UC Riverside
UC Riverside Previously Published Works

Title
GPU Performance Estimation using Software Rasterization and Machine Learning

Permalink
https://escholarship.org/uc/item/0fb2j1d6

Journal
ACM Transactions on Embedded Computing Systems, 16(5s)

ISSN
1539-9087

Authors
O'neal, Kenneth
Brisk, Philip
Abousamra, Ahmed
et al.

Publication Date
2017-10-31

DOI
10.1145/3126557

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

GPU Performance Estimation using Software Rasterization and Machine Learning

KENNETH O'NEAL and PHILIP BRISK, University of California, Riverside
AHMED ABOUSAMRA, ZACK WATERS, and EMILY SHRIVER, Intel Corporation

This paper introduces a predictive modeling framework to estimate the performance of GPUs during pre-silicon design. Early-stage performance prediction is useful when simulation times impede development by rendering driver performance validation, API conformance testing and design space explorations infeasible. Our approach builds a Random Forest regression model to analyze DirectX 3D workload behavior when executed by a software rasterizer, which we have extended with a workload characterizer to collect further performance information via program counters. In addition to regression models, this work produces detailed feature rankings which can provide valuable architectural insight, and accurate performance estimates for an Intel integrated Skylake generation GPU. Our models achieve reasonable out-of-sample-error rates of 14%, with an average simulation speedup of 327x.

CCS Concepts: • **Hardware → Electronic design automation**;

Additional Key Words and Phrases: GPU simulation, predictive model, random forest regression

ACM Reference format:
Kenneth O'Neal, Philip Brisk, Ahmed Abousamra, Zack Waters, and Emily Shriver. 2017. GPU Performance Estimation using Software Rasterization and Machine Learning. *ACM Trans. Embed. Comput. Syst.* 16, 5s, Article 148 (September 2017), 21 pages.
https://doi.org/10.1145/3126557

1 INTRODUCTION

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 cycle-accurate performance simulators to explore the architectural design space. Industrial cycle accurate simulators, which are used both for performance characterization and post-silicon validation, require much higher accuracy than their academic counterparts, necessitating longer simulation times. 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

This article was presented in the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS) 2017 and appears as part of the ESWEEK-TECS special issue.
This work is supported by the National Science Foundation, under grant CCF-528181.
Authors' addresses: K. O'Neal, Winston Chung Hall, Room 464, University of California, Riverside, Riverside, CA 92521; P. Brisk, Winston Chung Hall, Room 339, University of California, Riverside, Riverside, CA 92521; E. Shriver, 2501 NE Century Blvd, Hillsboro, OR 97124; A. Abousamra, 2501 NE Century Blvd, Hillsboro, OR 97124; Z. Waters, 2501 NE Century Blvd, Hillsboro, OR 97124.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2017 ACM 1539-9087/2017/09-ART148 $15.00
https://doi.org/10.1145/3126557

early-stage architectural design space exploration for GPUs, this paper presents a modeling framework that predicts the performance of a pre- silicon cycle-accurate GPU simulator 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 cycle-accurate simulation alone.

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 design space exploration, 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) design space exploration 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 to produce model features. The models are trained to predict the performance of the cycle-accurate GPU simulator 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 design space exploration, 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 quite different than what can be obtained from program counters in commercial-grade post-silicon 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 design space exploration. Our best performing model, random forest 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 cycle-accurate simulator.

2 FRAMEWORK IMPLEMENTATION

Figure 1 depicts our pre-silicon predictive modeling framework; 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 a cycle-accurate and internally validated GPU simulator (*GPUSim*) configured to model the same architecture.

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

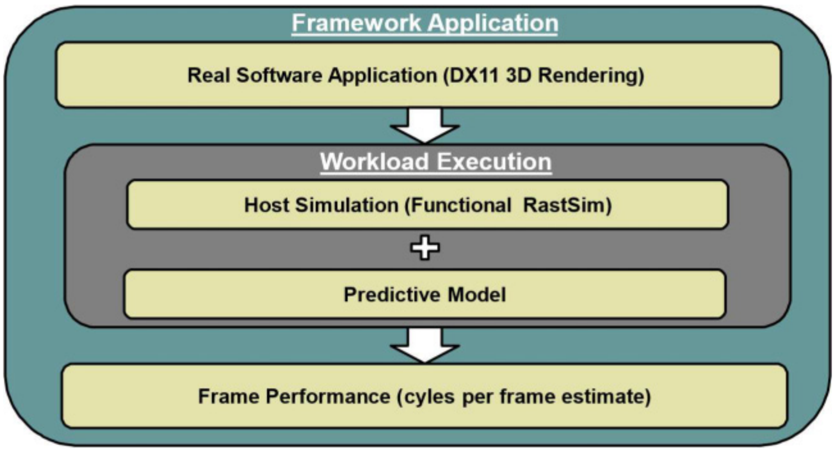

Framework Application
Real Software Application (DX11 3D Rendering)
Workload Execution
Host Simulation (Functional RastSim)
Predictive Model
Frame Performance (cyles per frame estimate)

Fig. 1. Modeling framework: a functional simulator executes 3D DirectX11 workloads. Performance counter measurements obtained from the simulator are used by predictive models to predict the performance of the cycle-accurate GPU simulator.

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.

Figure 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.

2.1 The Graphics Workload Library (GWL)

The *Graphics Workload Library (GWL)* contains 369 frames collected from 24 DirectX 11 games and GPU benchmarking tools, as listed in Table 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 cross-validation as discussed in Section 4.3.

2.2 Model Training and Validation Flow

Figure 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 4 illustrates the modeling training and deployment (prediction) phases of Figure 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 *cycles per frame (CPF)*; these golden reference values are used to train the model. We use *10-fold cross-validation* [14] to estimate *in-sample error* (E_{in}) and

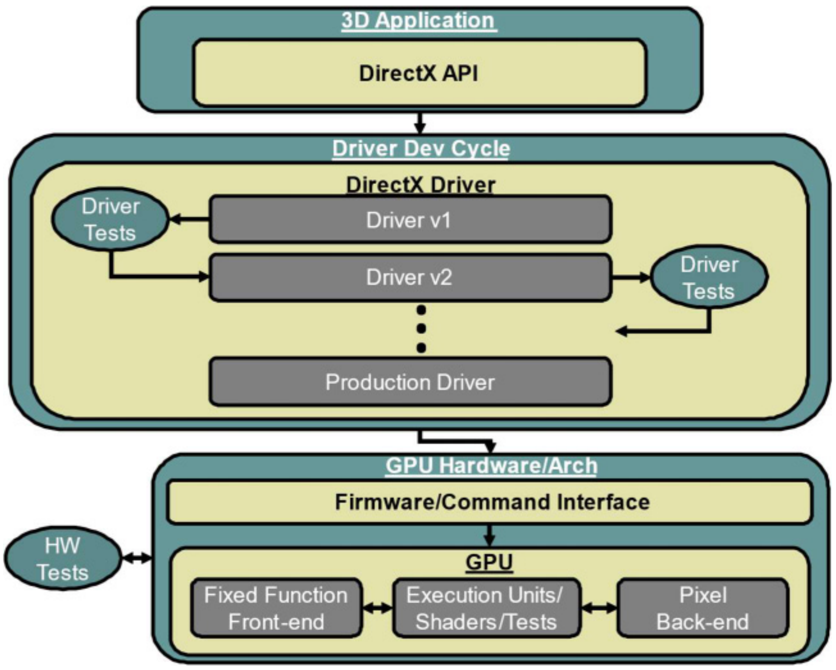


Fig. 2. GPU Hardware and software co-optimization process being targeted for rapid performance estimation. Hardware and software are co-optimized in lock-step fashion, requiring repeated simulation in a traditional design environment.

out-of-sample-error (E_{out}) to validate the model. The model is used to predict the CPF of previously unseen workloads.

2.3 RastSim

RastSim is a proprietary extension to the OpenSWR [15] rasterizer, which is fully integrated into the Mesa 3D Graphics Library [23] and normally targets the OpenGL API. As shown in Figure 5, RastSim consists of two primary subsystems: (1) the *RastSim Command Interface* and state tracker; and (2) 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 *Workload Characterization Framework (WCF)* that has been integrated into Mesa3D [23] as "archrast," which instruments the Rasterization Core and Command Interface to track render pipeline behavior, instruction counts, and workload execution state.

2.4 GPUSim

GPUSim is a proprietary cycle-accurate simulator 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

Table 1. 3D DirectX 11 Workloads, and Their Frame Counts

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

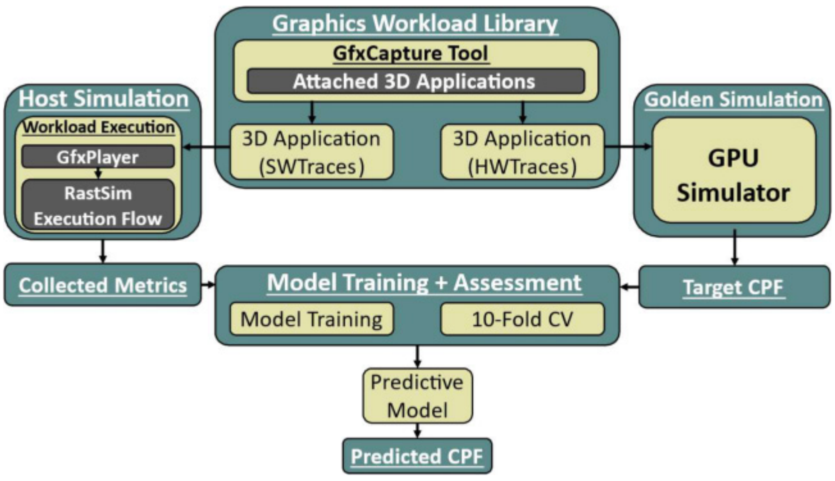


Fig. 3. Framework overview. Model training and validation require workload execution on both RastSim and GPUSim.

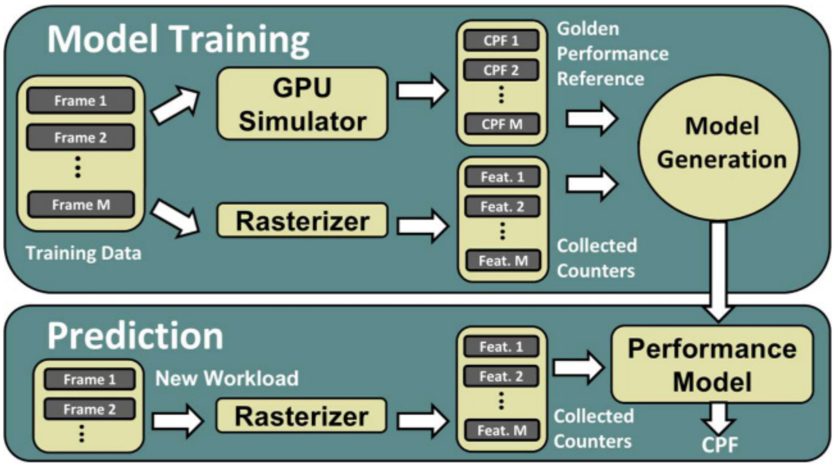


Fig. 4. Model training (top) and prediction flow (bottom).

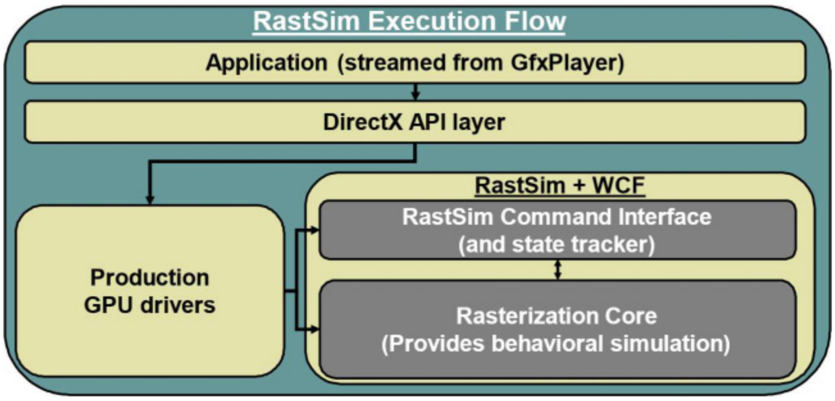

RastSim Execution Flow
Application (streamed from GfxPlayer)
DirectX API layer
Production GPU drivers
RastSim + WCF
RastSim Command Interface (and state tracker)
Rasterization Core (Provides behavioral simulation)

Fig. 5. RastSim utilizes GPU drivers, GfxPlayer and the DirectX API to provide a behavioral simulation of the Intel GPU DirectX pipeline to produce performance counters.

reference performance estimates for model training. To avoid disclosure of propriety information, CPF estimates produced by GPUSim are reported in normalized form.

3 RASTSIM GPU MODEL

We configured RastSim and GPUSim to model a 2-slice Intel Skylake GPU (Figure 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.

3.1 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 GT Interface (GTI), which performs I/O, and the State Variable Manager (SVM), which holds execution state variables. RastSim does not simulate the GA units,

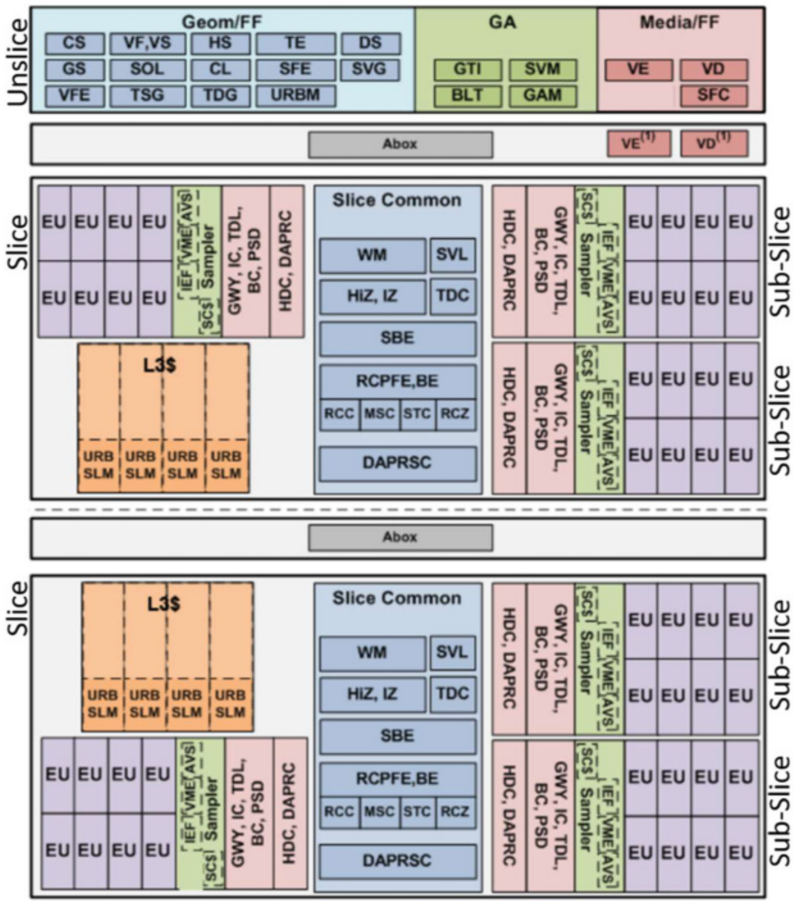


Fig. 6. Intel Skylake GT3 GPU architecture [17].

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 Compute Shader (CS), the Vertex Fetch (VF) and Vertex Shading (VS) units. It also models the three stages of DirectX 11 tessellation: the Hull Shader (HS), Tesselator (TE), and Domain Shader (DS) units, along with the Geometry Shader (GS), Clipper (CL) and the Stream Output Logic (SOL). The WCF provides 15 counters to track these Unslice behaviors, as shown in Figure 7.

3.2 Slice Architecture

Each slice is decomposed into three subgroups: (1) the *Slice Common* (Figure 8) which provides additional fixed function architectural units; (2) the *Sub-Slice* (Figure 9) which contains 24

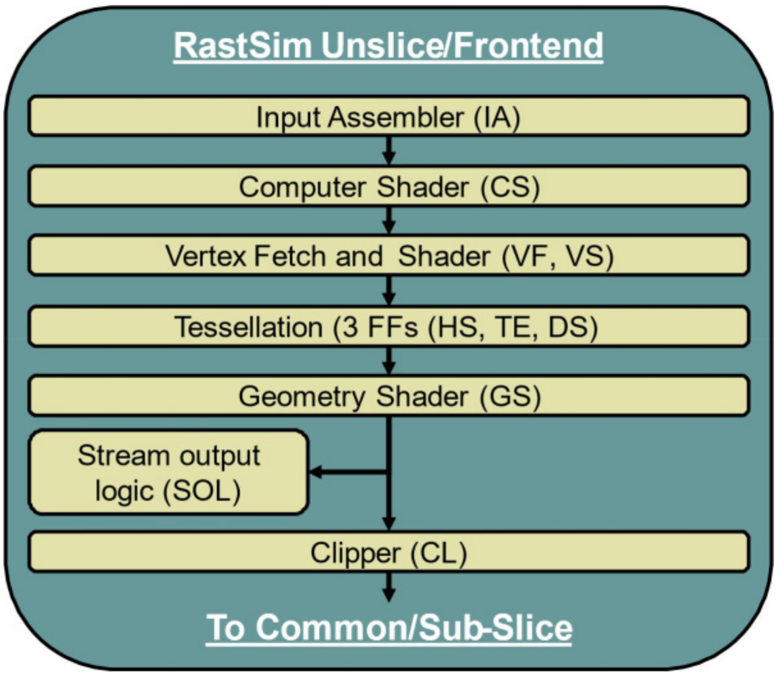


Fig. 7. Unslice units modeled in RastSim.

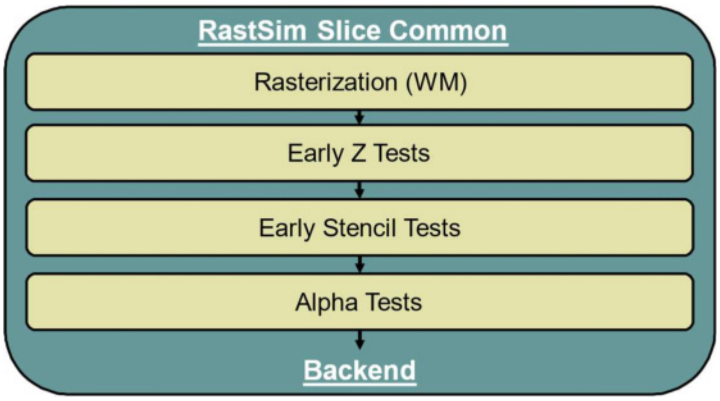


Fig. 8. Slice Common units modeled by RastSim. RastSim does not model caches, thread dispatch logic, or dedicated media units.

Execution Units (*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.

3.2.1 *Slice Common and L3 Cache.* The Slice Common fixed function 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 Figure 8, RastSim models

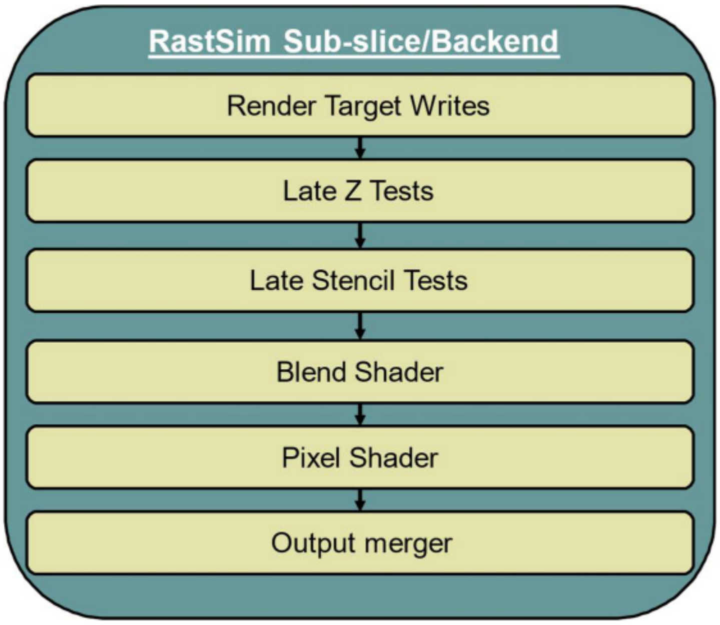


Fig. 9. Sub-slice units modeled by the software Rasterizer.

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.

3.2.2 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 Instruction Cache (IC), Local EU thread dispatcher (TDL), Data Cluster (HDC), render cache (DAPRC) a Pixel Shader dispatcher (PSD), and a Barycentric Calculator (BC). As shown in Figure 9, RastSim models programmable units such as the Pixel Shader Dispatch (PSD), Pixel Shader (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 thread dispatching 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.

4 REGRESSION MODELING FRAMEWORK

We employ a non-linear *Random Forest (RF)* regression model [7] to estimate pre-silicon GPU performance. Our model building procedure also produces and evaluates 14 linear regression models, which are used as a baseline for comparison. 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. Figure 10 depicts the ensemble of models that were trained; readers unfamiliar with the underlying statistical concepts described are encouraged to consult Ref. [14].

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 cycles-per-frame (CPF), one for each workload. Each

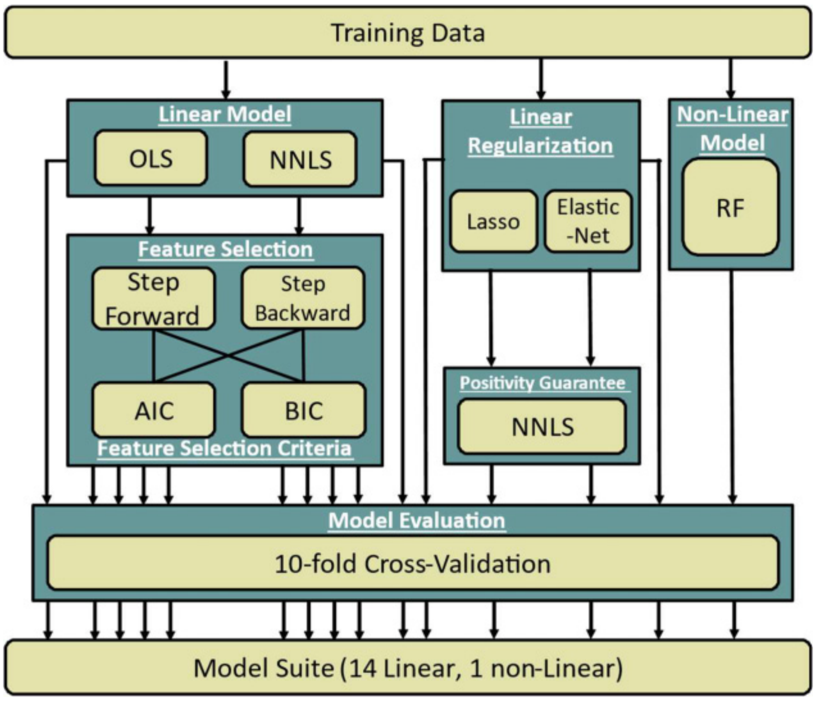


Fig. 10. Models produced and evaluated via our model framework.

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.

4.1 Linear Regression and Regularization Models

We generate 14 linear regression models, and one non-linear model that are placed into 5 modeling categories: **Category 1**, *Ordinary Least Squares* (OLS), and **Category 2**, *Non-Negative Least Squares* (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 *Akaike Information Criterion* (*AIC*) [1] and the *Bayesian Information Criterion* (*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 [29] and Elastic-Net [36], 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.

4.2 Random Forest Regression Model

Random Forest (RF) is an ensemble method, which aggregates the predictions of a collection of regression trees [6]. 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 [14] 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 *Residual Sum of Squares (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 [10]:

$$RSS\,(Split) = \sum_{i=0}^{|D_{left}|} (\mathrm{y_i} - \mathrm{y_L})^2 + \sum_{i=0}^{|D_{right}|} (\mathrm{y_i} - \mathrm{y_R})^2 \tag{1}$$

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 $RF = \cup_{j=1}^{n} T_j$. 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:

$$RF\,(m_i) = \frac{1}{n} \sum_{j=1}^{n} T_j\,(m_i)\,, \tag{2}$$

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.

4.3 Model Evaluation

We use *10-fold cross-validation* (*CV*) [14] to estimate model *generalizability*, i.e., its predictive capability when applied to unseen data. 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 Percentage Error (MAPE)* for each CV fold, and take the average to produce the *out-of-sample error* (E_{out}) [19]:

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

$$E_{out}\left(\hat{f}\right) = \frac{100}{M} \sum_{i=1}^{M} \left| \frac{y_i - \hat{f}^{-k(i)}\,(X_i)}{y_i} \right|. \tag{3}$$

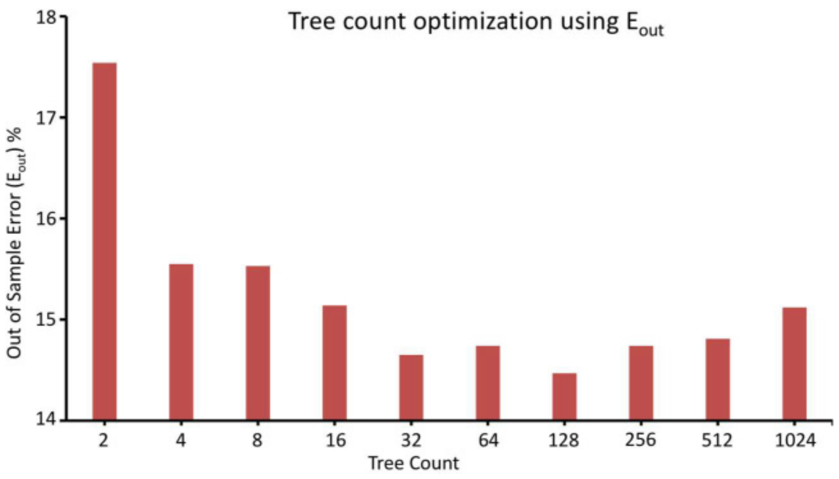


Fig. 11. Quantifying the impact of the number of trees in the RF model on E_{out}; lower values are better.

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

$$APE\left(X_i\right) = 100\left|\frac{y_i - f\left(X_i\right)}{y_i}\right|. \tag{4}$$

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 *inlier ratio* (I_R) is the percentage of traces that are inliers, i.e.:

$$I_R\left(f, T\right) = \frac{100}{M}\left|\left\{X_i | APE\left(X_i\right) \leq T, 1 \leq i \leq M\right\}\right|. \tag{5}$$

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

4.4 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}.

Figures 11 and 12 respectively depict the E_{out} value and inlier ratios at varying thresholds for RF models, where the number of trees varies from $n = 2^i$, $1 < i < 10$. Based on these results, we selected an RF model with $n = 128$ trees, which minimized E_{out} and demonstrated good inlier performance.

5 RASTERIZATION 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 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:

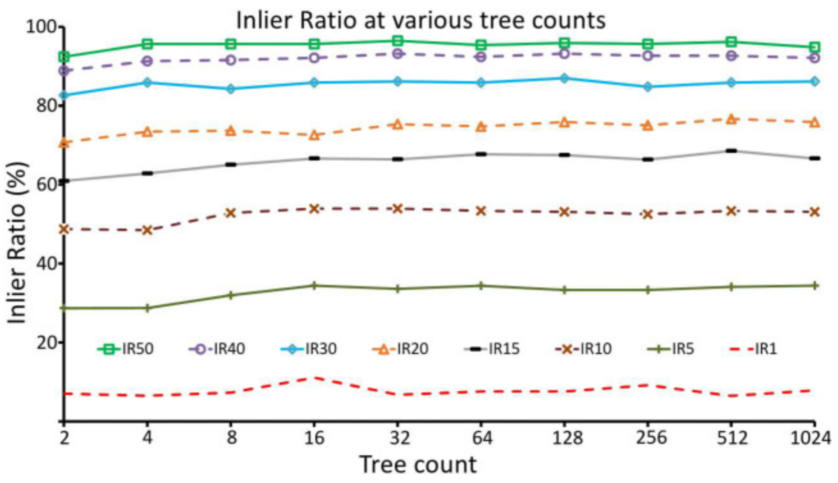
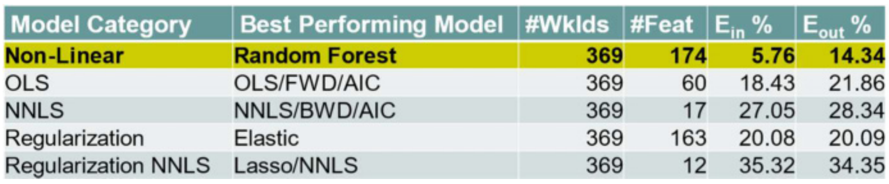


Fig. 12. Quantifying the impact of the number of trees in the RF model on I_R at varying thresholds; higher values are better.

Table 2. Comparison of the Best Model Category Errors Using the RastSim Workload Characterization Extensions



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

- The 5 best performing models obtained using the performance counters that we added to RastSim through the WCF (Table 2), including the inlier ratios (I_R) at various thresholds (Figure 13).
- The 5 performing models trained exclusively using performance counters originally available in RastSim (Table 3), including the inlier ratios (I_R) at various thresholds (Figure 14).

5.1 Predictive Model Results

Table 2 clearly indicates that RF is the best performing model, achieving an out-of-sample error 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 design space exploration; GPUSim is still required for detailed performance characterization and post-silicon performance validation.

Figure 13 reports the inlier ratios at 8 different threshold values $\boldsymbol{T} \in \{\mathbf{50\%, 40\%, 30\%, 20\%, 15\%, 10\%, 5\%,\ 1\%}\}$. RF and Elastic-net achieve the highest inlier ratios 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.

5.2 WCF Impact in RastSim

Table 3 and Figure 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 counter. The best performing model, once again, is RF, although its

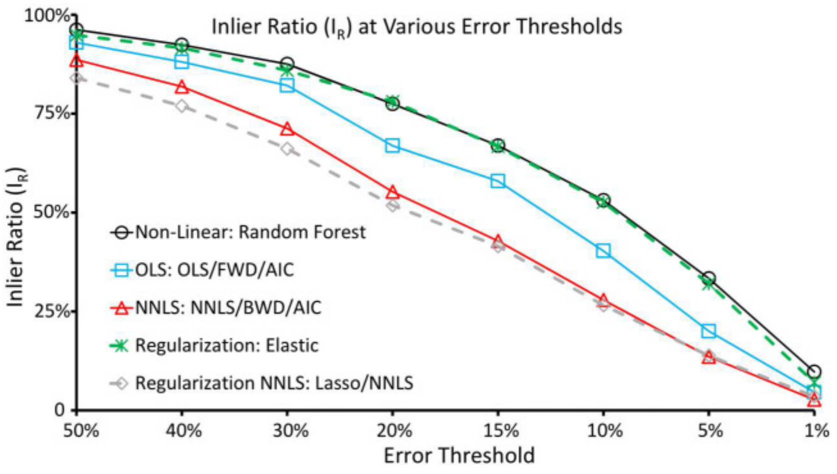


Fig. 13. Skylake Inlier rates at various error thresholds using the RastSim workload characterization extensions.

Table 3. Comparison of the Best Model Category Errors Without the RastSim Workload Characterization Extensions

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

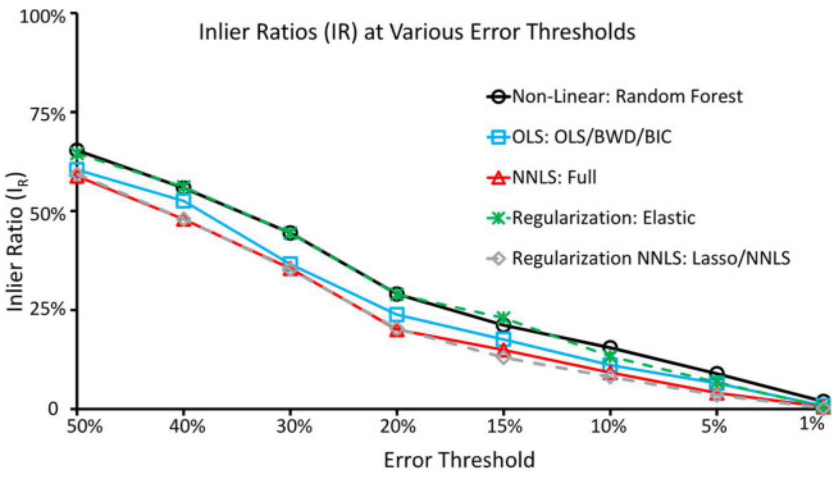


Fig. 14. Skylake Inlier rates at various error thresholds using the RastSim workload characterization extensions.

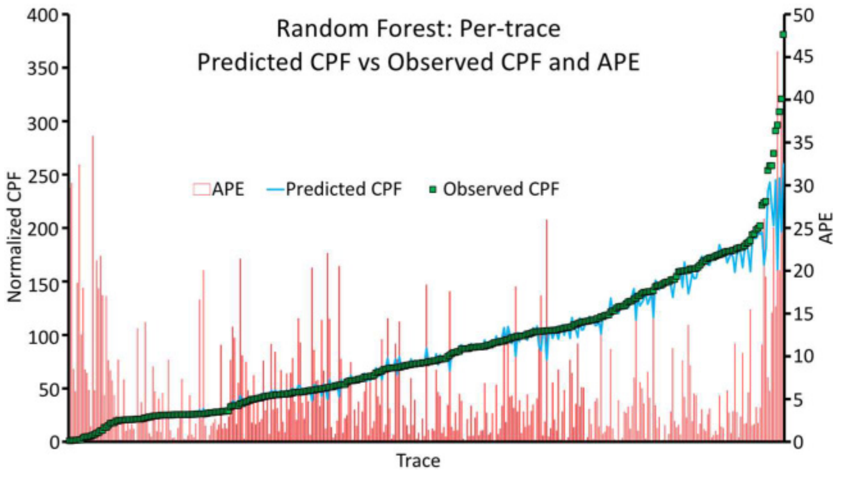


Fig. 15. Predicted and observed CPF and APE for each GWL workload using the RF model (Table 2). Data points are sorted in non-decreasing order of observed CPF.

out-of-sample error jumps to 52.34% (Table 3); this justifies the introduction of the WCF and its additional performance counters for this RastSim use case.

Figure 14 reports the inlier ratios at the same threshold values as Figure 13. Once again, RF and Elastic-net achieve the highest inlier ratios, although they are much lower than the results reported in Figure 13. For example, RF achieves a 29.03% inlier ratio at the 20% threshold, and a 15.47% inlier ratio 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).

5.3 Relative Accuracy Preservation

Figure 15 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 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.

To capture this information, it is possible to extend RastSim to model thread dispatching 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.

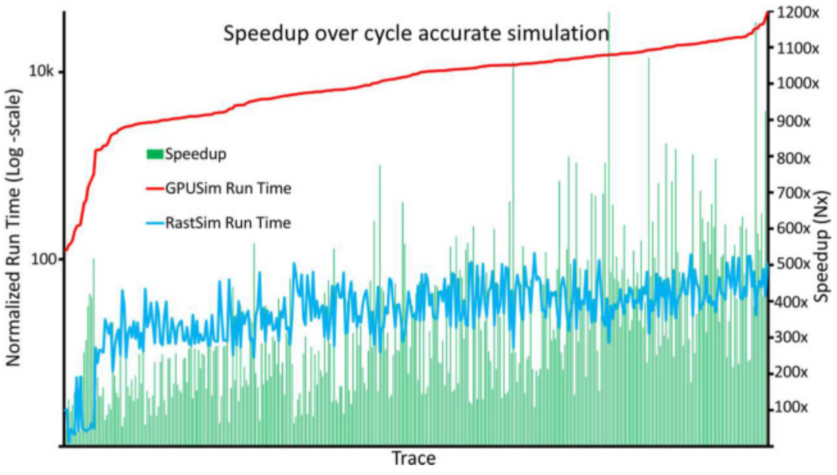


Fig. 16. Normalized RastSim and GPUSim execution time and speedup for each GWL workload. Data points are ordered in non-decreasing order of GPUSim execution time.

5.4 Predictive Model Speedup

The motivation for predictive modeling is to obtain workload performance estimates faster than cycle-accurate simulation. Figure 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 design space exploration. It is also worth noting that the WCF causes an average slowdown of 3x compared to native RastSim execution; the speedups included in Figure 16 include the WCH overhead.

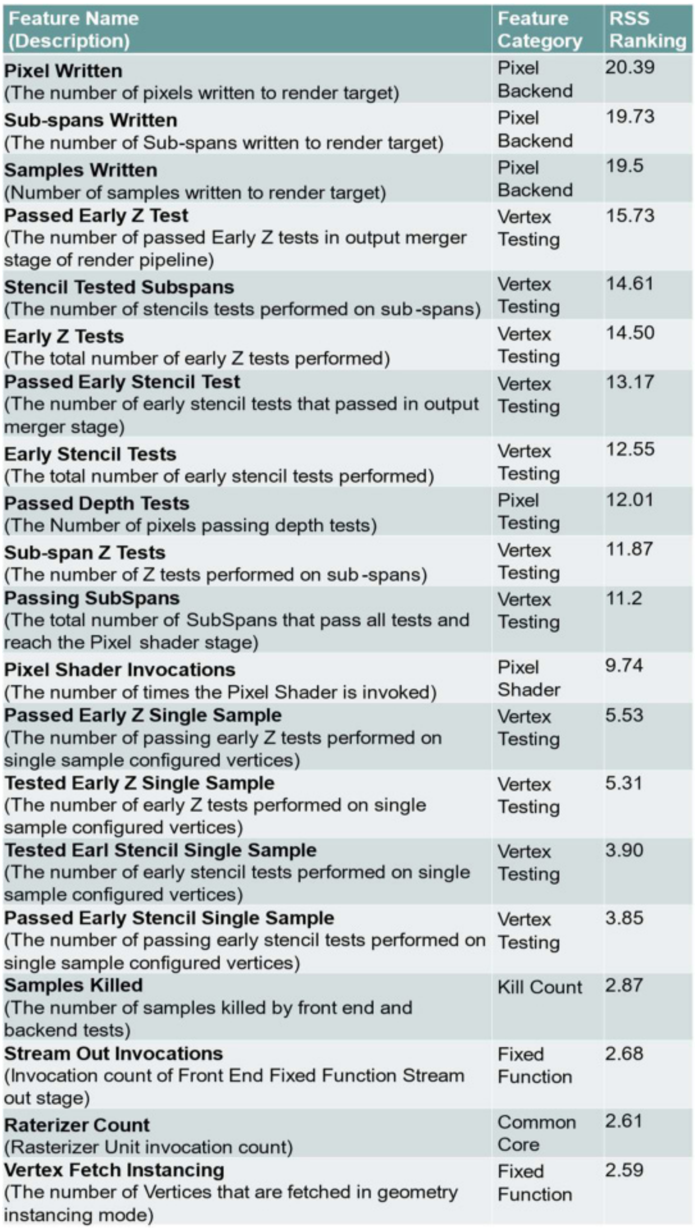
5.5 RF Feature Ranking

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 Equation (1), when it is used as a split point, for all trees in the forest for which it was selected [34]. Table 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.

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.

Table 4. The 20 Highest Ranked Performance Counters in the RF Model Based on RSS Ranking



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

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 9th 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 17th ranked feature in Table 4 (Samples Killed) is indicative of work *avoided* in late pipeline stages.

Two front end fixed function unit counters (Stream Out Invocations and Vertex Fetch Instancing), ranked 18th and 20th, track the number of times the stream out fixed function unit is used. Stream Out Invocations tracks the last stage in the Geom/FF units in the GPU unsliced pipeline, while Vertex Fetch Instancing refers to the first stage in the render pipeline during instancing mode. Rasterizer Counter, ranked 19th, 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 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.

6 RELATED WORK

Functional simulators lack timing information [12], while cycle-accurate simulators provide detailed timing simulation in addition to the functional simulation. The speed of cycle accurate architectural simulation is cost prohibitive, typically executing between 1 thousand instructions per second (KIPS) and 1 million instructions per second (MIPS) [25].

Cycle-accurate GPU architectural simulators (e.g., GPGPU-Sim [4], Multi2Sim [30], and Atilla [5]) run orders of magnitude slower than their functional counterparts. The importance of detailed architectural simulation is well-established, and simulators are often used for design space exploration (DSE), performance evaluation of workloads given a design, assessing architectural innovations, and for performance tuning of software [13]. The feasibility of these tasks is often improved by reducing cycle accurate simulation time in a variety of ways.

Techniques to reduce simulation time include raising the level of abstraction (lowering the level of detail), as employed RastSim here for GPUs and Sniper for CPUs [11], parallelization [25], FPGA acceleration [9], synthetic benchmark reduction [33], and representative statistical sampling [27, 32].

In principle, the predictive modeling techniques advocated in this paper should be viewed as being complementary to cycle-accurate simulation. In particular, feature ranking can indicate architectural performance bottlenecks, which can then be studied in far greater detail using cycle-accurate simulation.

6.1 Predictive Models for CPUs

Features that effectively predict CPU performance are not the same as than those that predict GPU performance, due to architectural differences. Linear regression models [18, 35], and artificial neural networks (ANNs) [20] have been applied to produce performance and power estimates for CPUs. Similar to our work, these models are built using performance and program counter readings.

Ma et al. [22] report that models trained using detailed simulators can be more accurate that 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.

6.2 Predictive Models for GPUs

Predictive models for GPUs have also been created using linear regression [3], decision trees [22], random forests [8] and ANNs [28]. To the best of our knowledge, no prior work has evaluated predictive models for pre-silicon GPU performance evaluation based on features obtained by functional simulation.

Wu et al. [31] introduce predictive models for post-silicon GPU performance to assist expert programmers with architecture-aware application tuning; their design space is limited to three degrees of freedom. XAPP [2] takes a similar approach, but uses performance counter measurements obtained from CPU execution to predict the performance of application kernels that could be mapped to a GPU. Similar to our work, their models are built using performance counter readings, they do not cross abstraction layers, as the host and target devices in both cases are both post-silicon.

Gerum et al. [12] 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. Performance counter measurements obtained from the hardware execution are input to an analytical model, which predicts performance at native execution speeds. 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.

Zhang et al. [34] create a modeling framework to predict the performance of *existing* ATI GPUs to understand the relationship between program behavior, GPU performance and power consumption to leverage these insights to provide instructive programming principles that can improve software design practices. 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 random forest regression and RSS-based feature ranking.

7 CONCLUSIONS AND FUTURE WORK

This paper has demonstrated that functional GPU simulation can accurately predict GPU performance, and that this approach can be applied during pre-silicon design space exploration. 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 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 [24]. 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 actions that must be performed and supported to render these applications at a function level. 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 GPGPU tasks we do not believe our models are generally applicable. It is also very common for these alternative workloads to exercise different hardware units, following a largely different execution pipeline in the GPU. Our training set reflects this assumption. We sincerely hope that our modifications have clarified the application space of our models.

Predictive modeling and feature ranking are not oracles, and relevant open questions remain. For example: (1) Can these models also accurately predict power/energy consumption? (2) Can these models generalize to rendering workloads that utilize different APIs, such as OpenGL? (3) Can these models generalize to GPGPU workloads written in languages such as CUDA or OpenCL? (4) Can these models generalize to FPGA workloads written in Vivado HLS-compatible C, or other domain-specific languages? (5) Can these models generalize to compute-intensive CPU workloads? And (6) Can these models generalize to different Skylake GPUs, and newer Intel GPU architectures? Future work will attempt to provide definitive answers to these questions.

REFERENCES

[1] Hirotugu Akaike. 1974. A new look at the statistical model identification. *IEEE Transactions on Automatic Control* 19 (1974), 716–723.

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

[3] Peter E. Bailey, David K. Lowenthal, Vignesh Ravi, Barry Rountree, Martin Schulz, De Supinski, and R. Bronis. 2014. Adaptive configuration selection for power-constrained heterogeneous systems. In *Parallel Processing (ICPP), 2014 43rd International Conference on.* 371–380.

[4] Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt. 2009. Analyzing CUDA workloads using a detailed GPU simulator. In *Performance Analysis of Systems and Software. 2009. ISPASS 2009. IEEE International Symposium on.* 163–174.

[5] Victor Barrio, Moya Barrio, Carlos González, Jordi Roca, Agusta Fernández, and E. Espasa. 2006. ATTILA: A cycle-level execution-driven simulator for modern GPU architectures. In *Performance Analysis of Systems and Software, 2006 IEEE International Symposium on.* 231–241.

[6] Leo Breiman, Jerome Friedman, Charles J. Stone, and Richard A. Olshen. 1984. *Classification and Regression Trees.* CRC Press, 1984.

[7] Leo Breiman. 2001. Random forests. *Machine Learning* 45 (2001), 5–32.

[8] Jianmin Chen, Bin Li, Ying Zhang, Lu Peng, and Jih-kwon Peir. 2011. Tree structured analysis on GPU power study. In *Computer Design (ICCD), 2011 IEEE 29th International Conference on.* 57–64.

[9] Derek Chiou, Dam Sunwoo, and Joonsoo Kim. 2007. Fpga-accelerated simulation technologies (fast): Fast, full-system, cycle-accurate simulators. In *Proceedings of the 40th Annual IEEE/ACM international Symposium on Microarchitecture.* 249–261.

[10] Adele Cutler. *Random Forests for Regression and Classification.* Retrieved 2017-07-14 from https://goo.gl/0d7mxj.

[11] Davy Genbrugge, Stijn Eyerman, and Lieven Eeckhout. 2010. Interval simulation: Raising the level of abstraction in architectural simulation. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on.* 1–12.

[12] Christoph Gerum, Oliver Bringmann, and Wolfgang Rosenstiel. 2015. Source level performance simulation of gpu cores. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2015.* 217–222.

[13] Qi Guo, Tianshi Chen, Yunji Chen, and Franz Franchetti. 2016. Accelerating architectural simulation via statistical techniques: A survey. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35 (2016), 433–446.

[14] Trevor Hastie, Jerome Friedman, and Robert Tibshirani. 2001. *The elements of Statistical Learning.* Springer series in statistics New York, 2001.

[15] Intel Corporation. *OpenSWR: A scalable High-Performance Sotware Rasterizer for SciVis.* Retrieved 2017-07-14 from https://goo.gl/G8faFn.

[16] Intel Corporation. *Intel Open Source HD Graphics, Intel Iris Graphics, and Intel Iris Pro Graphics Programmer's Reference Manual.* Retrieved 2017-07-14 from https://goo.gl/KX3wgK.

[17] Intel Corporation. *The Compute Architecture of Intel Processor Graphics Gen9.* Retrieved 2017-07-14 from https://goo.gl/RMmUc6.

[18] Engin Ïpek, Sally A. McKee, Rich Caruana, Bronis R. de Supinski, and Martin Schulz. 2006. *Efficiently Exploring Architectural Design Spaces Via Predictive Modeling.* ACM, 2006.

[19] Ron Kohavi. 1995. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Ijcai* 1137–1145.

[20] Benjamin C. Lee and David M. Brooks. 2006. Accurate and efficient regression modeling for microarchitectural performance and power prediction. In *ACM SIGOPS Operating Systems Review*. 185–194.

[21] Andy Liaw and Matthew Wiener. 2002. Classification and regression by randomforest. *R News* 2 (2002), 18–22.

[22] Xiaohan Ma, Mian Dong, Lin Zhong, and Zhigang Deng. 2009. Statistical power consumption analysis and modeling for GPU-based computing. In *Proceeding of ACM SOSP Workshop on Power Aware Computing and Systems (HotPower)*.

[23] Mesa 3D Graphics Library. *Gallium Driver*, SWR. Retrieved 2017-07-14 from https://goo.gl/YkuYyP.

[24] Microsoft Corporation. *Windows Hardware Certification Kit User's Guide*. Retrieved 2017-07-14 from https://goo.gl/s0TCzJ.

[25] Jason E. Miller, Harshad Kasture, and George Kurian. 2010. Graphite: A distributed parallel simulator for multicores. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*. 1–12.

[26] B. N. Petrov and F. Csáki. 1973. Information theory: Proceedings of the 2nd International symposium. Akadémiai Kiado. 1973, 1971, 267–281.

[27] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. 2002. Automatically characterizing large scale program behavior. In *ACM SIGARCH Computer Architecture News*. 45–57.

[28] Shuaiwen Song, Chunyi Su, Barry Rountree, and Kirk W. Cameron. 2013. A simplified and accurate model of power-performance efficiency on emergent GPU architectures. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*. 673–686.

[29] Robert Tibshirani. 1996. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society. Series B (Methodological)*. (1996), 267–288.

[30] Rafael Ubal, Byunghyun Jang, Perhaad Mistry, Dana Schaa, and David Kaeli. 2012. Multi2Sim: A simulation framework for CPU-GPU computing. In *Parallel Architectures and Compilation Techniques (PACT), 2012 21st International Conference on*. 335–344.

[31] Gene Wu, Joseph L. Greathouse, Alexander Lyashevsky, Nuwan Jayasena, and Derek Chiou. 2015. GPGPU performance and power estimation using machine learning. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*. 564–576.

[32] Roland E. Wunderlich, Thomas F. Wenisch, Babak Falsafi, and James C. Hoe. 2003. SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling. In *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*. 84–95.

[33] Zhibin Yu, Lieven Eeckhout, Nilanjan Goswami, Tao Li, Lizy John, Hai Jin, and Chengzhong Xu. 2013. Accelerating GPGPU architecture simulation. In *ACM SIGMETRICS Performance Evaluation Review*. 331–332.

[34] Ying Zhang, Yue Hu, Bin Li, and Lu Peng. 2011. Performance and power analysis of ATI GPU: A statistical approach. In *Networking, Architecture and Storage (NAS), 2011 6th IEEE International Conference on*. 149–158.

[35] Xinnian Zheng, Lizy K. John, and Andreas Gerstlauer. 2016. Accurate phase-level cross-platform power and performance estimation. In *Proceedings of the 53rd Annual Design Automation Conference*. 4.

[36] Hui Zou and Trevor Hastie. 2005. Regularization and variable selection via the elastic net. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* 67 (2005), 301–320.

Received April 2017; revised June 2017; accepted June 2017