput using design pragmas (pipelining, loop unrolling, array partitioning). Finding the right HLS parameter settings is a complex problem. Synthesizing, placing, and routing each design point to characterize its performance by direct execution would be ideal, the amount of time required is untenable.

To address this need, we present *HLSPredict* [10], a cross-platform machine learning framework that can predict FPGA performance and power consumption using program counter measurements obtained from direct execution of a workload on a commercially available off-the-shelf host CPU executing sequential C/C++ code.

Using HLSPredict, a designer can rapidly ascertain if the time required to port a sequential application to an FPGA using HLS will yield sufficient improvements in performance and/or power to justify the design effort. Our experiments show that HLSPredict is 36.24x faster than HLS for baseline (unoptimized) workloads, and 43.78x faster than HLS for workloads that were optimized by appropriately setting pragmas, with error rates averaging 9.8% for performance and 7.8% for power prediction, respectively. Fig. 4:1 illustrates the high-level goal of HLSPredict and Fig. 4:2 provides a motivating example of a HLSPredict use case.

This paper makes the following technical contributions:

**HLSPredict leverages machine learning techniques:** we create a suite of 10 models that accurately predict FPGA performance (cycle counts) and pre-RTL power (in Watts) using CPU counters as features.

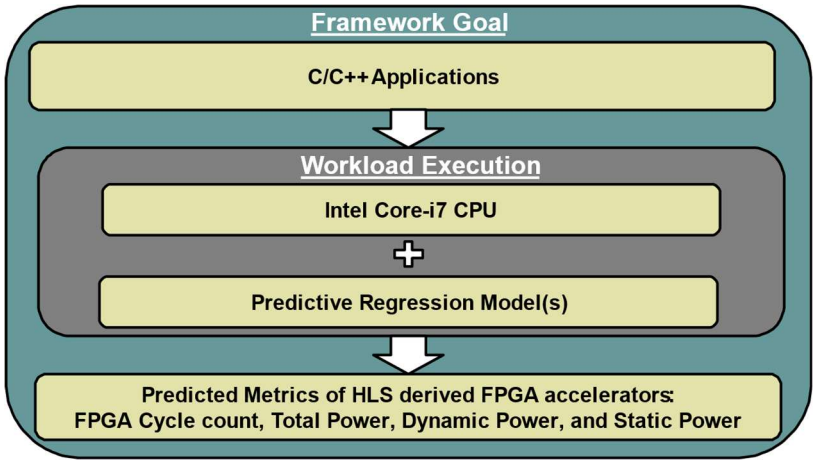


Figure 4:1 HLSPredict Model Goal

A collection of polyhedral C/C++ applications are executed on a host Intel Core-I7 CPU is used to execute C/C++ applications. Performance counter measurements obtained from the host CPU are used by predictive models to predict the performance and power of the HLS derived custom FPGA accelerators.

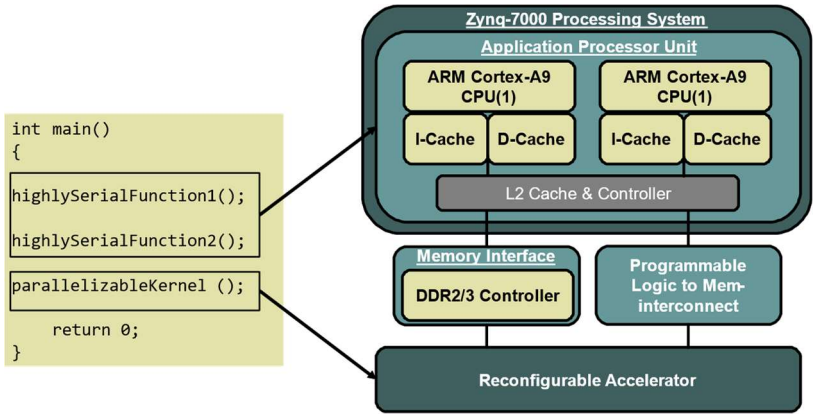


Figure 4:2 Hybrid CPU/FPGA Compute Example.

Hybrid CPU/FPGA compute model. The best platform choice is workload dependent and is not easily determined. HLSPredict guides developers to the best choice.

**HLSPredict is generalizable**: we create two classes of predictive models. The first-class targets unoptimized (baseline) HLS solutions; the second targets optimized solutions, which we created by using loop unrolling, array partitioning and pipelining directives in Vivado HLS. In both scenarios, HLSPredict obtains accurate estimates.

**HLSPredict ensures time synchronicity between the host CPU and the target FPGA accelerator by using sub-traces for model training**: sub-traces are epochs of workload execution time in the form of CPU counter measures for the host, and FPGA cycle-counts for the target.

**HLSPredict identifies the CPU microarchitectural subsystems that best predict FPGA performance and power:** we rank model features using a model-appropriate statistically rigorous approach.

The paper is organized as follows. Section 4.1 introduces the modeling framework, model training, and sub-trace generation methods. Sections 4.2-4.5 present the FPGA accelerator design, regression models, experimental methodology, and results. Section 4.6 places HLSPredict in the larger context of related work on predictive modeling, and Section 4.7 concludes the paper and outlines avenues for future research on this topic.

## 4.1 HLSPredict Modeling Framework

Fig. 4:3 illustrates the HLSPredict modeling framework. The *Workload Library (WL)* is a collection of the 30 Polybench/C 4.1 workloads [44], and 2 Polybench/GPU convolution workloads [45] manually ported to C; we list them in Table 4:1.

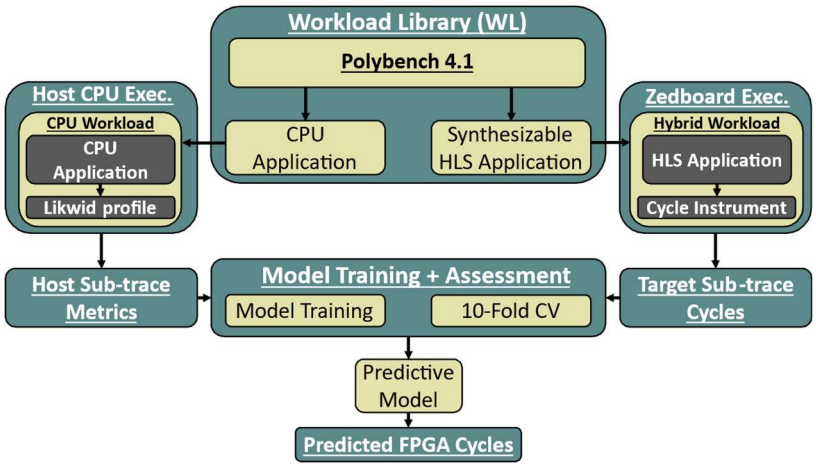


Figure 4:3 HLSPredict Prediction Framework

FPGA Perf prediction framework: (1) Traces are collected and stored in the WL. (2) Workloads execute on the desktop CPU host, and hybrid CPU/FPGA platform. (3) Performance counter measurements are collected, a model trained and used to predict an FPGA target metric. A complementary version of this framework exists for power prediction as well.

Each workload has two versions: one for execution on the host CPU and one for HLS synthesis, and several workloads are optimized (highlighted in yellow in Table 4:1 to improve performance over the baseline HLS implementation; the optimized workloads have an average speed up of 14.49x over the baseline. Table 4:1 also lists the average speedup per workload, for those that were amenable to optimization. All workloads were instrumented: the baseline workloads yielded 3941 sub-traces and the optimized workloads yielded 2018 sub-traces.

| Workload | Category | # Sub-traces | Avg. Speedup |
|---|---|---|---|
| 2mm | linear-algebra: kernels | 128 | 3.094x |
| 3mm | linear-algebra: kernels | 192 | 22.45x |
| adi | stencil | 128 | NA |
| atax | linear-algebra: kernels | 64 | 4.43x |
| bicg | linear-algebra: kernels | 64 | 2.77x |
| cholesky | linear-algebra: solvers | 128 | NA |
| conv2d | convolution | 124 | NA |
| conv3d | convolution | 102 | 9.33x |
| correlation | data mining | 106 | NA |
| covariance | data mining | 114 | NA |
| deriche | medley | 126 | NA |
| doitgen | linear-algebra: kernels | 112 | 56.70x |
| durbin | linear-algebra: solvers | 127 | NA |
| fdtd 2d | stencil | 100 | NA |
| floyd warshall | medley | 128 | 1.34x |
| gemm | linear-algebra: blas | 128 | 79.01 |
| gemver | linear-algebra: blas | 104 | 6.04x |
| gesummv | linear-algebra: blas | 128 | 2.54x |
| gramschmidt | linear-algebra: solvers | 128 | 4.34x |
| heat 3d | stencil | 100 | 7.42x |
| jacobi 1d | stencil | 128 | 9.17x |
| jacobi 2d | stencil | 128 | 8.86x |
| lu | linear-algebra: solvers | 128 | NA |
| ludcmp | linear-algebra: solvers | 86 | NA |
| mvt | linear-algebra: kernels | 128 | 2.73x |
| nussinov | medley | 127 | NA |
| seidel-2d | stencil | 128 | NA |
| symm | linear-algebra: blas | 127 | NA |
| syr2k | linear-algebra: blas | 128 | 12.97x |
| syrk | linear-algebra: blas | 128 | 13.10x |
| trisolv | linear-algebra: solvers | 128 | NA |
| trmm | linear-algebra: blas | 192 | NA |

Table 4:1 HLSPredict Workloads/Accelerators
Workloads, their sub-trace counts, and the average speedups obtained by optimizing workloads on the FPGA (yellow).

We instrument the host workload and execute it on the CPU to generate performance counter measurements using pre-configured *likwid-perfctr* (a performance counter API) event groups, one read per sub-trace [46, 47]. We then synthesize the same workload using Vivado HLS and execute it on the FPGA to obtain the target performance (FPGA cycle count) and power measures (total, dynamic, and static), one read per sub-trace.

The next step is to train a model to predict FPGA cycle counts and power consumption using CPU counters as features. Given a trained model, we can then execute previously unseen workloads on the host CPU to obtain performance counter readings and apply the model to rapidly obtain pre-RTL estimates of FPGA accelerator cycle counts and power.

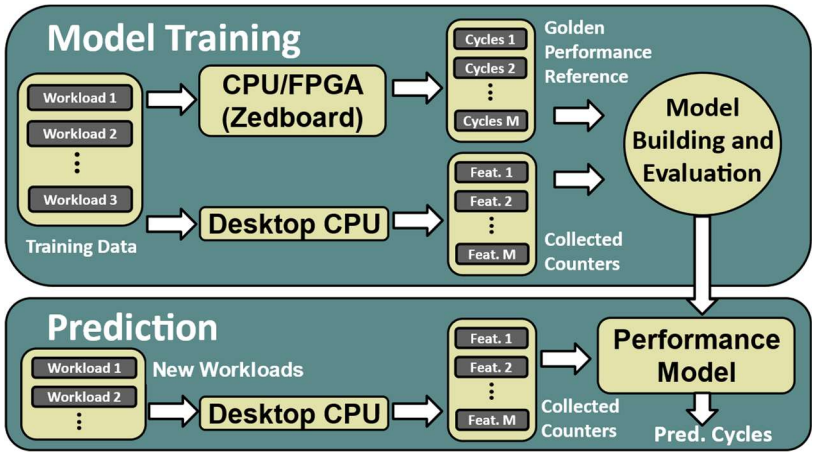


Figure 4:4 HLSPredict Model Training and Application

Model Training (Top): Performance counters are collected from the host CPU, and target metrics are collected from the synthesized FPGA accelerator design. Models are then trained to predict FPGA cycle count and power. (Bottom): During model deployment, a new workload runs on the host CPU and the model is then applied to predict FPGA cycle counts and power. Again, a complementary version for each power metric also exists but is not depicted.

Fig. 4:4 details the model training process, the required collateral (each workload's CPU program counter measurements, FPGA cycle counts and FPGA power measures), and how to apply the model in practice to estimate FPGA cycle counts and power. The predictive models and their training process are programmed using Python3.6 and Scikit-learn 0.18.1 [48].

## 4.2 FPGA Accelerator Design and Template

All Polybench workloads were synthesized using Vivado HLS. Each workload is specified in synthesizable C and verified via C/RTL co-simulation. In all instances, the HLS accelerator design is integrated into an architectural template shown in Fig. 4:5.

The template leverages the *Advanced eXtensible Interface (AXI*) stream protocol for communication, and *Direct Memory Access (DMA)* for memory transactions. We use the *Accelerator Coherency Port (ACP)* of the DMA to leverage the CPU cache hierarchy, creating a simple but effective system on chip (SOC). This template assumes (and we select) workload input sizes that are small enough to fit into onboard BRAM and are templatized to accept standard data types but evaluated using the 64-bit double type.

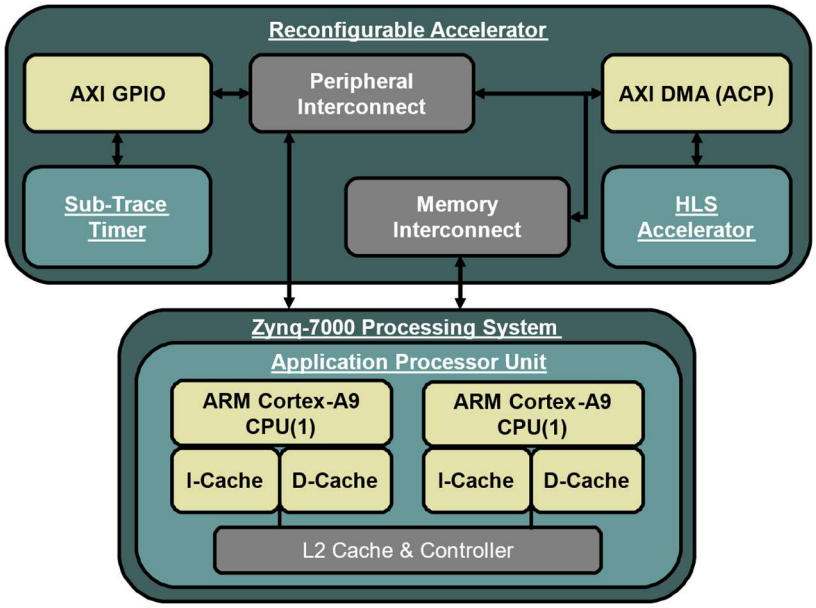


Figure 4:5 HLSPredict Hybrid CPU-FPGA architectural template
The template for all workloads. Each workload updates the HLS accelerator, and its internal trace generation logic.

## 4.3 Regression Models

HLSPredict trains and evaluates 8 linear and 2 non-linear models, summarized in Fig. 4:6. The choice to use an ensemble of models is driven by the suspicion that correlations between

collected features (host CPU performance counters), and the amount of non-linearity in the relationship between features and FPGA cycles or power measurements may vary as the degree of parallelism exploited by HLS changes. Ref. [8] describes each of the models listed below. Each model is chosen to target one of these behaviors. Linear models based on OLS are useful when the relationship between features and target FPGA cycles are linear, and features are non-correlated. NNLS is an OLS variant that removes features with negative coefficients from the model, often sacrificing overall model accuracy. OLS could predict negative execution time for some workload; NNLS provably cannot.

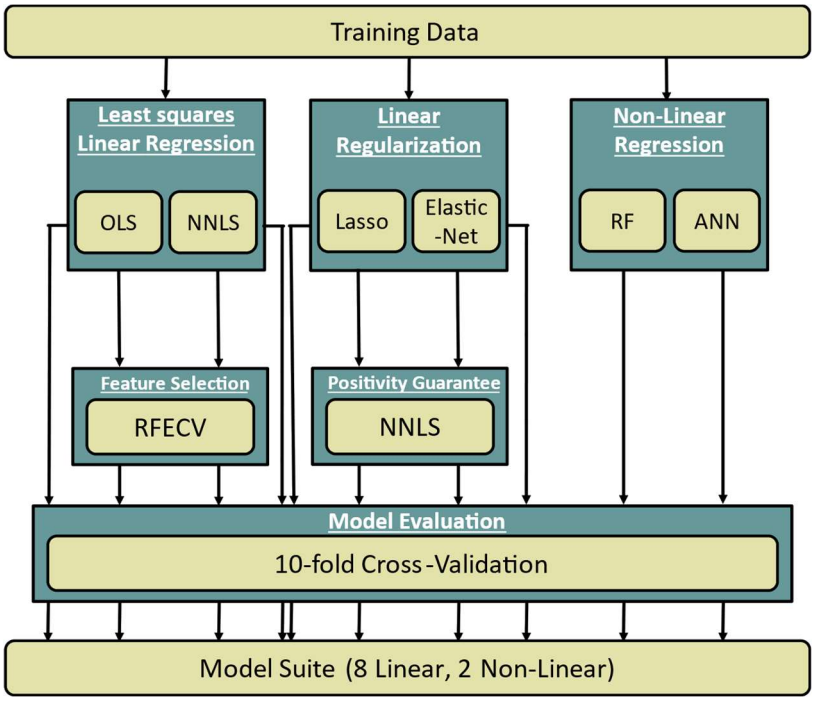


Figure 4:6 HLSPredict Model Training and Application

HLSPredict creates eight linear models, and two non-linear models (RF, ANN). We now employ one feature selection technique, the RFECV and have re-implemented the previous modelling suite in Python using Scikit-Learn.

*Feature selection* removes features from the model by setting their corresponding coefficients to zero, improving *prediction accuracy* by sacrificing bias to reduce variance. HLSPredict uses *Recursive Feature Elimination with Cross-Validation (RFECV)*, which repeatedly

trains a model, and one-by-one ranks features by their impact on model accuracy [49] within a nested CV loop; this allows RFECV to determine the correct number and combination of features to include in the model, while ensuring model generalizability. Our framework includes OLS and NNLS with and without RFECV.

*Regularized* models select features during model construction, which can help to mitigate variance and noise on the prediction curve and can improve model accuracy. We employ two regularization models, the *Lasso* (L1 regularization), and *Elastic-net* (hybrid L1/L2 regularization). Our model suite includes Lasso and Elastic-net, both with and without NNLS post-processing.

HLSPredict also contains two non-linear models: RF, and an *Artificial Neural Network (ANN)*, which capture the presence of non-linear behavior which often becomes prevalent as additional compute parallelism is exposed. HLSPredict uses 10-fold CV as a precursor to estimate the usefulness of a trained model in practice.

For each prediction scenario reported, we train all 10 models, selecting the one the minimizes the $E_{out}$ as the most accurate. We also report 10% and 20% IRs for each model and compare them across thresholds of $T = 10\%$ and 20%, sufficient accuracies given model application speed.

## 4.4 Experimental Methodology

We present eight hardware-assisted, cross-architecture predictive model scenarios based on dynamic x86 CPU performance measurements which provide 75 features. For each scenario, we report the most accurate model in terms of $E_{out}$ and $I_R$. In practice, each model would be trained once using actual FPGA cycle counts and synthesized power measurements as targets, which are obtained from direct execution of the training workloads, which have been synthesized, placed, and routed; once the models have been trained, they can safely be applied to predict the performance and power consumption of previously unseen workloads without involving the FPGA or its

synthesis tools. Thus, the cost of model training is amortized over the lifetime of the model once it has been deployed. Table 4:2 outlines the scenarios below.

For each scenario, we generate two sets of performance models, one for the baseline workloads, and the other for the optimized workloads, the latter of which allows us to quantify HLSPredict's ability to scale to FPGA workloads with highly varying degrees of parallelism and compute-performance.

We also produce six predictive models for static, dynamic, and total power for the baseline and optimized workloads. For our power modeling scenarios, we only collect counters at the end of the workload's execution, omitting sub-traces, as power per epoch is not a meaningful measure. In all cases our models only utilize the host CPU performance counter measures to predict the FPGA cycle count of our workloads, without requiring user intervention, HLS, or other information gleaned from the FPGA.

| Scenario | Feature Source | Target Data | $E_{out}$ (%) |
|---|---|---|---|
| $Scenario_1$ | Baseline sub-traces | Baseline FPGA Cycles | 9.08 |
| $Scenario_2$ | Optimized sub-traces | Optimized FPGA Cycles | 9.79 |
| $Scenario_3$ | Baseline workloads | Baseline Total Power | 7.84 |
| $Scenario_4$ | Baseline workloads | Baseline Dynamic Power | 4.28 |
| $Scenario_5$ | Baseline workloads | Baseline Static Power | 1.88 |
| $Scenario_6$ | Optimized workloads | Optimized Total Power | 4.48 |
| $Scenario_7$ | Optimized workloads | Optimized Dynamic Power | 4.69 |
| $Scenario_8$ | Optimized workloads | Optimized Static Power | 2.21 |

Table 4:2 HLSPredict Prediction Scenarios Evaluated
We use eight prediction scenarios, two target FPGA performance and six target power to validate HLSPredict.

### 4.4.1 Host CPU

Host workload execution is performed on a desktop PC with a Haswell generation Intel(R) Core(TM) i5-4670K CPU running at 3.40GHz with 16GB of DDR3 DRAM. All frequency scaling and sleep states made available in the *Basic Input-Output System* (BIOS) are disabled and non-essential background OS processes in Ubuntu 16.04 are killed. This minimizes 3rd party application interference when executing each workload. We do not modify the drivers or OS to

ensure that their influence is captured in our models; the drawback is that we are unable to control frequency and deep sleep states. Our models also capture the influence of OS-related background tasks that cannot be readily killed, and deep sleep states not available in the BIOS.

Host CPU performance counters, which are used as model features, can be grouped into two broader categories: *core-local* and *socket-wide* counters. These can be further broken down into three groups each. For core-local counters we profile *fixed-purpose* counters, *registers*, which track single events chosen by Intel, four *general-purpose* counters which are programmable via a config and a counter register, and a single *fixed thermal counter*. The socket-wide counters track metrics related to everything other than CPU cores, and can be categorized into *energy counters*, consisting of only fixed counters, the *uncore global counters*, consisting of two programmable counters and one fixed clock cycle counter, and, lastly, two programmable *last level cache counters*.

### 4.4.2 Target HLS Derived FPGA Accelerator

Accelerator synthesis targets a Xilinx Zynq-7000 SOC (XC7Z020) development board. This platform features a dual-core CPU with ARM Cortex-A9 MPCore processor cores operating at 667MHz, as well as 256KB of on-chip random access memory. The CPU features 32KB of L1 cache for each core and 512KB of shared L2 cache. In addition to on-chip memory, the board includes 512MB of DDR3 DRAM, 256MB of quad-serial peripheral interface (SPI) flash, and a 4GB SD card. The XC7Z020 platform also contains an integrated FPGA, featuring 53,200 LUTs, 220 DSP slices and 4.9MB of *Block Random Acceess Memory* (BRAM).

Synthesis of FPGA accelerators and FPGA sub-trace cycle count collection is required only for model training. We created two versions of each Polybench workload using the Xilinx Vivado 2017.2 HLS toolset, one using default HLS behavior and the other optimized for performance. To accelerate workloads, we pipeline the outermost loop, unroll inner loops, and

partition arrays using Vivado HLS directives [50]. When physical resource becomes a constraining factor, we eschew pipelining the outermost loop to limit duplication of hardware resources. If needed, we reduce the unroll factor of the innermost loops until the design will fit on the FPGA. We cannot optimize loops that have a variable iteration count or inter-loop index dependencies.

### 4.4.3 Sub-trace Generation

A Sub-trace is an epoch (or sub-region) within a much longer execution stream of a workload. We produce sub-traces for both the CPU and HLS workloads to improve model fidelity for performance estimation. Sub-traces only encompass active compute time; they do not account for FPGA streaming, communication, or data initialization time. Sub-traces for CPU workloads consist of the performance counter measures for each epoch; for FPGAs they consist of a one-time reading of the FPGA cycle count, which resets when the next sub-trace starts.

*5.3.1 CPU Sub-Trace Generation.* Each Polybench workload is instrumented with the likwid-perfctr Marker API by modifying kernel parameters to pass the likwid external variables (#perfCounters, perfCounters, time, executionCount). We use these parameters, and API functions to collect the specified number of sub-traces, each of which consists of the CPU performance counter readings made available by likwid pre-configured Haswell performance groups [51]. Subtrace execution times are dominated by API initialization, file I/O overheads and the repeated executions needed to collect all counters. With sub-traces, the average execution time per workload is 43.23s, an average slowdown of 12,171x over direct execution of non-instrumented code. Sub-trace execution is still considerably faster than the average FPGA synthesis time, which is 25.7 m for baseline and 31.8 m for optimized workloads.

*5.3.2 FPGA Sub-Trace Generation.* To generate sub-traces, we built a timer in Verilog to count FPGA clock cycles for a workload-dependent number of loop iterations, and to write the cycle count to an array in memory. The timer communicates with the HLS-generated workload-

accelerator *Intellectual Property* (IP) block through a handshaking protocol and communicates with memory using the AXI General Purpose Input-Output (GPIO) protocol.

The architecture template is assembled using Vivado IP integrator and programmed via the Xilinx Software Development Kit (SDK), which dumps the collected metrics to a file. The HLS IP block pauses while sub-trace clock cycles are written to memory via AXI GPIO. Reported FPGA cycle counts do not include the time to write timer values to memory or other delays related to instrumentation.

## 4.5 Experimental Results

We generate 10 models for each of the 8 prediction scenarios we target (see Table 4:2). For each model, we report the $E_{out}$, 20% and 10% IRs, the number of selected features, and the number of available features; we also report the APE for each workload. During model application, our method produces power and performance estimates 36.24x faster for baseline workloads, and 43.78x faster for optimized workloads than FPGA synthesis and execution. Each speedup number is calculated by 1) collecting the CPU execution time (including performance counter collection), 2) profiling the time it takes to apply all of the most accurate models for all scenarios related to either the baseline workloads ($Scenarios_{1,3,4,5}$) or the optimized workloads ($Scenarios_{2,6,7,8}$) and 3) collecting the FPGA synthesis and execution time in cycles and computing the wall- clock time using the clock period for the same set of workloads. We then sum 1) and 2) for each respective case and divide by 3) for the same case.

### 4.5.1 Predicting Default HLS Accelerator Cycles

Table 4:3 reports the out-of-sample accuracy of all models generated by HLSPredict for $Scenario_1$. We see that RF is more accurate than any linear model variant, and the non-linear ANN with an average $E_{out}$ of 9.08% across all workloads and sub-traces. This indicates that linear models are inefficient for this cross-platform prediction scenario. Fig. 4:7 reports the observed and

predicted FPGA cycle times, and the resulting relative absolute percentage error (APE), ordering sub-traces by their observed FPGA cycle in non-decreasing order from left to right.

| Model | $E_{out}$(%) | $I_R$(20%) | $I_R$(10%) | # Features Used |
|---|---|---|---|---|
| Random Forest | 9.08 | 94.03 | 90.79 | 69 |
| NNLS | 462.63 | 25.65 | 17.11 | 13 |
| Lasso/NNLS | 521.20 | 20.25 | 8.00 | 4 |
| Lasso | 521.20 | 20.25 | 8.00 | 4 |
| OLS | 655.74 | 48.55 | 33.06 | 75 |
| NNLSRFECV | 1019.68 | 10.32 | 5.99 | 3 |
| ANN | 1615.67 | 35.19 | 20.43 | 75 |
| OLSRFECV | 13515.60 | 5.27 | 2.80 | 2 |
| Elastic/NNLS | 59644.50 | 7.05 | 5.33 | 72 |
| Elastic | 59644.50 | 7.05 | 5.33 | 72 |

Table 4:3 HLSPredict Scenario$_1$ Model Comparison

We compare the $E_{out}$ and the $I_R$ at 10% and 20% error thresholds for all 8 models generated for Scenario1. RF is most accurate, with all linear models performing poorly.

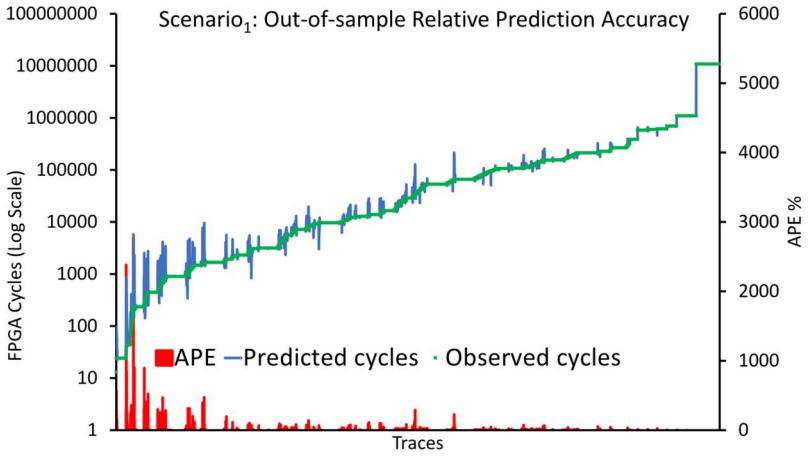


Figure 4:7 HLSPredict Scenario$_1$ Relative Error

Here we demonstrate the Observed FPGA cycle count, the Predicted FPGA cycle counts, and the APE of each sub-trace used for FPGA baseline performance prediction (Scnaerio$_1$) via the RF model.

The sub-traces with largest error tend to be those with the smallest FPGA cycle count, noting that a small absolute deviation can lead to large APE. Fig. 4:8 reports the $I_R$ at various thresholds *T*: RF, even with aggressive inlier thresholds has few outliers, and is highly accurate for

most traces. RF obtains an $I_R$ of 94.03% inliers at T=20%, while Lasso, the second most competitive model, obtains an $I_R$ of 48.5%.

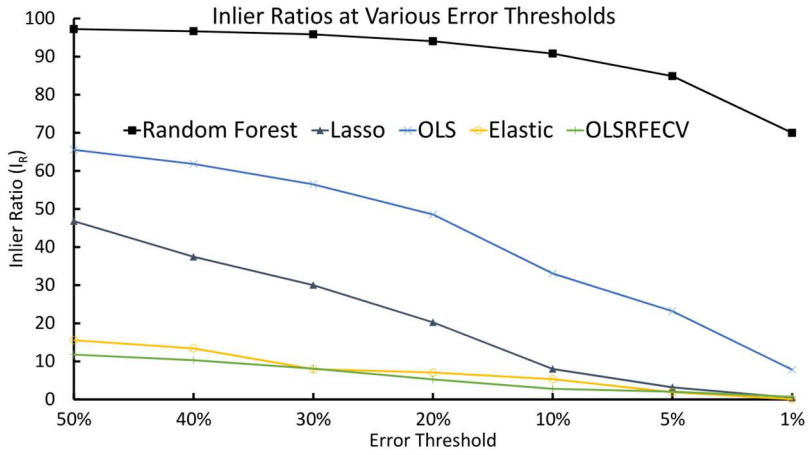


Figure 4:8 HLSPredict Scenario$_1$ $I_R$
Inlier Percentages for all models demonstrating RF's clear accuracy advantage for Scenario1.

### 4.5.2 Predicting Optimized HLS Accelerator Cycles

Table 4:4 reports the accuracy of the HLSPredict models for Scenario$_2$. Once again, we observe that RF, which obtains an $E_{out}$ of 9.79%, is much more accurate than the linear models and the non-linear ANN. Fig. 4:9 reports the APE and predicted/observed FPGA cycle counts, with most outliers being present in the smallest sub-traces. One notable outlier exists, in the middle of the curve in Fig. 4:9, with an exorbitant error of ~5890%; it explains the seemingly counterintuitive observation that Scenario$_2$ has a lower $E_{out}$ than Scenario$_1$, yet higher inlier-ratios at $T \leq 10\%$. Therefore, it is important to evaluate models using both $I_R$ in addition to $E_{out}$, as mean-based statistics are sensitive to outliers.

### 4.5.3 Predicting Default HLS Accelerator Power

Next, we use HLSPredict framework to create predictive models for FPGA (static, dynamic, and total) power consumption in Watts. For these experiments, we do not leverage the sub-trace methodology on either the CPU or FPGA, as power per sub-trace is neither meaningful

nor collectible. Instead, we collect one end-to-end set of performance counter measurements per workload, encompassing the entirety of the execution on the host CPU.

| Model | $E_{out}$(%) | $I_R$(20%) | $I_R$(10%) | # Features Used |
|---|---|---|---|---|
| Random Forest | 9.79 | 96.98 | 95.24 | 68 |
| OLS | 112.46 | 40.49 | 31.42 | 75 |
| OLSRFECV | 115.73 | 41.23 | 31.47 | 74 |
| ANN | 257.08 | 17.15 | 11.35 | 75 |
| NNLSRFECV | 305.21 | 12.04 | 9.17 | 3 |
| NNLS | 392.58 | 7.53 | 3.62 | 14 |
| Lasso/NNLS | 506.00 | 5.35 | 0.35 | 3 |
| Lasso | 506.00 | 5.35 | 0.35 | 3 |
| Elastic/NNLS | 10334.21 | 0.00 | 0.00 | 72 |
| Elastic | 10334.21 | 0.00 | 0.00 | 72 |

Table 4:4 HLSPredict Scenario$_2$ Model Comparison

We compare the $E_{out}$ and the $I_R$ for all 8 models generated for Scenario$_2$. RF is again the most accurate.

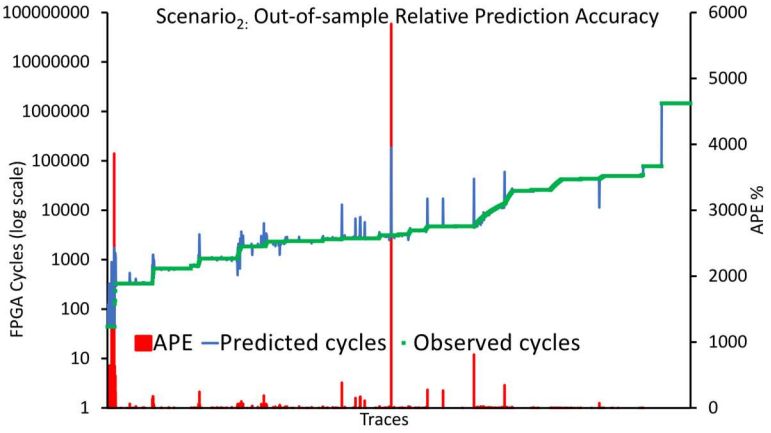


Figure 4:9 HLSPredict Scenario$_2$ Relative Error

Here we demonstrate the Observed FPGA cycle count, the Predicted FPGA cycle counts, and the APE of each sub-trace used for FPGA optimized performance prediction (Scenario$_2$) via the RF model.

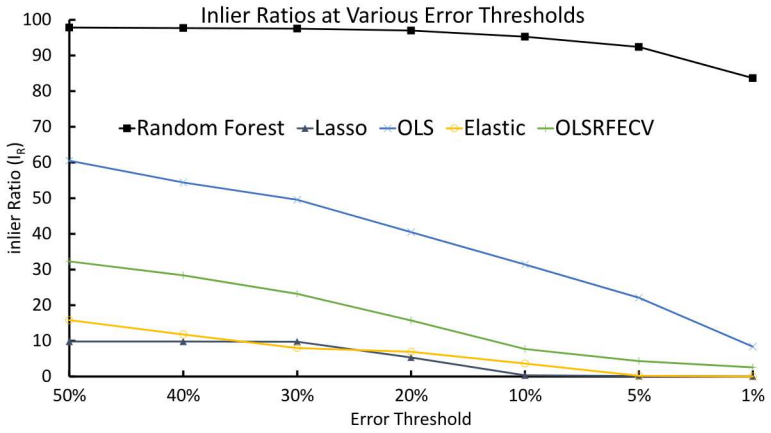


Figure 4:10 HLSPredict Scenario$_2$ $I_R$

Inlier Percentages for all models demonstrating RF's clear accuracy advantage for Scenario$_2$.

Table 4:5 reports the $E_{out}$ and $I_R$ of the three most accurate models (to conserve space) for Scenarios$_{3\text{-}5}$. RF is the most accurate in all cases with 7.84%, 4.28% and 1.88% $E_{out}$ for total, dynamic and static power respectively.

For Scenario$_3$, total power, the runner up models, NNLS/RFECV and OLS/RFECV, obtain >30% error, but for dynamic and static power, they obtain within 0.5% of RF's error. We do not have a statistical explanation for this observation; we suspect that RFECV had difficulty with this data set. In the case of Scenario$_4$, dynamic power, the NNLS/RFECV model has higher inlier rates at the 10% threshold. In this case, the linear models are competitive with RF, suggesting that CPU counters correlate more strongly with FPGA power consumption than with performance. RF construction randomly selects a subset of features to include in each regression tree, so it is likely, although not guaranteed, that all or nearly all features will appear in at least one tree in a forest of sufficient size. In contrast, RFECV removes statistically insignificant features from linear regression models.

| Model | $E_{out}$(%) | $I_R$(20%) | $I_R$(10%) | # Features Used |
|---|---|---|---|---|
| Random Forest | 7.84 | 93.75 | 81.25 | 60 |
| NNLSRFECV | 31.70 | 9.38 | 3.13 | 3 |
| OLSRFECV | 32.21 | 9.38 | 3.13 | 2 |

Table 4:5 HLSPredict Scenario$_3$ Model Comparison

Comparison of the top 3 Total Power models for optimized workloads.

| Model | $E_{out}$(%) | $I_R$(20%) | $I_R$(10%) | # Features Used |
|---|---|---|---|---|
| Random Forest | 4.28 | 100.00 | 90.63 | 70 |
| OLSRFECV | 4.30 | 100.00 | 84.38 | 2 |
| NNLSRFECV | 4.57 | 100.00 | 93.75 | 2 |

Table 4:6 HLSPredict Scenario$_4$ Model Comparison

Comparison of the top 3 Dynamic Power models for optimized workloads.

| Model | $E_{out}$(%) | $I_R$(20%) | $I_R$(10%) | # Features Used |
|---|---|---|---|---|
| Random Forest | 1.88 | 100.00 | 100.00 | 66 |
| NNLSRFECV | 2.20 | 100.00 | 100.00 | 2 |
| OLSRFECV | 2.39 | 100.00 | 100.00 | 3 |

Table 4:7 HLSPredict Scenario$_5$ Model Comparison

Comparison of the top 3 Static Power models for optimized workloads.

### 4.5.4 Predicting Optimized HLS Accelerator Power

We use the same approach as outlined in the preceding subsection to predict the power consumption of our optimized accelerator implementations; Table 4:6 reports the out-of-sample accuracy and IRs of three most accurate models, noting that the most accurate model in this case is NNLS/RFECV, which outperforms RF by 1.24%, 1.34% and 0.91% for total, Scenario$_6$, dynamic, Scenario$_7$, and static power, Scenario$_8$, respectively. NNLS/RFECV uses far fewer features than RF due to the feature selection processes employed. This shortens model application time and highlights statistically significant features that are useful for power prediction. We show and analyze these features in Section 6.5.

| Model | $E_{out}$(%) | $I_R$(20%) | $I_R$(10%) | # Features Used |
|---|---|---|---|---|
| NNLSRFECV | 4.48 | 100.00 | 88.24 | 3 |
| Random Forest | 5.72 | 100.00 | 82.35 | 43 |
| Lasso | 5.83 | 100.00 | 88.24 | 6 |

Table 4:8 HLSPredict Scenario$_6$ Model Comparison

Comparison of the top 3 Total Power models for optimized workloads.

| Model | $E_{out}$(%) | $I_R$(20%) | $I_R$(10%) | # Features Used |
|---|---|---|---|---|
| NNLSRFECV | 4.69 | 100.00 | 88.24 | 3 |
| Random Forest | 6.03 | 100.00 | 82.35 | 68 |
| Lasso | 6.08 | 100.00 | 82.35 | 6 |

Table 4:9 HLSPredict Scenario$_7$ Model Comparison

Comparison of the top 3 Total Power models for optimized workloads.

| Model | $E_{out}$(%) | $I_R$(20%) | $I_R$(10%) | # Features Used |
|---|---|---|---|---|
| NNLSRFECV | 2.21 | 100.00 | 100.00 | 8 |
| Random Forest | 3.12 | 100.00 | 100.00 | 62 |
| Lasso/NNLS | 3.48 | 100.00 | 94.12 | 7 |

Table 4:10 HLSPredict $Scenario_8$ Model Comparison
Comparison of the top 3 Total Power models for optimized workloads.

### 4.5.5 Random Forest Cycle Feature Ranking

RF models typically use all features as long as the number of trees in the forest is sufficiently large. In our models, RF uses between 60 and 70 of the available 75 features; those not included are omitted due to the random feature sampling employed by RF. Tables 4:11 and 4:12 report the top 10 most impactful features used for FPGA cycle prediction via RF, ranked by RSS, $Scenario_1$ and $Scenario_2$. Feature ranking [52] provides insight into how CPU and FPGA performance correlates.

The second column indicates whether the metric is a programmable counter provided by the likwid-perfctr event groups, or a fixed counter; we explicitly list the category for fixed counter measures and refer to programmable counters simply as programmable. Ref [51] provides a detailed description of the peformance counters that are summarized and discussed here.

| Counter | Counter Type | RSS Value |
|---|---|---|
| UOPS_ISSUED_FLAGS_MERGE | Programmable | 0.35220 |
| UOPS_ISSUED_STALL_CYCLES | Programmable | 0.09915 |
| UOPS_EXECUTED_USED_CYCLES | Programmable | 0.08886 |
| MEM_UOPS_RETIRED_LOADS | Programmable | 0.08758 |
| L2_TRANS_L2_WB | Programmable | 0.08508 |
| UOPS_EXECUTED_PORT_PORT_7 | Programmable | 0.08119 |
| UOPS_EXECUTED_PORT_PORT_2 | Programmable | 0.06955 |
| UOPS_EXECUTED_PORT_PORT_1 | Programmable | 0.03573 |
| BR_INST_RETIRED_ALL_BRANCHES | Programmable | 0.01695 |
| L2_RQSTS_MISS | Programmable | 0.01365 |

Table 4:11 HLSPredict $Scenario_1$ RF Feature Ranking
RF top-10 feature ranking for FPGA cycle count prediction ($Scenario_1$: baseline workloads).

The 3 highest ranking counters for Scenario$_1$, shown in Table 4:11 track *micro-op* (UOP) execution. UOPS are low-level hardware instructions, created in the CPU front-end by decoding program code into architecure operations, which handle items like register arithmetic, and data transfer into and out of registers and CPU busses[53]. ISSUED_FLAGS_MERGE, highest ranked, relates to a merge operation required when executing shifts, which are prevelant in multiplication, a ubiquitous operation in virtually all polybench workloads. ISSUE_STALL_CYCLES, the 2$_{nd}$ highest ranking counter, reports the number of cycles stalled when issuing uops, indicates resource saturation; finally the 3$_{rd}$ highest-ranking counter, UOPS_EXECUTED_USED_CYCLES, tracks the number of cycles that execute micro-ops of any type.

For Scenario$_2$ (Table 4:12), ISSUE metrics arise among the highest ranking counters, but UOPS_EXECUTED_USED_CYCLES is no longer present in the top-10 features. In this case, the highest ranking counters focus on stall cycles that occur during the CPU pipeline's retirement stage, and L2-L3 cache communication (L2_LINES_IN_ALL). Notably, the UOPS_EXECUTED_PORT_# metrics increase in their ranking between Tables 4:11 and 4:12. The UOPS_EXECUTED_PORT_# metrics track the number of uops scheduled and executed on a particular port. Some uops can only be executed on a specific or subset of the ports. The summation of all executed uops on all ports indicates the number of uops executed while the workload is running, and individual port metrics can highlight broad groups of uops that can only be executed on that PORT_#.

| Counter | Counter Type | RSS Value |
|---|---|---|
| L2_LINES_IN_ALL | Programmable | 0.21432 |
| UOPS_RETIRED_STALL_CYCLES | Programmable | 0.21357 |
| UOPS_EXECUTED_PORT_PORT_7 | Programmable | 0.13210 |
| UOPS_EXECUTED_PORT_PORT_3 | Programmable | 0.11764 |
| UNCORE_CLOCK | UncoreGlobal | 0.10641 |
| CPU_CLOCK_UNHALTED_TOTAL_CYCLES | Programmable | 0.10580 |
| UOPS_EXECUTED_PORT_PORT_6 | Programmable | 0.05560 |
| UOPS_ISSUED_STALL_CYCLES | Programmable | 0.01204 |
| UOPS_RETIRED_STALL_CYCLES | Programmable | 0.00759 |
| MEM_LOAD_UOPS_RETIRED_ALL_ALL | Programmable | 0.00450 |

Table 4:12 HLSPredict Scenario$_2$ RF Feature Ranking
RF top-10 feature ranking for FPGA cycle count prediction (Scenario$_2$: optimized workloads).

Due to the out-of-order nature of the x86 CPU, the UOPS_EXECUTED_PORT metrics can and often do execute in parallel for certain operations, though none of these metrics track that explicitly, indicating that the total number of operations, rather than the degree of parallelism leveraged in the CPU microarchitecture, can better predict FPGA performance and power. This shows that RF selects useful features, as improvement in accuracy was obtained through exposing and utilizing additional fine-grained parallelism via HLS directives, wherein the degree of parallelism and the number of uops executed become increasingly important predictors.

| Counter | Counter Type | RSS Value |
|---|---|---|
| CYCLE_ACTIVITY_STALLS_LDM_PENDING | Programmable | 0.118565 |
| CACHE_LOOKUP_WRITE_MESI | Programmable | 0.105664 |
| L2_TRANS_L2_WB | Programmable | 0.075476 |
| CACHE_LOOKUP_WRITE_MESI | Programmable | 0.061265 |
| CYCLE_ACTIVITY_CYCLES_NO_EXECUTE | Programmable | 0.047509 |
| DTLB_LOAD_MISSES_WALK_DURATION | Programmable | 0.042106 |
| L2_TRANS_L2_WB | Programmable | 0.040278 |
| L2_LINES_IN_ALL | Programmable | 0.035629 |
| CACHE_LOOKUP_READ_MESI | Programmable | 0.035433 |
| CACHE_LOOKUP_WRITE_MESI | Programmable | 0.035087 |

Table 4:13 HLSPredict Scenario$_3$ RF Feature Ranking
RF top-10 feature ranking for FPGA Total Power prediction (Scenario$_3$: baseline workloads).

### 4.5.6 Power Model Feature Ranking and Analysis

Table 4:13 presents the top ten highest-ranked features as ranked by RF for Scenario3, the most accurate model. The counters reported in Table 4:13 correspond to cache metrics. For example, CYCLE_ACTIVITY_STALLS_LDM_PENDING, the highest-ranking counter, measures CPU stalls caused by traffic in the cache hierarchy. We see that for $Scenario_6$, a similar trend holds, albeit in this case NNLSRFECV, a linear model that employs features selection is the most accurate. In this case only 2 CPU counters and the model intercept are required to obtain a highly accurate model for all workloads. The two CPU counters are INT_MISC_RECOVERY_CYCLES, which measures the number of cycles used for recovery after tasks like SSE exceptions, memory disambiguation, etc. and the second CPU counter is DTLB_STORE_MISSES_WALK_DURATION, which measures the duration in cycles that a TLB walk will take in the event of a TLB miss. Of note, both metrics are cycle counts, indicating that time spent executing during recovery or memory hierarchy misses are important power predictors. While only one of the counters employed by NNLSRFECV for Scenario6 (CYCLE_ACTIVITY_STALLS_LDM_PENDING) is present in Table 4:13, which lists the most influential counters for RF on the baseline workloads; they are both used in the most accurate model for $Scenario_{7\text{-}8}$, (NNLSRFECV).

# Chapter 5 Related Works

For CPUs and GPUs, CASs are employed to provide performance and power estimates. For FPGA designs, FPGA synthesis tools [6, 7], whose design flows typically include software simulation before synthesis, are used to obtain early estimates of design performance. In all cases, the importance of detailed architectural simulation is [54] well-established, and simulators are often used for design prototyping and verification, DSE, performance evaluation of workloads given a design, assessing architectural innovations, and for software/hardware co-design and performance tuning of software [1].

Cycle-accurate architectural simulators such as GPGPU-Sim [55], Atilla [56], and Multi2Sim [57] run orders of magnitude slower than native execution [58, 59] due to their detailed timing and functional execution. Functional simulators lack timing information [1], but run considerably faster than CAS. The feasibility of these tasks is often improved by reducing cycle accurate simulation time in a variety of ways. The speed of cycle accurate architectural simulation is cost prohibitive, typically executing between 1 thousand instruction per second (KIPS) and 1 million instructions per second (MIPS) [2].

Techniques to reduce simulation times, such as representative statistical sampling [8, 60, 54, 9], synthetic benchmark reduction [61], synthetic benchmark generation, such as for cache coherent traffic [62, 63], parallelization efforts [2, 4, 64], FPGA hardware assistance [65], and attempts to raise the level of abstraction (lower the level of detail), as employed for GPUs by RastSim of Chapter 3, and CPUs by Sniper [3] are helpful but remain prohibitively slow.

In response, we have turned to cross-architecture predictive modeling, as a potential solution to this conundrum. The predictive models are intended to speedup pre-silicon GPU and pre-RTL FPGA design tasks by avoiding cycle-accurate simulation in favor of estimating design performance via predictive models instead. Although predictive models are less precise than cycle-

accurate simulation, and cycle-accurate simulation is required for CPU and GPU model training and is explicitly leveraged as model input for DSE targeting FPGA-based accelerators, predictive modeling can and should reduce the amount of simulation and synthesis required once a model is deployed. This provides a substantial productivity advantage compared to existing design and synthesis methodologies for these targets. In principle, the predictive modeling techniques advocated in this paper should be viewed as being complementary to cycle-accurate simulation.

The remainder of this section evaluates predictive modeling targeting CPUs (Section 5.1), GPUs (Section 5.2), and FPGAs (5.3), with specific emphasis on cross-platform models based on machine learning.

## 5.1 CPU models

### 5.1.1 Statistical Models for CPUs

The majority of work on predictive modeling targets CPUs executing general-purpose workloads. The overall objective is to limit the number of simulations required to evaluate design points in a much larger architectural design space. Our predictive modeling frameworks for GPUs and FPGAs are inspired by predictive models for CPU performance and power. CPU predive modeling has leveraged several statistical and machine learning techniques. We compare these in Table 5:1.

For example, Ipek et al. [66] consider a design space comprising 250K design points, and sample a representative subspace (~2K points, or 1-2% of the total design space), using active learning techniques. Model training leverages iterative refinement to build an ensemble of *artificial neural networks* (ANNs), in which each represents one of a 10-fold CV process. Error rates range from 2-5%, with models created for individual CPUs, a multiprocessor, and the memory subsystem.

Similarly, Lee et al. [67] trains regression models to predict performance and power consumption of a large number of design points (22 billion) using a representative sub-sample

(4000), obtained via *uniform at random* (UAR) sampling. The regression models include linear least-squares models as well as non-linear spline functions, in which the predictive function is decomposed into multiple piecewise polynomials. Lee et al. train four categories of models, Baseline, variance stabilized, regional and application specific. Appliation specific models are most accurate for performance, achieving an average error of 4.1%, while regional models, in which applications are grouped by feature similarity, are best for power achieving 4.3% average error.

D*ynamic thermal management* (DTM) often employs power models to reduce energy consumption and prevent thermal emergency events. For example, Nath et. al. [68] developed an analytical multi-core power model for the Intel *Knights Ferry* (KNF) architecture, comprising static and dynamic models for compute, memory, and interconnect power, with a 4.73% average error rate. The model relies on expert knowledge to identify the most relevant performance counters for inclusion as features; the model itself is trained using the selected features and the HotSpot thermal simulator [69] configured to model KNF thermal properties. The model itself has low overhead and reduces energy consumption by 14%, and the occurrence of thermal emergency events by 58%.

Like HALWPE of Chapter 2 and HLSPredict of Chapter 4, these models are built using performance counter readings and program metrics as features; however, the features that effectively predict CPU performance and power differ from those that are useful for GPU and FPGA accelerator performance are different due to architectural dissimilarities.

Ma et al. [70] report that models trained using detailed simulators can be more accurate than models based on performance counters obtained from direct execution on hardware; the reason is that simulators can be configured to collect performance metrics on architectural subsystems for which post-silicon performance counters are not available. Although we do not compare with models obtained from direct execution on post-silicon GPU hardware, we exploited this observation to construct the WCF for RastSim in Chapter 3.

Approximate analytical models for out-of-order processors can perform high-level microarchitectural analysis [71]. This method requires a trace-driven off-line analysis of the model parameters to determine program locality behavior and miss rates as well as drain time after branch miss-prediction. The model does not generalize for workloads not included in the off-line analysis.

| Paper | Model | Features | Accuracy (avg.) | Speed | Benefit | Drawback |
|---|---|---|---|---|---|---|
| [66] | Ensemble of ANNs | Architectural parameters and latencies | Performance : 95-98% | One order of magnitude fewer simulations required | Predictive models are trained on a subset of design points to predict the full space | Requires detailed architectural expertise and many simulations to characterize latencies. |
| [67] | Linear and piecewise cubic spline models | Architectural parameters and latencies | Performance : ~95.9% Power: ~95.6% | ~7 orders of magnitue fewer simulations required | Predictive models are used to avoid exhaustive simulation | Requires detailed architectural expertise and many simulation to characterize latencies |
| [68] | Analytical model | Architectural parameters and throughput measurements | Power: ~95.3% | NA | Performance feedback after model training reduces energy and avoids thermal throttling | Performance overhead; Hand-tuned analytical modeling requires expert knowledge to select features and derivce equations; this limits portability. |
| [72] | PCA with Lasso and CLSLR | Host CPU performance counters collected once per workload | Lasso: ~83-73% CLSR: ~ 99% For total workload | Near-native direct execution on CPU host (~500 MIPs) | ***Cross-platform***; executes faster than simulator or instrumented workload | End-to-end prediction sacrifices accuracy when compared to the phase-level approach. [12] |
| [12] | Modified Lasso | Host CPU performance counters collected once per phase | Performance and Power: >90% at phase boundaries | Near-native direct execution on CPU host (~500 MIPS) | ***Cross-platform***; models power and performance; high fidelity phase-level predictions | Choice of phase-granulartiy impacts accuracy and speed; some parameter tuning is required to balance accuracy vs. overhead. |

Table 5:1 Comparison of statistical models for CPUs.

We compare the approach, accuracy, benefits and drawbacks of predictive CPU models.

### 5.1.2 Cross-Architecture Models for CPUs

LACross [72] appears to be the first work to perform cross-platform and cross-instruction set architecture (ISA) CPU performance prediction. The initial iteration runs at near-native

hardware speeds, and accurately predicts the performance of 157 Association for Computing Machinery (ACM) *International Collegiate Programming Contest (*ACM-ICPC) programs from a variety of application domains. LACross used two commercially available processors as *host* and *target* interchangeably, an Intel Core-i7 920 with 24GB DRAM and the AMD Phenom II X6 1055T with 8GB DRAM. LACross first executes each workload on the host to collect performance counter measurements and executes each workload on the target to measure performance. The host performance counter measurements, one read per workload, are used to train two regression models, the *LASSO L1 regularization model (LASSO)* [35] and the *Constrained Locally Sparse Linear Regression Model* (CLSR), each combined with *Principle Components Analysis* (PCA) [73] to extract latent semantics in the feature data, which improves model accuracy. The average LASSO cross validation (CV) error, an estimate of model generalizability, was ~17% and ~27% for the two respective targets, while the CLSLR model, achieved a much lower CV error of less than 1%, on average, which suggests the existence of a non-linear relationship between host performance counters and target ISA. Although the average model prediction error is not reported, it is notably higher than the reported CV error. Due to the one-time counter collection, Lasso accuracy suffers due to a relatively limited number of data points.

A subsequent extension to LACross [12] used a compiler to instrument program basic blocks, which exposed program phases at a much finer granularity than end-to-end execution. The user specifies the phase granularity: finer granularity increases the profiling overhead, but improves accuracy; lower granularity is faster, but has lower accuracy. Program counter values are collected once per phase, and each phase is treated as a data point. This work also employs an Intel Core-i7 920 as a host and ARM Cortex-series processors targets, representing not only cross-ISA prediction, but also demonstrating the ability to predict the performance and power of embedded targets using a desktop CPU as a host. The average error reported was less than 10% in each case.

The viability of cross-platform performance prediction has profound implications for future CPU design methodologies. At present, architectural DSE necessitates the use of a simulator to characterize workload performance and/or power consumption at each design point. High simulation execution times limits both the number of design points that can be explored, and the number of workloads that could be used to characterize each design point. A cross-platform predictive model which uses a commercially available CPU as the host and targets a simulator could significantly increase the throughput of next-generation DSE processes. Another practical consideration is that analytical models require expert knowledge to design, while purely statistical models, such as those employed by LACross, can be derived automatically.

Like the cross-generational HALWPE (Chapter 2) and cross-platform HLSPredict (Chapter 4), the predictive features are performance counter measurements obtained from direct execution, which lends credence to our approach. RastSim, a cross-abstraction model approach uses simulator features created in the WCF and has the potential to use equivalent or higher quality of features, without the need to overcome ISA differences.

## 5.2 GPU Models

Predictive modeling for GPUs is largely inspired by CPU approaches. Predictive models for GPUs based on linear regression [74] decision trees [70], RFs [52] and ANNs [75] can accurately predict performance and power consumption for the GPUs on which they were trained. These techniques are primarily leveraged for DSE wherein models are trained to avoid exhaustive simulation on the target platform. Table 5:2 lists and compares statistical models for GPUs. While we omit explicit discussion of our own work in the Related works, we list them in the table for easy comparison to other cross-architecture methods.

### 5.2.1 Statistical Models for GPUs

One such example is a paper that uses ANNs to predict performance and power of OpenCL applications as architectural parameters of the GPU target scale [59], thereby avoiding a more costly exhaustive enumeration. Architectural parameters that are considered include core frequency, memory bandwidth, and the number of available *compute units* (CUs) with 448 different possible configurations. The model clusters kernels with similar scaling behavior, a-priori, via k-means clustering; the scaling trend is a model that predicts the performance and power consumption of a workload from hardware performance counter measurements. An ANN classifier is trained to predict the cluster of scaling trends that a previously unseen workload most closely matches; given the classification, the scaling trend predicts the performance and power consumption of that workload. The new workload is simulated using one parameter combination to obtain a baseline that can be scaled according to its ANN-predicted trend, thereby avoiding exhaustive simulation for each workload.

Our GPU oriented frameworks (Chapters 2 and 3) differ from the ANN predictor in several respects. First, HALWPE achieves cross-generation performance prediction, while the ANN predictor is limited to variants of the current-generation (host) GPU with three degrees of freedom. Chapter 3, functional simulation-based prediction is also limited to prediction on the same host but achieves cross-abstraction prediction wherein a GPU can be evaluated in advance of manufacturing, perhaps during pre-silicon design. Using the same host also has the distinct benefit of leveraged counters having been impacted by all architectural features available and exercised in the design. Second, no approach we present in this manuscript required modification of the firmware. This enables the usage of production drivers and software. In its favor, the ANN predicts performance and power consumption, while our approaches are presently limited to CPF performance prediction. Our evaluation of HALWPE focuses on graphics and gaming workloads,

whereas the ANN predictor was evaluated using OpenCL workloads spanning several application domains.

Zhang et al. [76] create a modeling framework to predict the performance of *existing* ATI GPUs to understand the relationship between program behavior, GPU performance and power consumption. They leverage these predictions to provide insights to programmers so that they can utilize and improve software design practices to maximize device performance. Our interest, in contrast, is to build a predictive regression model that extends functional simulation to provide pre-silicon GPU architectural performance estimates. The key similarity between our works and Zhang et al.'s is that we both employ RF regression and RSS-based feature ranking.

Gerum et al. [77] predict the performance of a GTX480 GPU, simulated using GPGPUSim, using a combination of source-level simulation, static analysis, and direct execution of instrumented source code. The code is fist statically analyzed and compiled using a Clang derivative to generate the Nvidia Parallel Thread Execution (PTX) code from the OpenCL workloads. From there, performance counter measurements obtained from the hardware execution are input to an analytical model, which predicts performance at native execution speeds. All but two workloads are modeled with more than 80% accuracy with the others around 60%-70%, for a rough average of 80%. The analytical models require a-priori knowledge of performance indicators as a precursor to model construction. Predictive models, in contrast, do not require this information, although do require the usage of a cycle-accurate GPU simulator to provide golden reference values during training.

### 5.2.2 Cross-Architecture Models for GPUs

XAPP [78] is a suite of cross-platform models that predict the degree of speedup or slowdown that would result from porting a C/C++ CPU workload to CUDA and executing it on a GPU. XAPP instruments program binaries to produce a set of microarchitecturally independent

features, which are selected to represent characteristics that correlate with typical GPU execution behavior, and, by extension, performance. Workload characteristics are measured using MICA [79] or PIN [80] and capture characteristics such as instruction level parallelism, shared memory bandwidth, memory throughput, memory coalescing, and bank conflicts in shared memory, among others. XAPP collects 17 features in total, which are converted into a training set ensemble via random bootstrap sampling with replacement [81].

| Paper | Model | Features | Accuracy (avg.) | Speed | Benefit | Drawback |
|---|---|---|---|---|---|---|
| [59] | K-means clustering and ANNs | Performance counters | Performance: ~85%<br>Power: ~90% | 20% of target executions eliminated | Power and perf. models used to avoid executing all design points. | Large percentage of design points required to train model |
| [76] | RF regression | Performance counters | Performance: ~86.9%<br>Power: ~95.7% | Near Native hardware execution | Model Perforamnce feature ranking to gleam insight into useful features | No speedup or design benefits. Only an insight tool for fixed GPU. |
| [77] | Analytical model | Source-level instrumentation and analysis | Performance: ~80%, no average given. | 2-5 orders of magnitude speedup over GPGPUSim. | Provides speedup over traditional functional and cycle-accurate simulation | Analytical model requires expert design, and may not port easily to other GPUs. |
| [78] | Forward stepwise regression | Workload characteristics that expose inherent GPU-compatible parallelism | Kepler Performance: ~64%<br>Maxwell Performance: ~73% | Static analysis incurs 10x-20x slowdown over native hardware | **Cross-platform;** Predict GPU performance from CPU features | High model error. |
| [82] | Analytical model | Single-threaded CPU memory and computation traces and latencies | Jetsen TK1 Performance: ~ 91% | CPU execution plus code instrumentation and PTX transform time | **Cross-platform;** Estimate GPU performance from CPU C code.. | Input sizes must be chosen to limit insturmetation overhead. |
| [83] | RF regression | Functional simulation execution statistics | Intel Skylake GPU Performance:~85.7% | ~328x faster than cycle-accurate simulation | **Cross-abstracton;** Host and target use same software stack. | Functional simulatior instrumetation overhead |
| [16] | OLS with feature selection and RF regression | Prev. generation performance counters and API metrics | Boradwell GT2/GT3 Performance: ~93%<br>Skylake GT3 Performance: ~91% | 29,000x - 44,000x faster than cycle-accurate simulation | **Cross-generation;** Predict pre-silicon next-generation GPU performance. | Cannot directly account for large-scale architectural. changes |

Table 5:2 Comparison of statistical models for GPUs.
We compare the approach, accuracy, benefits and drawbacks of predictive GPU models.

For each training set, XAPP trains a least-squares regression model that includes higher-order polynomial terms to capture non-linear relationships. For a new workload, the ensemble model reports the mean prediction of all models as the final predicted value. XAPP reported 36% average performance prediction error on a Nvidia Kepler GTX 660Ti, and 27% average performance prediction error on a Maxwell GTX 750. In contrast, our GPU frameworks focus on early-stage GPU architectural performance estimatio within an architecture family and early performance feedback for graphics software development.

CGPredict [82] is another cross-platform model that collects features from single-threaded non-optimized C code to predict the performance of Compute Unified Device Architecure (CUDA) code running on an embedded GPU, such as the Jetson TK1 Kepler. CGPredict employs an analytical model whose primary characteristics are based on the interaction between microarchitectural parallelism and memory access latencies and parallelism. CGPredict instruments source code via the compiler, and collects computation and memory traces, which are transformed during a memory behavior analysis stage, converting single-threaded CPU memory accesses to reflect GPU memory accesses and cache configurations. A subsequent computational analysis converts the computation trace to PTX format, which is specific to Nvidia GPUs. These transformed streams are coupled with estimated GPU computation and cache access latencies, which were derived from repeated execution of micro-benchmarks. A comprehensive analytical model predicts performance from the modified streams, achieving a relatively low predicted error of 9% across 15 kernels. The data input size for each benchmark must be carefully chosen to avoid excessive instrumentation and transformation latencies, while remaining long enough to realistically stress the GPU's compute and memory resources. While CGPredict aims to utilize CPU code to predict the performance of the same code when executed on GPGPUs, our work instead predicts 3D rendering workload performance and characterizes pre-silicon designs. Future

hopes to leverage our cross-architecture modeling techniques on GPGPUs to analyze pre-slicon GPGPU design. CGPredict suggests such approaches should lead to good results.

## 5.3 Statistical Models for FPGAs

HLS tools [6, 7] improve productivity over RTL implementation, but must search large design space and suffer from lengthy run times. In contrast, predictive modeling has a lengthy model construction phase, but the model can be used rapidly once deployed, e.g., during DSE. To our knowledge, HLSPredict is the first work to perform cross-architecture performance prediction targeting FPGAs, and for this reason we only compare against other predictive modelling approaches for FPGA design. These approaches are typically used in the context of DSE, and most target HLS design tasks. Table 5:3 summarizes the characteristics of several FPGA-based predictive modeling approaches, in the context of larger DSE frameworks and weighs their benefits and drawbacks. FPGAs are often used as acceleration engines for parallel and streaming workloads.

| Paper | Model | Features | Accuracy (avg.) | Speed | Benefit | Drawback |
|---|---|---|---|---|---|---|
| [84] | RF regression | HLS design details | >99% ADRS from Pareto-optima. | 2-4x fewer HLS runs required. | Avoids exhaustive enumeration of HLS design space | Requires repeated HLS calls to improve model accuracy, limiting speedup |
| [85] | Performance: Analytical Utilization: Hybrid Analytical + ANNs | Architectural template parameters that capture parallelism | Performance: ~94% Utilization: 88%-95% | 279x - 6333x faster than Vivado HLS. | Faster DSE by replacing HLS with predictive models | Requires HLS to characterize target FPGA; requires non-traditional HLS flow |
| [86] | Analytical | Source-level instrumentation and HLS simulation | Performance: ~99% (compute-bound workloads); ~95% (memory-bound workloads) | ~2 orders of magniute faster than target FPGA bitsream generation | Highly accurate; avoids HLS and target FPGA bitstream generation | Requires HLS simulation and board characterization; Vivado simulation is a performance bottleneck. |
| [87] | Performance: Anaytical Utilization: Gradient Boosted Machine model | HLS directives and workload features from compiler instrumentation. | Performance: ~88% Utilization: 81-87% | 2 - 3 orders of magnitude faster than target FPGA bitstream generation | Avoids HLS and target FPGA bitstream generation after model training and target FPGA device characterization | Requires custom microbenchmarks to characterize FPGA resource characeristics; higher instrumentation overhead than other approaches |

Table 5:3 Comparison of statistical models for FPGAs.
We compare the approach, accuracy, benefits and drawbacks of predictive FPGA models used for DSE.

FPGA accelerator design methodologies have traditionally been based on RTL-centric hardware design, and has more recently transitioned to HLS [6, 7]. Although HLS aims to be fully automatic, the typical designer is tasked with the problem of finding the correct combination of parameters (e.g., unroll factor, pipelining depth, etc.), which is a form of DSE. Due to long HLS times, direct evaluation of each design point is infeasible, which necessitates a turn toward modeling.

Liu et al. [84] explore predictive modeling for DSE via *Transductive Experimental Design* (TED) [88], which identifies and samples representative microarchitectural design points. These training sets are then used to build an RF regression model which is iteratively refined via repeated training and synthesis of additional directive permutations, wherein the training sets are updated after evaluating the prior trained model's accuracy. When used in conjunction with TED, iterative refinement improves prediction accuracy and identifies the *Pareto Optimal* set of design points, as measured using the *average distance from reference set* (ADRS) [89]. The user can specify an HLS budget (the maximum number of HLS synthesis run) and report Pareto Optimal design points using up 20 training workloads for budgets of up to 50 runs, and as few as 10 workloads for larger budgets up to 120 runs; this represents a reduction of more than half of the 242 total directive combinations in the search space. This model requires HLS-in-the-loop due to the feature choice (post-HLS design specifications) and iterative improvement techniques (HLS is used to verify performance improvement).

Koeplinger et al. [85] present a DSE methodology that repeatedly calls a bespoke HLS tool with inherent predictive modeling capabilities. Their models are created from C/C++ descriptions which the programmer has annotated with pragmas to indicate design patterns such as map, reduce, filter and groupBy [90]. The framework analyzes the annotated source code and performs transformations such as loop fusion and tiling and converts the program to a more precise specification called *Delite Hardware Definition Language* (DHDL). The DHDL specification is

converted to a set of parameterizable architectural templates, which account for the FPGA's on-chip resources (LUTs, BRAMs, routing, etc.) and off-chip memory bandwidth. Template parameters determine tiling sizes, parallelization factors, and coarse-grain pipelining, which are used alongside cycle-count and area utilization estimators to identify the Pareto Front of many design points. To construct the estimators, it's necessary to first characterize the board's various runtime latencies and resource availability using full synthesis runs, averaging 6 runs per template across several parameter configurations; this yields analytical models for utilization and cycle estimation for each template. The models are trained using ANNs with 200 design samples to compensate for the DHDL models, which do not capture on-board utilization. For six workloads consisting of millions of possible design points, they achieve average cycle estimation error of 6.1% and Look Up Table (LUT), DSP, and BRAM utilization estimates with 4.8%, 7.5% and 12.3% error respectively, while running 6,533x faster than DSE using Vivado HLS alone. While expedient, this technique requires extensive source code instrumentation, and is only compatible with applications that use standard design patterns.

HLScope+ [86] uses C code instrumentation and analytical modeling to improve the accuracy of the Vivado HLS simulator, adding the capability to handle input-dependent loop bounds. The instrumentation includes *dependency analysis*, which identifies independent regions of code to parallelize and *loop analysis*, which estimates loop cycle counts, culminating in execution cycle counts and DRAM transaction counts for each code module. This helps HLScope+ identify the application's critical execution path, from which it estimates the per-module stall rate. To compensate for the inaccuracy inherent to static analysis, loop analysis is extended with an analytical model.

HLScope+ accounts for DRAM bandwidth and latency by creating a high-level external memory module, which abstracts away individual memory accesses while accounting for resource

contention among *processing elements* (PEs); this model is more accurate than the Vivado HLS simulator, which optimistically assumes that memory can be fetched each cycle and ignores memory bandwidth and contention among PEs. An analytical model is created to predict memory access time and can be combined with the compute cycle count to estimate execution time. HLScope+ is evaluated using 14 HLS workloads, resulting in a 1.1% average error for compute-bound workloads and 5.0% average error for memory bound workloads. This is much faster than fully synthesizing each workload; however, the key drawback is that HLScope+ employs analytical models which require expert knowledge of the target FPGA and its memory interface and are not transferrable from one target to another.

MPSeeker [87] performs DSE using HLS simulation in conjunction with C/C++ source code instrumented at the Low Level Virtual Machine (LLVM) Intermediate Representation (IR) level, HLS design directives, and both predictive and analytical modeling techniques. MPSeeker's DSE considers several design directives including tile sizes, the number of PEs, loop unrolling, loop pipelining, and array partitioning. MPSeeker extends a single-PE analytical model [91] with the ability to estimate cycle counts for multi-PE designs that encompass coarse-grained parallelism. MPSeeker also includes an ensemble tree predictor called the *Gradient Boosted Machine* (GBM) [92], which uses 14 program features and user-defined parallelism directives to predict FPGA resource usage.

MPSeeker's profiler accepts C source code (tiled nested loop structure), FPGA resource constraints, and user-supplied directive settings (tile size, loop unrolling, pipelining, and array partitioning) to produce features. The features track key workload characteristics correlated to parallelism and memory subsystem behavior and are input to Lin-Analyzer to predict performance and the GBM model to predict resource utilization. A subsequent analytical equation combines the

outputs of the two models to account for further FPGA resource restrictions, which limit the number of PEs that can execute concurrently.

MPSeeker uses 10 microbenchmark kernels to ascertain the degree of loop resource consumption, communication interface behavior, and other similar properties, along with 5 larger benchmarks for evaluation; each kernel has 280 unique design configurations. MPSeeker's DSE procedure achieves 90% of the Pareto optimal performance, while running ~421x to ~4308x faster than FPGA bitstream generation. Lin-Analyzer reports an average error of 12.8%, while the GBM model reports average errors of 13.2%, 14.7%, 12.7%, and 19.4% for LUTs, flip-flops, DSP blocks, and BRAMs respectively. The reliance on expert-developed microbenchmarks to characterize FPGA resource constraints and the necessity to execute instrumented source code on the simulator are notable drawbacks of this approach.

Cross-architecture prediction (e.g., CPU host to FPGA target) could overcome many of these shortcomings. Doing so would allow for the creation of automatically derived statistical regression models that automatically select requisite features from direct execution on the host, eliminating the need for expert guidance and increasing portability. Given that cross-platform predictive models for both CPUs and GPUs has been successful, they can and should be applied to DSE for FPGAs as well. For this reason, we have produced and submitted for peer review HLSPredict of Chapter 4.

# Chapter 6 Conclusion

## 6.1 Cross-Generation Concluding Remarks

Chapter2 details the HALWPE cross-generation integrated GPU performance estimation framework has established the feasibility of cross-generation GPU CPF prediction using performance counter readings, DirectX metrics, and hardware queries. HALWPE achieved high accuracy when predicting across single and multiple generation deltas, and in the presence of significant slice scaling and software modifications. HALWPE predicted the CPF of a Broadwell GT2 GPU, a Broadwell GT3 GPU and a Skylake GT3 GPU with $E_{outs}$ of 7.45%, 7.47%, and 8.91% and speedups of 29481x, 43643x, and 44214x respectively. In addition, our simulator models demonstrate the utility of cross-generation GPU performance estimation in the context of driver scaling only, and compute parallelism scaling only. Our results and analysis suggest that predictive modeling can aid early-stage microarchitecture DSE and may be able to help with identification of performance bottlenecks; however, predictive modeling must be applied with care, as the models themselves are inherently finicky.

## 6.2 Cross-Abstraction Concluding Remarks

Chapter 3 details the functional simulator based, cross-abstraction, integrated GPU performance prediction framework. This work has further demonstrated the benefits of cross-architecture statistical predictive modeling by leveraging a functional GPU simulation extended with predictive regression models to accurately predict GPU performance. It also demonstrates the utility of this this approach when applied during pre-silicon DSE. Our experiments, which focus on an Intel Skylake GT3 GPU, achieve an out-of-sample-error rate of 14.3% while running three to four orders of magnitude faster than cycle-accurate simulation, in large part due to the WCF simulator extensions. In addition to moving co-optimization of GPU hardware and software to earlier design stages, this approach could provide additional benefits, such as early-stage driver

conformance testing [93]. It may also be possible to distribute the framework, and a trained model, as a pre-silicon evaluation platform for 3rd party vendors to assess workload performance when integrated into a larger system. Feature ranking can help GPU architects to identify performance bottlenecks on representative workloads as early as possible.

These models can be generally applied to any GPU that supports hardware acceleration for 3D DirectX rendering workloads, as the bulk of our counters specifically address render actions and units that must be performed and supported in the architecture to render these applications functionally. For separate workloads that do not exercise units intended to support the DirectX pipeline such as media workloads like video streaming, or video codec processing, or *General Purpose Graphical Processing Unit* (GPGPU) tasks we do not believe our models are generally applicable and would require a modified instance of the WCF to target the correct pipeline metrics. It is also very common for these alternative workloads to exercise different hardware units, as current GPUs used large amounts of FF hardware, consequently following a different execution pipeline in the GPU. Our training set reflects this assumption.

## 6.3 Cross-Platform Concluding Remarks

Chapter 4 discusses HLSPredict, our cross-platform FPGA accelerator modeling framework. The work generates predictive models that use CPU performance counters as model features to predict the performance and power of FPGA accelerators designed using HLS. This eliminates the need to construct bespoke analytical models for the target and the need to employ variants of HLS-in-the-loop DSE, which has been the primary method for speeding up the evaluation process when compared to exhaustive synthesis of the design space. Although model training is an overhead, trained models can be applied at near-native CPU execution speeds, obtaining a speedup of 36.24x for the baseline workloads and 43.78x for the optimized workloads

compared to HLS alone. HLSPredict achieved sufficiently low $E_{out}$ of 9.08% for baseline workloads, and 10.95% for optimized workloads.

## 6.4 Future Work

While each of our frameworks has demonstrated that cross-architecture predictive modeling in each of the three forms presented here can readily be used to make power and performance estimations before fabricating GPUs or before synthesizing FPGAs, the technical approach is still relatively new. To mature the approach and spur its adoption, it is necessary to advance and further evaluate the research by extending the current frameworks to answer several remaining open questions.

For example: Can either HALWPE, or the functional simulator-based GPU modeling approaches predict power/energy consumption with equal effectiveness? For the frameworks to be extended, model retraining with the proper target metric, e.g. total power or dynamic power would be required. Further, the target simulator would need to be inspected to determine if enabling the power metric is possible, or if a more fundamental change like extending the simulator to model power, or even using a different power-oriented simulator will be necessary. Finally, the effective features for prediction would likely change, requiring additional analysis, and in the case of the functional simulator approach, the WCF may need to be extended or rearchitected to obtain useful features.

Can these GPU models be re-used to predict the CPF of similar targets without retraining, and if not, what level of change necessitates retraining? Criteria for model retraining/reuse would be beneficial and will need to be evaluated on a case by case basis for each framework. In the case of Chapter 2, HALWPE we demonstrate that a modicum of compute parallelism scaling and software driver scaling might result in accurate predictions without model retraining or feature

changes. These studies will need to be extended to identify when this no longer holds, and then performed on our remaining frameworks for GPUs and FPGAs.

Can our GPU frameworks generalize to other GPU vendors such as Nvidia or AMD. To evaluate this, academic or industrial simulators that target recent GPU advances from these vendors will be needed. Further the simulator will need to model a GPU device of the same family for which a previously manufactured product might be available. The instrumentation of the host and targets would need to be redone, and models retrained, effectively reimplementing the framework for the new platforms.

Can the GPU frameworks also generalize to rendering workloads that utilize different APIs, such as OpenGL? In each iteration of our GPU frameworks we target only DirectX API workloads. To evaluate this, a high-quality set of OpenGL workloads would need to be acquired, representative traces sampled, and a GWL created. The Haswell GT2 host employed by HALWPE of Chapter 2 will need to target non-DirectX counters, instead shifting the emphasis to the OpenGL pipeline hardware instead. RastSim will require similar modifications in the abstracted functional simulator host.

Further, can these GPU oriented models generalize to GPGPU workloads written in languages such as CUDA or OpenCL? In this case, the simulators, would need to modified, or different simulators leveraged, which support the GPGPU programming pipeline. Further, the instrumentation of the host, be it the commercial GPU in Chapter 2, or the functional simulator in Chapter 3 would need to be completely re-worked to target features that are relevant to the different compute model, and new models trained.

Can these approaches be leveraged and applied to compute-intensive CPU workloads? As outlined in the related works, we have seen cross-platform (e.g. predictions from one CPU to another) perform well. It would be valuable to utilize our cross-generation and cross-abstraction

approaches for modern out-of-order CPUs to determine if the performance benefits and high accuracy can be maintained, while exploring additional pre-silicon use cases.

HLSPredict of Chapter 4 currently predicts FPGA cycle count as well as several power metrics. Can the HLSPredict be extended or modified to also predict various utilization percentages of LUTs, LUT RAMs, DSP blocks, and more. Additional workload information, in the form of new counters not encompassed by the current set of CPU performance counters would be required. One potential avenue is to leverage static source code analysis to produce those new features relevant to utilization such as loop trip counts, inter-loop dependency analysis, etc.

Can HLSPredict be further optimized to reduce the performance counter collection overhead without sacrificing accuracy, to provide increased speedup? Currently. several counters are required, necessitating repeated executions of the same data point on the host to collect all necessary counters. More aggressive feature selection may reduce the number of repeated runs, reducing host execution overhead, thereby increasing the speedup of the approach.

Finally, can HLSPredict's prediction methodology be integrated into and used as the basis of follow up studies that target DSE using HLS tools. HLSPredict will need to be extended to automatically perform DSE on the FPGA target to collect training samples. Further, as the design parameters, in the form of Vivado HLS directives are modified, the predictive model will need to be made aware of this. Approaches exist to automatically cluster and scale workload predictions based on similar performance counter observations. Other approaches might include integrating in tool provided (in the case of training) or user provided (in the case of model application) features that indicate the directive settings.

## 6.5 Closing Remarks

Predictive modeling has immense potential but has not yet been fully integrated into the CPU or GPU architectural design process; on the other hand, predictive models are widely used in

design exploration for FPGAs, but often rely on repeated calls to HLS tools and/or non-portable analytical modeling efforts. On the CPU/GPU side, the next step is to evaluate the potential of cross-platform modeling as an alternative to cycle-accurate simulation, especially during early design stages, by demonstrating accurate predictions of architectural modifications. In contrast, cross-platform models for FPGAs have not yet been demonstrated; however, once this is accomplished, DSE tools that target FPGAs will benefit significantly. Long-term, there is still a considerable amount of work to be done to convince the research community that predictive models are more than an intellectually meritorious curiosity and have the capability to significantly increase designer engineering productivity.

# REFERENCES


[1] Q. Guo, T. Chen, Y. Chen, and F. Franchetti, "Accelerating Architectural Simulation Via Statistical Techniques: A Survey," *IEEE Trans. Comput. Des. Integr. Circuits Syst.*, vol. 35, no. 3, pp. 433–446, Mar. 2016.

[2] J. E. Miller *et al.*, "Graphite: A distributed parallel simulator for multicores," in *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, 2010, pp. 1–12.

[3] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation," in *SC '11 Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011, p. 1.

[4] D. Sanchez and C. Kozyrakis, "ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-Core Systems," in *Proceedings of the 40th Annual International Symposium on Computer Architecture - ISCA '13*, 2013, vol. 41, no. 3, pp. 475–486.

[5] M. Pellauer, M. Adler, M. Kinsy, A. Parashar, and J. Emer, "HAsim: FPGA-based high-detail multicore simulation using time-division multiplexing," in *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, 2011, pp. 406–417.

[6] A. Canis *et al.*, "LegUp : High-Level Synthesis for FPGA-Based Processor / Accelerator Systems," *FPGA '11 Proc. 19th ACM/SIGDA Int. Symp. F. Program. gate arrays*, pp. 33–36, 2011.

[7] Xilinx, "Vivado High-Level Synthesis." [Online]. Available: https://goo.gl/2kpNwy. [Accessed: 30-Mar-2018].

[8] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe, "SMARTS: accelerating microarchitecture simulation via rigorous statistical sampling," in *30th Annual International Symposium on Computer Architecture, 2003. Proceedings.*, pp. 84–95.

[9] E. Perelman *et al.*, "Using SimPoint for accurate and efficient simulation," in *Proceedings of the 2003 ACM SIGMETRICS international conference on Measurement and modeling of computer systems - SIGMETRICS '03*, 2003, vol. 31, no. 1, p. 318.

[10] K. O'Neal, M. Liu, H. Tang, A. Kalantar, and P. Brisk, "HLSPredict: Cross Platform

Performance Prediction for FPGA High-Level Synthesis," in *Submitted For Publication*, 2018.

[11] D. Marculescu, A. S. I. G. on D. Automation, K. ACM Digital Library., Y. Hoskote, L. K. John, and A. Gerstlauer, "Learning-Based Power modeling of System-Level Black-Box IPs," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, 2015, pp. 847–853.

[12] X. Zheng, L. K. John, and A. Gerstlauer, "Accurate phase-level cross-platform power and performance estimation," in *Proceedings of the 53rd Annual Design Automation Conference on - DAC '16*, 2016, pp. 1–6.

[13] R. Kohavi, *A study of cross-validation and bootstrap for accuracy estimation and model selection*. International Joint Conferences on Artificial Intelligence, Inc., 1995.

[14] K. O'Neal and P. Brisk, "Predictive Modeling for CPU, GPU, and FPGA Performance and Power Consumption: A Survey," in *IEEE Computer Society Annual Symposium on VLSI (ISVLSI) '18.*

[15] K. O'Neal, P. Brisk, E. Shriver, and M. Kishinevsky, "Hardware-Assisted Cross-Generation Prediction of GPUs Under Design," *IEEE Trans. Comput. Des. Integr. Circuits Syst.*, pp. 1–1, 2018.

[16] K. O'Neal, P. Brisk, E. Shriver, and M. Kishinevsky, "HALWPE: Hardware-Assisted Light Weight Performance Estimation for GPUs," in *Proceedings of the 54th Annual Design Automation Conference 2017 on - DAC '17*, 2017, pp. 1–6.

[17] Intel Corporation, "Open Source Intel ® HD Graphics Programmer's Reference Manual (PRM)." [Online]. Available: https://goo.gl/9KSJks. [Accessed: 17-May-2018].

[18] Intel Corporation, "Intel GPA GPU Metrics." [Online]. Available: https://goo.gl/3Ird2x. [Accessed: 17-May-2018].

[19] Microsoft Corporation, "New Resource Types Direct3D 11." [Online]. Available: https://goo.gl/gX4MZz. [Accessed: 17-May-2018].

[20] Microsoft Corporation, "Introduction to a Device in Direct3D 11." [Online]. Available: https://goo.gl/bi6USV. [Accessed: 17-May-2018].

[21] Microsoft Corporation, "IDirect3DQuery9 interface." [Online]. Available: https://goo.gl/UZDyRD. [Accessed: 17-May-2018].

[22] Microsoft Corporation, "IDirect3DIndexBuffer9 interface." [Online]. Available: https://goo.gl/flRm1A. [Accessed: 17-May-2018].

[23] Microsoft Corporation, "IDirect3DDevice9 interface." [Online]. Available: https://goo.gl/HCSNTw. [Accessed: 17-May-2018].

[24] Microsoft Corporation, "ID3D11Device interface." [Online]. Available: https://goo.gl/gzeD0o. [Accessed: 17-May-2018].

[25] Microsoft Corporation, "ID3D11DeviceContext interface." [Online]. Available: https://goo.gl/4VnmIj. [Accessed: 17-May-2018].

[26] Microsoft Corporation, "ID3D10Device interface." [Online]. Available: https://goo.gl/4YNKpl. [Accessed: 17-May-2018].

[27] Intel Corporation, "The Compute Architecture of Intel ® Processor Graphics Gen9." [Online]. Available: https://goo.gl/RMmUc6. [Accessed: 17-May-2018].

[28] Intel Corporation, "The Compute Architecture of Intel® Processor Graphics Gen8," 2014. [Online]. Available: https://goo.gl/TnpAGc. [Accessed: 17-May-2018].

[29] Intel Corporation, "The Compute Architecture of Intel® Processor Graphics Gen7.5." [Online]. Available: https://goo.gl/5HZ54v. [Accessed: 17-May-2018].

[30] H. Akaike, "A New Look at the Statistical Model Identification," *IEEE Trans. Automat. Contr.*, vol. 19, no. 6, pp. 716–723, Dec. 1974.

[31] H. Akaike, “Information Theory and an Extension of the Maximum Likelihood Principle,” Springer, New York, NY, 1992, pp. 610–624.
[32] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning*. New York, 2001.
[33] C. L. Lawson and R. J. Hanson, *Solving Least Squares Problems*. Philadelphia: SIAM, 1995.
[34] R. R. Hocking, “The Analysis and Selection of Variables in Linear Regression,” *Biometrics*, vol. 32, no. 1, p. 1, Mar. 1976.
[35] R. Tibshirani, “Regression Shrinkage and Selection via the Lasso,” *J. R. Stat. Soc. Ser. B*, vol. 58, pp. 267–288, 1996.
[36] L. Breiman and Leo, “Random Forests,” *Mach. Learn.*, vol. 45, no. 1, pp. 5–32, 2001.
[37] C. Leys, C. Ley, O. Klein, P. Bernard, and L. Licata, “Detecting outliers: Do not use standard deviation around the mean, use absolute deviation around the median,” *J. Exp. Soc. Psychol.*, vol. 49, no. 4, pp. 764–766, Jul. 2013.
[38] K. O’Neal, P. Brisk, A. Abousamra, Z. Waters, and E. Shriver, “GPU Performance Estimation using Software Rasterization and Machine Learning,” *ACM Trans. Embed. Comput. Syst.*, vol. 16, no. 5s, pp. 1–21, Sep. 2017.
[39] “OpenSWR — Gallium 0.4 documentation.” [Online]. Available: http://gallium.readthedocs.io/en/latest/drivers/openswr.html. [Accessed: 30-Apr-2018].
[40] H. Zou and T. Hastie, “Regularization and Variable Selection via the Elastic Net,” *J. R. Stat. Soc. Ser. B (Statistical Methodol.*, vol. 67, pp. 301–320, 2005.
[41] A. E. Hoerl and R. W. Kennard, “Ridge Regression: Applications to Nonorthogonal Problems,” *Technometrics*, vol. 12, no. 1, pp. 69–82, Feb. 1970.
[42] L. Breiman, *Classification and Regression Trees*. Routledge, 2017.
[43] A. Cutler, “Trees and Random Forests,” 2013. [Online]. Available: https://goo.gl/ZWyQYY.
[44] Louis-Noël Pouchet, “PolyBench/C.” [Online]. Available: https://goo.gl/2eNA9L. [Accessed: 28-Mar-2018].
[45] Louis-Noël Pouchet, “PolyBench/GPU.” [Online]. Available: https://goo.gl/7PTHSJ. [Accessed: 28-Mar-2018].
[46] J. Treibig, G. Hager, and G. Wellein, “LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments,” in *Proceedings of the International Conference on Parallel Processing Workshops*, 2010, pp. 207–216.
[47] LIKWID, “RRZE-HPC/likwid: Performance monitoring and benchmarking suite.” [Online]. Available: https://goo.gl/Wyhur3. [Accessed: 28-Mar-2018].
[48] F. Pedregosa *et al.*, “Scikit-learn: Machine Learning in Python,” *J. Mach. Learn. Res.*, vol. 12, no. Oct, pp. 2825–2830, 2011.
[49] I. Guyon, J. Weston, S. Barnhill, and V. Vapnik, “Gene selection for cancer classification using support vector machines,” *Mach. Learn.*, vol. 46, no. 1–3, pp. 389–422, 2002.
[50] Vivado, “Vivado Design Suite User Guide: System-Level Design Entry,” vol. 901, pp. 1–120, 2013.
[51] LIKWID, “RRZE-HPC/likwid Haswell Performance Groups.” [Online]. Available: https://goo.gl/jQEuZG. [Accessed: 28-Mar-2018].
[52] J. Chen, B. Li, Y. Zhang, L. Peng, and J. K. Peir, “Tree structured analysis on GPU power study,” in *Proceedings - IEEE International Conference on Computer Design: VLSI in Computers and Processors*, 2011, pp. 57–64.

[53] A. Fog, “The microarchitecture of Intel, AMD and VIA CPUs An optimization guide for assembly programmers and compiler makers,” 2018.
[54] T. Sherwood *et al.*, “Automatically characterizing large scale program behavior,” *ACM SIGARCH Comput. Archit. News*, vol. 30, no. 5, p. 45, Dec. 2002.
[55] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, “Analyzing CUDA workloads using a detailed GPU simulator,” in *ISPASS 2009 - International Symposium on Performance Analysis of Systems and Software*, 2009, pp. 163–174.
[56] V. M. del Barrio, C. Gonzalez, J. Roca, and A. Fernandez, “ATTILA: a cycle-level execution-driven simulator for modern GPU architectures,” *2006 IEEE Int. Symp. Perform. Anal. Syst. Softw.*, pp. 231–241, 2006.
[57] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli, “Multi2Sim: A simulation framework for CPU-GPU computing,” in *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2012, pp. 335–344.
[58] A. Gutierrez *et al.*, “Sources of error in full-system simulation,” in *ISPASS 2014 - IEEE International Symposium on Performance Analysis of Systems and Software*, 2014, pp. 13–22.
[59] G. Wu, J. L. Greathouse, A. Lyashevsky, N. Jayasena, and D. Chiou, “GPGPU performance and power estimation using machine learning,” in *Proceedings of HPCA 2015*, 2015, pp. 564–576.
[60] W. Jia, K. A. Shaw, and M. Martonosi, “Starchart: Hardware and software optimization using recursive partitioning regression trees,” *Proc. 22nd Int. Conf. Parallel Archit. Compil. Tech.*, pp. 257–267, Oct. 2013.
[61] Z. Yu *et al.*, “Accelerating GPGPU architecture simulation,” in *Proceedings of the ACM SIGMETRICS/international conference on Measurement and modeling of computer systems - SIGMETRICS ’13*, 2013, vol. 41, no. 1, p. 331.
[62] M. Badr and N. E. Jerger, “SynFull: Synthetic traffic models capturing cache coherent behaviour,” in *Proceedings - International Symposium on Computer Architecture*, 2014, pp. 109–120.
[63] J. Yin, O. Kayiran, M. Poremba, N. E. Jerger, and G. H. Loh, “Efficient synthetic traffic models for large, complex SoCs,” in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016, pp. 297–308.
[64] S. Lee and W. W. Ro, “Parallel GPU architecture simulation framework exploiting work allocation unit parallelism,” in *International Symposium on Performance Analysis of Systems and Software*, 2013, pp. 107–117.
[65] D. Chiou *et al.*, “FPGA-Accelerated Simulation Technologies (FAST): Fast, Full-System, Cycle-Accurate Simulators,” in *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, 2007, pp. 249–261.
[66] E. Ïpek, S. A. McKee, R. Caruana, B. R. de Supinski, and M. Schulz, “Efficiently exploring architectural design spaces via predictive modeling,” *ACM SIGPLAN Not.*, vol. 41, no. 11, p. 195, Oct. 2006.
[67] 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*, 2006, vol. 40, no. 5, p. 185.
[68] R. Nath, D. Carmean, and T. S. Rosing, “Power modeling and thermal management techniques for manycores,” in *2013 IEEE Symposium on Computers and Communications (ISCC)*, 2013, pp. 000740–000746.
[69] K. Skadron, M. R. Stan, K. Sankaranarayanan, W. Huang, S. Velusamy, and D. Tarjan,

"Temperature-aware microarchitecture: Modeling and Implementation," *ACM Trans. Archit. Code Optim.*, vol. 1, no. 1, pp. 94–125, Mar. 2004.

[70] X. Ma, M. Dong, L. Zhong, and Z. Deng, "Statistical power consumption analysis and modeling for GPU-based computing," in *Proceeding of SOSP Workshop on Power Aware Computing and Systems (HotPower)*, 2009.

[71] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith, "A mechanistic performance model for superscalar out-of-order processors," *ACM Trans. Comput. Syst.*, vol. 27, no. 2, pp. 1–37, May 2009.

[72] X. Zheng, P. Ravikumar, L. K. John, and A. Gerstlauer, "Learning-based analytical cross-platform performance prediction," in *2015 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, 2015, pp. 52–59.

[73] J. Shlens, "A Tutorial on Principal Component Analysis," Apr. 2014.

[74] P. E. Bailey, D. K. Lowenthal, V. Ravi, B. Rountree, M. Schulz, and B. R. De Supinski, "Adaptive configuration selection for power-constrained heterogeneous systems," in *Proceedings of the International Conference on Parallel Processing*, 2014, vol. 2014–Novem, no. November, pp. 371–380.

[75] S. Song, C. Su, B. Rountree, and K. W. Cameron, "A simplified and accurate model of power-performance efficiency on emergent GPU architectures," in *Proceedings - IEEE 27th International Parallel and Distributed Processing Symposium, IPDPS 2013*, 2013, pp. 673–686.

[76] Y. Zhang, Y. Hu, B. Li, and L. Peng, "Performance and power analysis of ATI GPU: A statistical approach," in *Proceedings - 6th IEEE International Conference on Networking, Architecture, and Storage, NAS 2011*, 2011, pp. 149–158.

[77] C. Gerum, O. Bringmann, and W. Rosenstiel, "Source Level Performance Simulation of GPU Cores," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2015*, 2015, pp. 217–222.

[78] N. Ardalani, C. Lestourgeon, K. Sankaralingam, and X. Zhu, "Cross-architecture performance prediction (XAPP) using CPU code to predict GPU performance," *Proc. 48th Int. Symp. Microarchitecture - MICRO-48*, pp. 725–737, 2015.

[79] K. Hoste and L. Eeckhout, "Comparing Benchmarks Using Key Microarchitecture-Independent Characteristics," in *2006 IEEE International Symposium on Workload Characterization*, 2006, pp. 83–92.

[80] C.-K. Luk *et al.*, "Pin: Building Customized Program Analysis Tools with Dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation - PLDI '05*, 2005, vol. 40, no. 6, p. 190.

[81] L. Breiman, "Bagging predictors," *Mach. Learn.*, vol. 24, no. 2, pp. 123–140, Aug. 1996.

[82] S. Wang, G. Zhong, and T. Mitra, "CGPredict: Embedded GPU Performance Estimation from Single-Threaded Applications," *ACM Trans. Embed. Comput. Syst.*, vol. 16, no. 5s, pp. 1–22, Sep. 2017.

[83] K. O'Neal, P. Brisk, A. Abousamra, Z. Waters, and E. Shriver, "GPU Performance Estimation using Software Rasterization and Machine Learning," *ACM Trans. Embed. Comput. Syst.*, vol. 16, no. 5s, pp. 1–21, Sep. 2017.

[84] H.-Y. Liu and L. P. Carloni, "On learning-based methods for design-space exploration with High-Level Synthesis," in *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2013, pp. 1–7.

[85] D. Koeplinger *et al.*, "Automatic generation of efficient accelerators for reconfigurable

hardware," in *ACM SIGARCH Computer Architecture News*, 2016, vol. 44, no. 3, pp. 115–127.

[86] Y. Choi, P. Zhang, P. Li, and J. Cong, "HLscope+: fast and accurate performance estimation for FPGA HLS," in *Proceedings of the 36th International Conference on Computer-Aided Design*, 2017, pp. 691–698.

[87] G. Zhong, A. Prakash, S. Wang, Y. Liang, T. Mitra, and S. Niar, "Design Space exploration of FPGA-based accelerators with multi-level parallelism," in *Proceedings of the 2017 Design, Automation and Test in Europe, DATE 2017*, 2017, pp. 1141–1146.

[88] K. Yu, J. Bi, and V. Tresp, "Active learning via transductive experimental design," in *Proceedings of the 23rd international conference on Machine learning - ICML '06*, 2006, pp. 1081–1088.

[89] G. Palermo, C. Silvano, and V. Zaccaria, "ReSPIR: A response surface-based pareto iterative refinement for application-specific design space exploration," *IEEE Trans. Comput. Des. Integr. Circuits Syst.*, vol. 28, no. 12, pp. 1816–1829, Dec. 2009.

[90] A. K. Sujeeth *et al.*, "OptiML: An implicitly parallel domain specific language for machine learning," in *IN PROCEEDINGS OF THE 28TH INTERNATIONAL CONFERENCE ON MACHINE LEARNING, SER. ICML*, 2011.

[91] G. Zhong, A. Prakash, Y. Liang, T. Mitra, and S. Niar, "Lin-analyzer: A high-level performance analysis tool for FPGA-based accelerators," in *Proceedings of the 53rd Annual Design Automation Conference on - DAC '16*, 2016, pp. 1–6.

[92] Click C. et al., "Gradient Boosting Machine (GBM) — H2O 3.18.0.8 documentation," 2016. [Online]. Available: https://goo.gl/viLQGw. [Accessed: 01-May-2018].

[93] Microsoft Corporation, "Windows Hardware Certification Kit User's Guide." [Online]. Available: https://goo.gl/s0TCzJ. [Accessed: 22-May-2018].

# LIST OF ABBREVIATIONS

| | |
|---|---|
| ACM | Association for Computing Machinery |
| ACM-ICPC | ACM International Collegiate Programming Contest |
| ACP | Accelerator Coherency Port |
| ADRS | Average Difference from Reference Set |
| AIC | Akaike Information Criterion |
| ANN | Artificial Neural Network |
| APE | Absolute Percentage Relative Error |
| API | Application Programming Interface |
| AXI | Application eXtensible Interface |
| BC | Barycentric Calculator |
| BIC | Bayesian Information Criterion |
| BIOS | Basic input-output system |
| BLT | Blitter |
| BRAM | Block Random Access Memory |
| CAS | Cycle-Accurate Simulator |
| CL | Clipper |
| CLSR | Constrained Locally Sparse Regression |
| CPF | Cycles Per Frame |
| CPU | Central Processing Unit |
| CS | Compute Shader |
| CV | Cross-Validation |
| DAPRC | Render Cache |
| DC | Data Cluster |
| DHDL | Delite Hardware Definition Language |
| DRAM | Dynamic Random Access Memory |
| DSE | Design Space Exploration |
| DSE | Design Space Exploration |
| DSP | Digital Signal Processor |
| DTM | Dynamic Thermal Management |
| Eout | Out-of-sample Error |
| EU | Execution Unit |
| FF | Fixed Function |
| FPGA | Field Programmable Gate Array |
| GA | Global Asset |
| GAM | Graphics Arbiter Model |
| GBM | Gradient Boosted Machines |
| GPGPU | General Purpose Graphical Processing Units |
| GPIO | General Purpose input-output |
| GPU | Graphical Processing Unit |
| GS | Geometry Shader |
| GTI | Graphics Translation Interface |
| GWL | Graphics Workload Library |
| HIZ | Hierarchical-Z |

| | |
|---|---|
| HLS | High-Level Synthesis |
| HS | Hull Shader |
| IA | Input Assembler |
| IC | Instruction Cache |
| IP | Intellectual Property |
| IR | Inlier Ratio |
| IR | Intermediate Representation |
| ISA | Instruction Set Architecture |
| IZ | Intermediate-Z |
| KIPS | Thousands of Instructions Per Second |
| KNF | Knights Ferry |
| LLC | Lower-level cache |
| LLVM | Low Level Virtual Machine |
| LUT | Look Up Table |
| MAPE | Mean Absolute Reltaive Percentage Error |
| MIPS | Millions of Instructions Per Second |
| MSAA | Multi-Sampling Anti-Aliasing |
| NNLS | Non- Negative Least Squares |
| OLS | Ordinary Least Squares |
| OS | Operating System |
| PC | Personal Computer |
| PCA | Principle Component Analysis |
| PE | Processing Elements |
| PS | Pixel Shader |
| PSD | Pixel Shader |
| PTX | Parallel Thread Execution |
| QOS | Quality of Service |
| RF | Random Forest |
| RFECV | Recursive Feature Elimination with Cross Validation |
| RSS | Residual Sum of Squares |
| RTL | Register Transfer Level |
| SD | Secure Digital |
| SDK | Software Development Kit |
| SOL | Stream Output Logic |
| SPI | Serial Peripheral Interface |
| SVM | State Variable Manager |
| TD | Thread Dispatch |
| TE | Tesselator |
| TED | Transductive Experimental Design |
| UAR | Uniforma at Random |
| VF | Vertex Fetch |
| VS | Vertex Shader |
| WCF | Workload Characterization Framework |
| WL | Workload Library |
| WM | Windower |