GPGPU Performance and Power Estimation Using Machine Learning

Gene Wu*, Joseph L. Greathouse†, Alexander Lyashevsky†, Nuwan Jayasena†, Derek Chiou*

*Electrical and Computer Engineering
The University of Texas at Austin
{Gene.Wu, Derek}@utexas.edu

†AMD Research
Advanced Micro Devices, Inc.
{Joseph.Greathouse, Alexander.Lyashevsky, Nuwan.Jayasena}@amd.com

***Abstract*—Graphics Processing Units (GPUs) have numerous configuration and design options, including core frequency, number of parallel compute units (CUs), and available memory bandwidth. At many stages of the design process, it is important to estimate how application performance and power are impacted by these options.**

This paper describes a GPU performance and power estimation model that uses machine learning techniques on measurements from real GPU hardware. The model is trained on a collection of applications that are run at numerous different hardware configurations. From the measured performance and power data, the model learns how applications scale as the GPU's configuration is changed. Hardware performance counter values are then gathered when running a new application on a single GPU configuration. These dynamic counter values are fed into a neural network that predicts which scaling curve from the training data best represents this kernel. This scaling curve is then used to estimate the performance and power of the new application at different GPU configurations.

Over an 8× range of the number of CUs, a 3.3× range of core frequencies, and a 2.9× range of memory bandwidth, our model's performance and power estimates are accurate to within 15% and 10% of real hardware, respectively. This is comparable to the accuracy of cycle-level simulators. However, after an initial training phase, our model runs as fast as, or faster than the program running natively on real hardware.

I. Introduction

Graphics processing units (GPUs) have become standard devices in systems ranging from cellular phones to supercomputers. Their designs span a wide range of configurations and capabilities, from small, power-efficient designs in embedded systems on chip (SoCs) to large, fast devices meant to prioritize performance. Adding to the complexity, modern processors reconfigure themselves at runtime in order to maximize performance under tight power constraints. These designs will rapidly change core frequency and voltage [1], [43], modify available bandwidth [14], and quickly power gate unused hardware to reduce static power usage [31], [37].

With this wide range of possible configurations, it is critical to rapidly analyze application performance and power. Early in the design process, architects must verify that their plan will meet performance and power goals on important applications. Software designers, similarly, would like to verify performance targets on a wide range of devices. These estimates can even result in better dynamic reconfiguration decisions [46], [49].

Design-time estimates are traditionally performed using low-level simulators such as GPGPU-Sim [8], which can be carefully configured to yield accurate estimates, often within 10-20% of real hardware. The performance of such simulators, however, precludes their use for online analysis or large design space explorations. GPGPU-Sim, for instance, runs millions of times slower than native execution [35]. Such overheads often result in the use of reduced input sets, which can further decrease accuracy, or sampling techniques which can still run many times slower than real hardware if they model each sample in a slow simulator [52].

To alleviate this problem, researchers have built a variety of analytic performance and power models [6], [23], [28], [30], [32], [38], [40], [45], [54], [55]. These range from estimates that use static code analyses [6] to linear regressions based on hardware performance counters [28]. These models are primarily built for, and trained on, single GPU configurations. This can yield accurate estimates but limits their ability to model large design spaces or runtime hardware changes.

This paper focuses on rapidly estimating the performance and power of GPUs across a range of hardware configurations. We begin by measuring a collection of OpenCL™ kernels on a real GPU at various core frequencies, memory bandwidths, and compute unit (CU) counts. This allows us to build a set of scaling surfaces that describe how the power and performance of these applications change across hardware configurations. We also gather performance counters, which give a fingerprint that describes each kernel's use of the underlying hardware.

Later, when analyzing previously unseen kernels, we gather the execution time, power, and performance counters at a single hardware configuration. Using machine learning (ML) methods, we use these performance counter values to predict which training kernel is most like this new kernel. We then estimate that the scaling surface for that training kernel also represents the kernel under test. This allows us to quickly estimate the power and performance of the new kernel at numerous other hardware configurations.

For the variables we can explore during the training phase, we find our ML-based estimation model to be as accurate as values commonly reported for microarchitectural simulations. We are able to estimate the performance of our test kernels across a 3.3× change in core frequency, a 2.9× change in memory bandwidth, and an 8× change in CUs with an average error of 15%. We are able to estimate dynamic power usage across the same range with an average error of 10%. In addition, this is faster than low-level simulators; after an offline training phase, online predictions across the range of supported settings takes only a single run on the real hardware. Subsequent predictions only require running the ML predictor, which can be much faster than running the kernel itself.

This work makes the following contributions:

- We demonstrate on real hardware that the performance and power of General-purpose GPU (GPGPU) kernels scale in a limited number of ways as hardware configuration parameters are changed. Many kernels scale similarly to others, and the number of unique scaling patterns is limited.
- We show that, by taking advantage of the first insight, we can perform clustering and use machine learning techniques to match new kernels to previously observed kernels whose performance and power will scale in similar ways.
- We then describe a power and performance estimation model that uses performance counters gathered during one run of a kernel at a single configuration to predict performance and power across other configurations with an average error of only 15% and 10%, respectively, at near-native-execution speed.

The remainder of this paper is arranged as follows. Section II motivates this work and demonstrates how GPGPU kernels scale across hardware configurations. Section III details our ML model and how it is used to make predictions. Section IV describes the experimental setup we used to validate our model and Section V details the results of these experiments. Section VI lists related work, and we conclude in Section VII.

II. Motivation

This section describes a series of examples that motivate the use high-level performance and power models before introducing our own model based on ML techniques.

A. Design Space Exploration

Contemporary GPUs occupy a wide range of design points in order to meet market demands. They may use similar underlying components (such as the CUs), but the SoCs may have dissimilar configurations. As an example, Table I lists a selection of devices that use graphics hardware based on AMD's Graphics Core Next microarchitecture. As the data shows, the configurations vary wildly. At the extremes, the AMD Radeon^{TM} R9 290X GPU, which is optimized for maximum performance, has 22× more CUs running at 2.9× the frequency and with 29× more memory bandwidth than the tablet-optimized AMD E1-6010 processor.

Tremendous effort goes into finding the right configuration for a chip before expending the cost to design it. The performance of a product must be carefully weighed against factors such as area, power, design cost, and target price. Applications of interest are studied on numerous designs to ensure that a product will meet business goals.

Low-level simulators, such as GPGPU-Sim [8] allow accurate estimates, but they are not ideal for early design space explorations. These simulators run 4-6 orders of magnitude slower than native execution, which limits the applications (and inputs) that can be studied. In addition, configuring such simulators to accurately represent real hardware is time-consuming and error-prone [21], which limits the number of design points that can be easily explored.

TABLE I: Products built from similar underlying AMD logic blocks that contain GPUs with very different configurations.

Name	CUs	Max. Freq. (MHz)	Max. DRAM BW (GB/s)
AMD E1-6010 APU [22]	2	350	11
AMD A10-7850K APU [2]	8	720	35
Microsoft Xbox One^{TM} processor [5]	12	853	68
Sony PlayStation$^{®}$ 4 processor [5]	18	800	176
AMD Radeon^{TM} R9-280X GPU [3]	32	1000	288
AMD Radeon^{TM} R9-290X GPU [3]	44	1000	352

One common way of mitigating simulation overheads is to use reduced input sets [4]. The loss of accuracy caused by this method led to the development of more rigorous sampling methods such as SimPoints [44] and SMARTS [51]. These can reduce simulation time by two orders of magnitude while adding errors of only 10-20% [52]. Nonetheless, this still runs hundreds to thousands of times slower than real hardware.

High-level models are a better method of pruning the design space during early explorations. These models may be less accurate than low-level simulators, but they allow rapid analysis of many full-sized applications on numerous configurations. We do not advocate for eliminating low-level simulation, as it offers valuable insight that high-level models cannot produce. However, high-level models allow designers to prune the search space and only spend time building low-level models for potentially interesting configurations.

B. Software Analysis

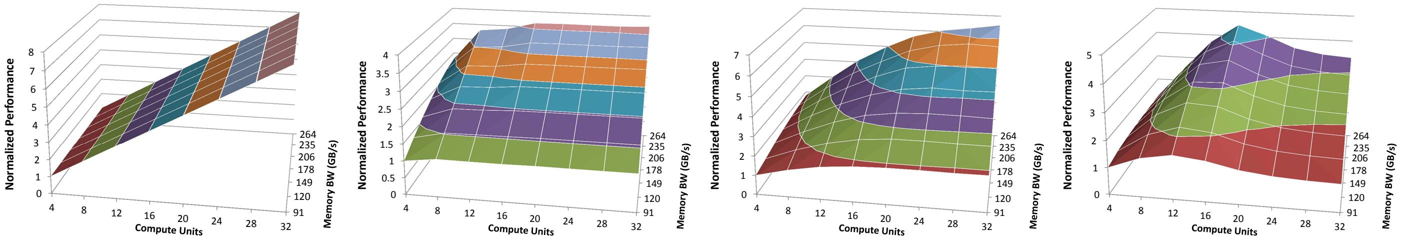
GPGPU software is often carefully optimized for the hardware on which it will run. With dozens of models on the market at any point in time (as partially demonstrated in Table I), it is difficult to test software in every hardware configuration that consumers will use. This can complicate the task of setting minimum requirements, validating performance goals, and finding performance and power regressions.

Low-level simulators are inadequate for this task, as they are slow and require great expertise to configure and use. In addition, GPU vendors are loath to reveal accurate low-level simulators, as they can reveal proprietary design information.

High-level models are better for this task. Many existing high-level models focus on finding hardware-related bottlenecks, but they are limited to studying a single device configuration [23], [28], [32], [38], [55]. There are others that estimate how an application would perform as parameters such as frequency change [39]. We will later detail why these relatively simple models have difficulty accurately modeling complex modern GPUs. Nonetheless, their general goal matches our own: provide fast feedback about application scaling.

C. Online Reconfiguration

Modern processors must optimize performance under tight power budgets. Towards this end, dynamic voltage and frequency scaling (DVFS) varies core and memory frequency in response to demand [14], [31]. Recent publications also advocate for disabling GPU CUs when parallelism is least helpful [41]. We refer to these methods as online reconfiguration.



(a) A compute-bound kernel.

(b) A bandwidth-bound kernel.

(c) A balanced kernel.

(d) An irregular kernel.

Fig. 1: Four different GPGPU performance scaling surfaces. Frequency is held at 1 GHz while CUs and bandwidth are varied.

Advanced reconfiguration systems try to predict the effect of their changes on the total performance of the system. For instance, boosting a CPU's frequency may prevent a nearby GPU from reaching its optimal operating point, requiring online thermal models to maximize chip-wide performance [42]. Similarly, using power and performance estimates to proactively choose the best voltage and frequency state (rather than reacting only to the results of previous decisions) can enable power capping solutions, optimize energy usage, and yield higher performance in "dark silicon" situations [46], [49].

Estimating how applications scale across hardware configurations is a crucial aspect of these systems. These estimates must be made rapidly, precluding low-level simulators, and must react to dynamic program changes, which limits the use of analytic models based on static analyses. We therefore study models that quickly estimate power and performance using easy-to-obtain dynamic hardware event counters.

D. High-Level GPGPU Model

The recurring theme in these examples is a desire for a fast, relatively accurate estimation of performance and power at different hardware configurations. Previous studies have described high-level models that can predict performance at different frequencies, which can be used for online optimizations [24], [39], [45]. Unfortunately, these systems are limited by their models designed after abstractions of real hardware.

Fig. 1 shows the performance of four OpenCL$^{\text{TM}}$ kernels on a real GPU. The performance (Y-axis) changes as the number of active CUs (X-axis) and the available memory bandwidth (Z-axis) are varied. Frequency is fixed at 1 GHz.

Existing models that compare compute and memory work can easily predict the applications shown in Fig. 1(a) and 1(b) because they are compute and bandwidth-bound, respectively. Fig. 1(c) is more complex; its limiting factor depends on ratio of CUs to bandwidth. This requires a model that handles hardware component interactions [23], [54].

Fig. 1(d) shows a performance effect that can be difficult to predict with simple models. Adding more CUs helps performance until a point, whereupon performance drops as more are added. This is difficult to model using linear regression or simple compute-versus-bandwidth formulae.

As more CUs are added, more pressure is put on the shared L2 cache. Eventually, the threads' working sets overflow the L2 cache, degrading performance. This effect has been shown in simulation, but simple analytic models do not take it into account [29], [34]. Numerous other non-obvious scaling results exist, but we do not detail them due to space constraints. Suffice to say, simple models have difficulty with kernels that are constrained by complex microarchitectural limitations.

Nevertheless, our goal is to create a high-level model that we can use to estimate performance and power across a wide range of hardware configurations. As such, we build an ML model that can perform these predictions quickly and accurately.

We begin by training on a large number of OpenCL kernels. We run each kernel at numerous hardware configurations while monitoring power, performance, and hardware performance counters. This allows us to collect data such as that shown in Fig. 1. We find that the performance and power of many GPU kernels scale similarly. We therefore cluster similar kernels together and use the performance counter information to fingerprint that scaling surface.

To make predictions for new kernels, we measure the performance counters obtained from running that kernel at one hardware configuration. We then use them to predict which scaling surface best describes this kernel. With this, we can quickly estimate the performance or power of a kernel at many configurations. The following section details this process.

III. Methodology

We describe our modeling methodology in two passes. First, Section III-A describes the methodology at a high level, providing a conceptual view of how each part fits together. Then Section III-B, Section III-C, and Section III-D go into the implementation details of the different parts of the model. For simplicity, these sections describe a performance model, but this approach can be applied to generate power models as well.

A. Overview

While our methodology is amenable to modeling any parameter that can be varied, for the purposes of this study, we define a GPU's hardware configuration as its number of CUs, engine frequency, and memory frequency. We take as an input to our predictor measurements gathered on one specific GPU hardware configuration called the *base hardware configuration*. Once a kernel has been executed on the base hardware configuration, the model can be used to predict the kernel's performance on a range of target hardware configurations.

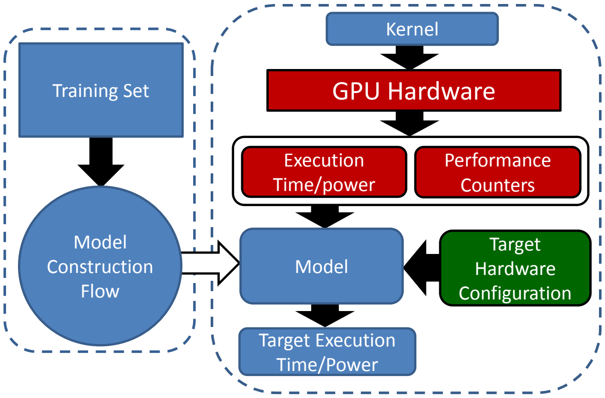
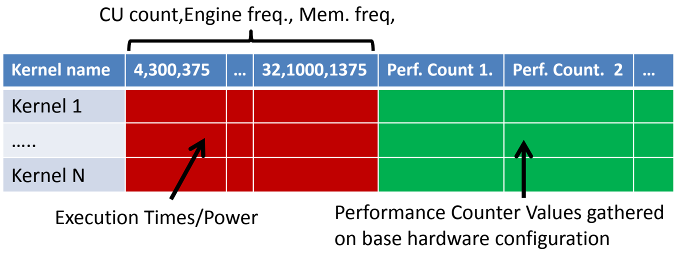


Fig. 2: The model construction and usage flow. Training is done on many configurations, while predictions require measurements from only one.


CU count,Engine freq., Mem. freq,
Execution Times/Power
Performance Counter Values gathered on base hardware configuration

Kernel name	4,300,375	...	32,1000,1375	Perf. Count 1.	Perf. Count. 2	...
Kernel 1	🟥	🟥	🟥	🟩	🟩	🟩
.....	🟥	🟥	🟥	🟩	🟩	🟩
Kernel N	🟥	🟥	🟥	🟩	🟩	🟩

Fig. 3: The model's training set, which contains the performance or power of each training kernel for a range of hardware configurations.

The model construction and usage flow are depicted in Fig. 2. The construction algorithm uses a training data set containing execution times and performance counter values collected from executing training kernels on real hardware. The values in the training set are shown in Fig. 3. For each training kernel, execution times and performance counter values across a range of hardware configurations are stored in the training set. The performance counter values collected while executing each training kernel on the base hardware configuration are also stored.

Once the model is constructed, it can be used to predict the performance of new kernels, from outside the training set, at any target hardware configuration within the range of the training data. To make a prediction, the kernel's performance counter values and base execution time must first be gathered by executing it on the base hardware configuration. These are then passed to the model, along with the desired target hardware configuration, which will output a predicted execution time at that target configuration. The model is constructed once offline but used many times. It is not necessary to gather a training set or rebuild the model for every prediction.

Model construction consists of two major phases. In the first phase, the training kernels are clustered to form groups of kernels with similar performance scaling behaviors across hardware configurations. Each resulting cluster represents one scaling behavior found in the training set.

In the simple example shown in Fig. 4, there are six training kernels being mapped to three clusters. Training kernels 1 and 5 are both bandwidth bound and are therefore mapped to the same cluster. Kernel 3 is the only one of the six that is purely compute bound, and it is mapped to its own cluster. The remaining kernels scale with both compute and memory bandwidth, and they are all mapped to the remaining cluster. While this simple example demonstrates the general approach, the actual model identifies larger numbers of clusters with more complex scaling behaviors in a 4D space.

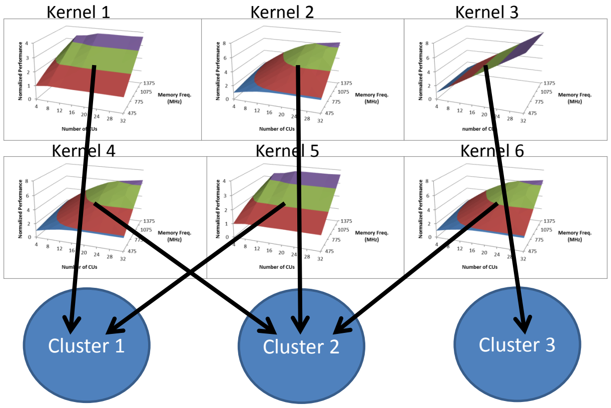


Fig. 4: Forming clusters of kernel scaling behaviors. Kernels that scale in similar ways are clustered together.

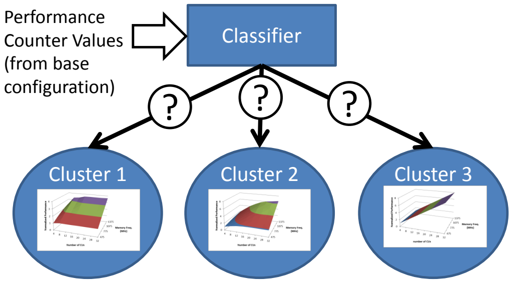


Fig. 5: Building a classifier to map from performance counter values to clusters.

In the second phase, a classifier is constructed to predict which cluster's scaling behavior best describes a new kernel based on its performance counter values. The classifier, shown Fig. 5, would be used to select between the clusters in Fig. 4. The classifier and clusters together allow the model to predict the scaling behavior of a new kernel across a wide range of hardware configurations using information taken from executing it on the base hardware configuration. When the model is asked to predict the performance of a new kernel at a target hardware configuration, the classifier is accessed first. The classifier chooses one cluster, and that cluster's scaling behavior is used to scale the baseline execution time to the target configuration in order to make the desired prediction.

Fig. 6 gives a detailed view of the model architecture. Notice that the model contains multiple sets of clusters and classifiers. Each cluster set and classifier pair is responsible for providing scaling behaviors for a subset of the CU, engine frequency, and memory frequency parameter space. For example, the top cluster set in Fig. 6 provides information about the scaling behavior when CU count is 8. This set provides performance scaling behavior when engine and memory frequencies are varied and CU count is fixed at 8. The exact

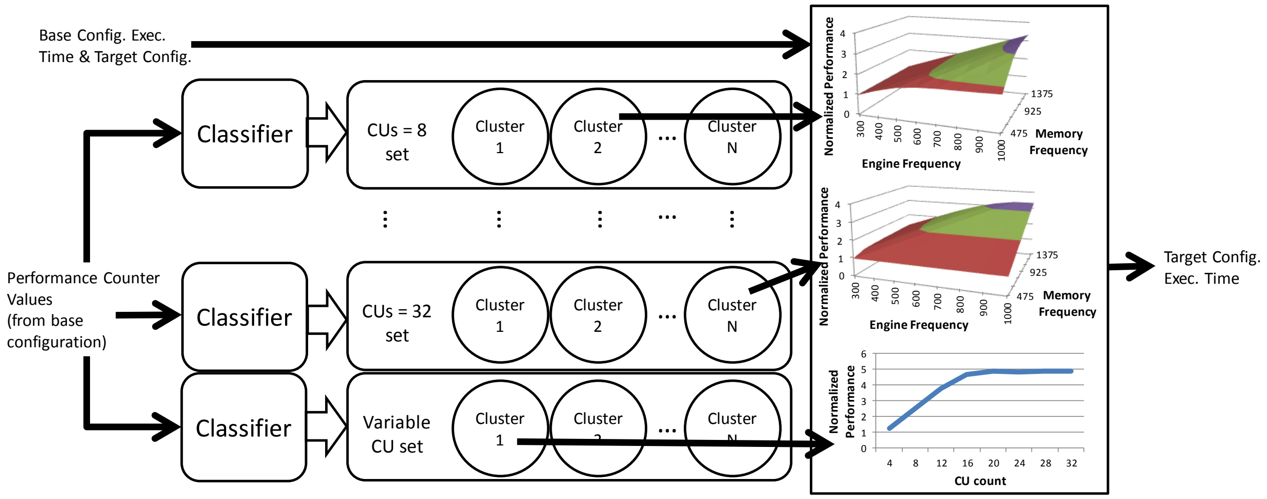


Fig. 6: Detailed architecture of our performance and power predictor. Performance counters from one execution of the application are used to find which cluster best represents how this kernel will scale as the hardware configuration is changed. Each cluster contains a scaling surface that describes how this kernel will scale when some hardware parameters are varied.

number of sets and classifier pairs in a model depends on the hardware configurations that appear in the training set.

The scaling information from these cluster sets allows scaling from the base to any other target configuration. By dividing the parameter space into multiple regions and clustering the kernels once for each region, the model is given additional flexibility. Two kernels may have similar scaling behaviors in one region of the hardware configuration space, but different scaling in another region. For example, two kernels may scale identically with respect to engine frequency, but one may not scale at all with CU count while the other scales linearly. For cases like this, the kernels may be clustered together in one cluster set but not necessarily in others. Breaking the parameter space into multiple regions and clustering each region independently allows kernels to be grouped into clusters without requiring that they scale similarly across the entire parameter space. In addition, by building a classifier for each region of the hardware configuration space, each classifier only needs to learn performance counter trends relevant to its region, which reduces their complexity.

The remainder of this section provides descriptions of the steps required to construct this model. The calculation of scaling values, which are used to describe a kernel's scaling behavior, is described in Section III-B. Section III-C describes how these are used to form sets of clusters. Finally, Section III-D describes the construction of the neural network classifiers.

B. Scaling Surfaces

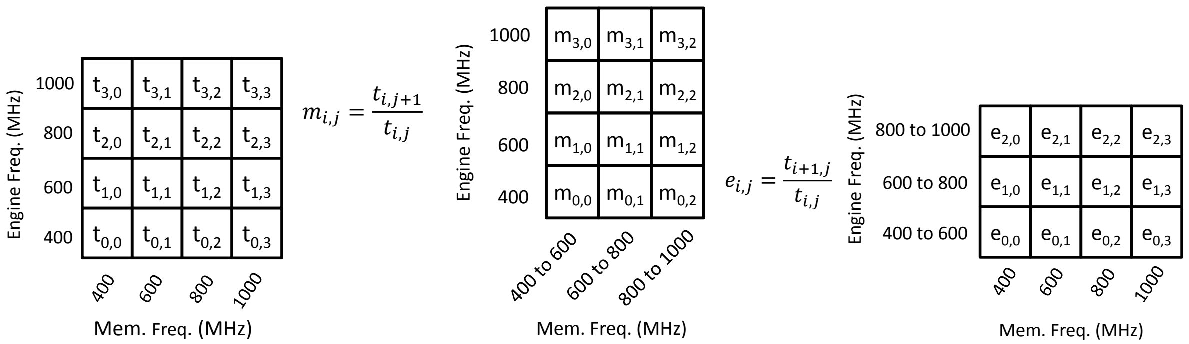
Our first step is to convert the training kernel execution times into performance scaling values, which capture how the kernels' performance changes as the number of CUs, engine frequency, and memory frequency are varied. Scaling values are calculated for each training kernel and then put into the clustering algorithm. Because we want to form groups of kernels with similar scaling behaviors, even if they have vastly different execution times, the kernels are clustered using scaling values rather than raw execution times.

To calculate scaling values, the training set execution times are grouped by kernel. Then, for each training kernel, a 3D matrix of execution times is formed. Each dimension corresponds to one of the hardware configuration parameters. In other words, the position of each execution time in the matrix is defined by the number of CUs, engine frequency, and memory frequency combination it corresponds to.

The matrix is then split into sub-matrices, each of which represents one region of the parameter space. Splitting the matrix in this way will form the multiple cluster sets seen in Fig. 6. An execution time sub-matrix is formed for each possible CU count found in the training data. For example, if training set data have been gathered for CU counts of 8, 16, 24, and 32, four sub-matrices will be formed per kernel.

The process to convert an execution time sub-matrix into performance scaling values is illustrated using the example in Fig. 7. An execution time sub-matrix with a fixed CU count is shown in Fig. 7(a). Because the specific CU count does not change any of the calculations, it has not been specified. In this example, it is assumed that the training set engine and memory frequency both range from 400 MHz to 1000 MHz with a step size of 200 MHz. However, the equations given apply to any range of frequency values, and the reshaping of the example matrices to match the new range of frequency values is straightforward.

Two types of scaling values are calculated from the execution times shown in Fig. 7. Memory frequency scaling values, shown in Fig. 7(b), capture how execution times change as memory frequency increases and engine frequency remains the same. Engine frequency scaling values, shown in Fig. 7(c), capture how execution times change as engine frequency increases and memory frequency remains the same. In the memory scaling matrix, each entry's column position corresponds to a change in memory frequency and its row position corresponds to a fixed engine frequency. In the engine scaling matrix, each entry's column position corresponds to a change in engine frequency and its row position corresponds to a fixed memory frequency.



(a) Execution times (b) Memory frequency scaling values. (c) Engine frequency scaling values.

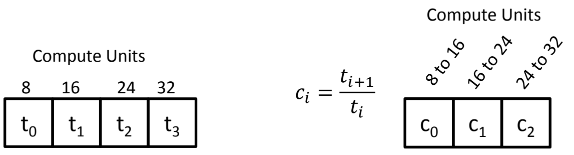
Fig. 7: Calculation of frequency scaling values.

As long as the CU counts of the base and target configurations remain the same, one or more of the shown scaling values can be multiplied with a base configuration execution time to predict a target configuration execution time. We will show how to calculate the scaling factors to account for variable CU counts shortly. The example in Equation 1 shows how these scaling values are applied. The base configuration has a 400 MHz engine clock and a 400 MHz memory clock. The base execution time is $t_{0,0}$. We can predict the target execution time, $t_{3,3}$, of a target configuration with a 1000 MHz engine clock and a 1000 MHz memory clock using Equation 1. Scaling values can also be used to scale from higher frequencies to lower frequencies by multiplying the base execution time with the reciprocals of the scaling values. For example, because $m_{0,0}$ can be used to scale from a 400 MHz to 600 MHz memory frequency, $\frac{1}{m_{0,0}}$ can be used to scale from a 600 MHz to 400 MHz memory frequency. Scaling execution times in this way may not seem useful when only training data is considered, as the entire execution time matrix is known. However, after the model has been constructed, the same approach can be used to predict execution times for new kernels at configurations that have not been run on real hardware.

$$t_{3,3} = t_{0,0} \prod_{j=0}^{2} e_{0,j} \prod_{i=0}^{2} m_{i,2} \qquad (1)$$

In addition to the engine and memory frequency scaling values, a set of scaling values is calculated to account for varying CU count. A sub-matrix, from which CU scaling values can be extracted, is constructed. All entries of this new sub-matrix correspond to the base configuration engine and memory frequency. However, each entry corresponds to one of the CU counts found in the training data. An example execution time sub-matrix with variable CU count is shown in Fig. 8(a). A set of CU scaling behavior values are calculated using the template shown in Fig. 8(b).

To predict a target configuration execution time, the CU scaling values are always applied before applying any engine or memory frequency scaling values. The difference between the base and target configuration CU counts determines which, if any, CU scaling values are multiplied with the base execution time. After applying CU scaling values to the base execution time, the resulting scaled value is further scaled using the memory and engine frequency sets corresponding to the target CU count. This yields a target execution time that corresponds to the CU count, engine frequency, and memory frequency of the target configuration.



(a) Execution times. (b) Compute unit scaling values.

Fig. 8: Calculation of CU scaling values.

C. Clustering

K-means clustering is used to create sets of scaling behaviors representative of the training kernels. The training kernels are clustered multiple times to form multiple sets of clusters. Each time, the kernels are clustered based on a different set of the scaling values discussed in Section III-B. Each set of clusters is representative of the scaling behavior of one region of the hardware configuration parameter space. The training kernels are clustered once for each CU count available in the training set hardware configurations and once based on their scaling behavior as CU count is varied. For example, if training data is available for CU counts of 8, 16, 24, and 32, the training kernels will be clustered five times and produce five independent cluster sets. Four sets of clusters account for scaling behavior with varying engine and memory frequency, and one set accounts for varying CU count.

Before each clustering attempt, a feature vector is formed for each training kernel. When forming clusters for a specific CU count, the appropriate engine and memory frequency scaling values (see Fig. 7) are chosen and concatenated together to form the feature vector. When forming the variable CU count clusters, the training kernel CU scaling values (see Fig. 8) are chosen. After each training kernel has a feature vector, the training kernels are clustered based on the values in these feature vectors using K-means clustering. The K-means algorithm will cluster the kernels with similar scaling behaviors together. For example, purely memory bound kernels

TABLE II: Classifier feature list.

VALUInsts	% instructions that are vector ALU instructions
SALUInsts	% instructions that are scalar ALU instructions
VFetchInsts	% instructions that are vector fetch instructions
SFetchInsts	% instructions that are scalar fetch instructions
VWriteInsts	% instructions that are vector write instructions
LDSInsts	% instructions that are local data store insts.
VALUUtilization	% active vector ALU threads in the average wave
VALUBusy	% of time vector instructions are being processed
SALUBusy	% of time scalar instructions are being processed
CacheHit	% L2 cache access that hit
MemUnitBusy	% time the average CU's memory unit is active
MemUnitStalled	% time the memory unit is waiting for data
WriteUnitStalled	% time the average write unit is stalled
LDSBankConflict	% time stalled on LDS bank conflicts
Occupancy	% of maximum number of wavefronts that can be scheduled in any CU (Static per kernel)
V_IPC	Vector instruction throughput
MemTime	Ratio of estimated time processing memory requests to total time
FetchPerLoadByte	Average number of load instructions per byte loaded from memory
WritePerStoreByte	Average number of store instructions per byte stored from memory
NumWorkGroups	Number of work groups (Saturates at 320)
ReadBandwidth	Average read bandwidth used
WriteBandwidth	Average write bandwidth used

will be clustered together and purely compute bound kernels will be clustered together. It is possible for some kernels to be grouped into the same cluster in one set of clusters, but not in another set of clusters. This can happen for many reasons. For example, kernels may scale similar with engine frequency, but differently with CU count. Kernels scaling similarly with engine or memory frequency at low CU counts, but differently at high CU counts can also cause this behavior. Clustering multiple times with different sets of scaling values, rather than using one set of clusters using all scaling values at once, allows these kinds of behaviors to be taken into account.

Each cluster's centroid (i.e., the vector of mean scaling values calculated from kernels belonging to the cluster) is used as its representative set of scaling values. Centroids take the same form as the scaling value matrices shown in either Fig. 7 or Fig. 8, depending on which type of scaling values they were formed from. When a cluster is chosen by a classifier in the final model, its centroid scaling values are applied to scale a base configuration execution time as described in Section III-B.

The number of clusters is a parameter of the K-means clustering algorithm. Using too few clusters runs the risk of unnecessarily forcing kernels with different scaling behaviors into the same cluster. The resulting centroid will not be representative of any particular kernel. On the other hand, using too many clusters runs the risk of overfitting the training set. In addition, the complexity of the classifier increases as the number of clusters increases. Training the classifier becomes more difficult when too many clusters are used.

D. Classifier

After forming representative clusters, the next step is to build classifiers, which are implemented using neural networks, that can map kernels to clusters using performance counter values. One neural network is built and trained per cluster set. The features used as inputs to the neural networks are listed in Table II. We use AMD CodeXL to gather a collection of performance counters for each kernel, and all of our features are either taken directly from these or derived from them. Before training the neural networks, the minimum and maximum values of each feature in Table II are extracted from the training set. Using its corresponding minimum and maximum values, each training set feature value is normalized to fall between 0 and 1. The normalized feature values are then used to train the neural network. This avoids the complications of training the neural network using features with vastly different orders of magnitude. All training set minimum and maximum features are stored away so that the same normalization process can be applied each time the model is used.

The neural network topology is shown in Fig. 9. The neural network outputs one value, between 0 and 1, per cluster in its cluster set. The cluster with the highest output value is selected as the chosen cluster for the kernel. We build a three layer, fully connected neural network. The input layer is linear and the hidden and output layers use sigmoid functions. The topology of the network can be seen in Fig. 9. The number of input nodes is determined by the number of performance counters and the number of output nodes is determined by the number of clusters. We set the number of hidden layer nodes to be equal to the number of output nodes.

After construction, the neural network is used to select which clusters best describe the scaling behavior of the kernel that the user wishes to study. The neural networks use the kernel's normalized feature values, measured on the base hardware configuration, to choose one cluster from each cluster set. The centroids of the selected clusters are used as the kernel's scaling values. The predicted target configuration execution time is then calculated by multiplying the base hardware configuration execution time by the appropriate scaling values, as described in Section III-B.

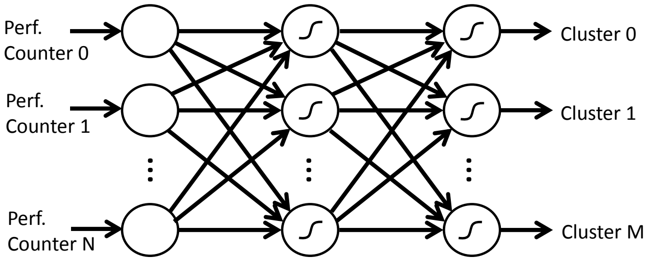


Fig. 9: Neural network topology.

IV. Experimental Setup

In order to validate the accuracy of our performance and power prediction models, we execute a collection of OpenCL$^{\text{TM}}$ applications on a real GPU while varying its hardware configuration. This allows us to measure the power and performance changes between these different configurations.

We used an AMD Radeon$^{\text{TM}}$ HD 7970 GPU as our test platform. By default, this GPU has 32 compute units (2048 execution units), which can run at up to 1 GHz, and 12 channels of GDDR5 memory running at 1375 MHz (yielding 264 GB/s of DRAM bandwidth). All of our experiments were performed on a CentOS 6.5 machine running the AMD Catalyst$^{\text{TM}}$ graphics drivers version 14.2 beta 1.3. We used the AMD APP SDK v2.9 as our OpenCL implementation and AMD CodeXL v1.3 to gather the GPU performance counters for each kernel.

As this work is about predicting performance and power at various hardware configurations, we modified the firmware on our GPU to support changes to the number of active CUs and both the core and memory frequencies. All of our benchmarks are run at 448 different hardware configurations: a range of eight CU settings (4, 8, ..., 32), eight core frequencies (300, 400, ..., 1000 MHz), and seven memory frequencies (475, 625, ..., 1375 MHz). For each kernel in every benchmark at all of these configurations, we gather the OpenCL kernel execution time, performance counters from AMD CodeXL, and the average estimated power of that kernel over its execution.

The AMD Radeon HD 7970 GPU estimates the chip-wide dynamic power usage by monitoring switching events throughout the core. It accumulates these events and updates the power estimates every millisecond [19]. We read these values using the system's CPU. We use these dynamic power values as the basis for our power model because static power does not directly depend on the kernel's execution.

We run GPGPU benchmarks from a number of suites: 13 from Rodinia v2.4[1] [9], eight from SHOC[2] [13], three from OpenDwarfs[3] [18], seven from Parboil[4] [47], four from the Phoronix Test Suite[5], eight from the AMD APP SDK samples[6], and six custom applications modeled after HPC workloads[7]. Because the power readings from our GPU's firmware are only updated every millisecond, we only gather data from kernels that either last longer than 1 ms on our fastest GPU configuration or which are idempotent and can be repeatedly run back-to-back. This yields 108 kernels across these 49 applications, which we do not list due to space constraints.

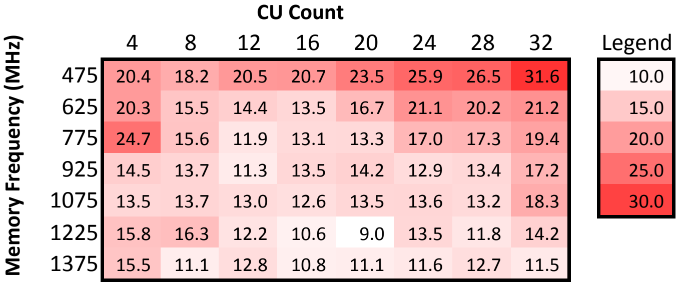
The data from a random 80% of the 108 kernels were used to train and construct our ML model, while the data from the remaining 20% were used for validation.

V. Experimental Results

First, we evaluate the accuracy of our model for performance prediction. Section V-A provides a detailed analysis of performance model accuracy across different base hardware configurations. Section V-B explores the model's sensitivity to the number of representative clusters used. Then, Section V-C evaluates our approach for modeling power consumption.

A. Performance Model Accuracy

The base hardware configuration used is a key parameter of the model and can influence its accuracy. To study the relationship between model accuracy and base hardware configuration, a model was constructed for every one of the 448 possible base hardware configurations. All models were constructed using 12 clusters per cluster set. Each model was then used to predict the execution time of each of the 22 validation kernels on each of the model's 447 possible target hardware configurations; we exclude a model's base hardware configuration from its list of possible target hardware configurations.



Memory Frequency (MHz)	4	8	12	16	20	24	28	32
475	20.4	18.2	20.5	20.7	23.5	25.9	26.5	31.6
625	20.3	15.5	14.4	13.5	16.7	21.1	20.2	21.2
775	24.7	15.6	11.9	13.1	13.3	17.0	17.3	19.4
925	14.5	13.7	11.3	13.5	14.2	12.9	13.4	17.2
1075	13.5	13.7	13.0	12.6	13.5	13.6	13.2	18.3
1225	15.8	16.3	12.2	10.6	9.0	13.5	11.8	14.2
1375	15.5	11.1	12.8	10.8	11.1	11.6	12.7	11.5

Fig. 10: Validation set error heat map at 1000 MHz core frequency. Each point represents the average error of estimating from that point's base configuration to all other configurations (including all other frequencies).

A heat-map of validation set error values from 56 models is shown in Fig. 10. All 56 models have a base configuration engine frequency of 1000 MHz. Their base CU counts and memory frequencies take on all combinations of the eight possible CU counts (4, 8, ..., 32) and the seven possible memory frequencies (475, 625, ..., 1375). We omit varying engine frequency here to make it easier to visualize the data and gain insights into how base configuration impacts model accuracy.

Each entry in Fig. 10 is the average error of one model across all validation kernel and target configuration combinations. High error values show up in the lowest memory frequency base configurations (i.e., configurations with little available memory bandwidth), getting worse at high CU counts (i.e., high compute capabilities). At these extremely low memory bandwidth configurations, the number of kernels that become memory bottlenecked increase. As the percentage of memory bottlenecked kernels becomes greater, the performance counters values, which are gathered on the base hardware configuration, become less diverse between kernels making it difficult for the classifier to assign kernels to different cluster scaling behaviors. This is particularly true for the 32 CU count, 1000 MHz engine frequency, and 475 MHz memory frequency base configuration case where the relatively low memory bandwidth is an extreme mismatch for large number of CUs, which can each generate separate memory requests in parallel. As the compute-to-memory bandwidth resources in the base hardware configuration becomes more balanced, the diversity among the kernel performance counter values increases. As a result, the neural network classifier can more easily distinguish between kernel scaling behaviors and the overall accuracy improves.

The heat-map shown in Fig. 11 shows the average error of the 56 models with base engine frequency of 300 MHz. Due to the reduced engine frequency, the rate of memory requests is much lower than the 1000 MHz base engine frequency configurations. As a result, the models at the top right corner are less imbalanced, causing their errors to be lower than

[1] Rodinia: backprop, b+tree, cfd, gaussian, heartwall, hotspot, kmeans, lavaMD, leukocyte, lud, particlefilter, srad, streamcluster

[2] SHOC: DeviceMemory, MaxFlops, BFS, FFT, GEMM, Sort, Spmv w/ additional CSR-Adaptive algorithm [20], Triad

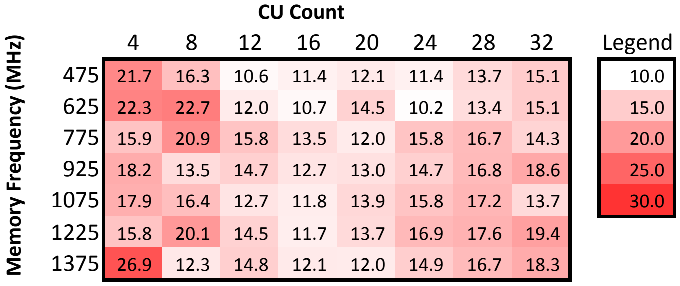
[3] OpenDwarfs: crc, swat, gem

[4] Parboil: stencil, mri-gridding, lbm, sad, histo, mri-q, cutcp

[5] Phoronix: juliaGPU, mandelbulbGPU, smallptGPU, MandelGPU

[6] AMD APP SDK: NBody, BlackScholes, BinomialOption, DCT, EigenValue, FastWalshTransform, MersenneTwister, MonteCarloAsian

[7] Custom: CoMD, LULESH, miniFE, XSBench, BPT [11], graph500 [12]



CU Count

Memory Frequency (MHz)

	4	8	12	16	20	24	28	32
475	21.7	16.3	10.6	11.4	12.1	11.4	13.7	15.1
625	22.3	22.7	12.0	10.7	14.5	10.2	13.4	15.1
775	15.9	20.9	15.8	13.5	12.0	15.8	16.7	14.3
925	18.2	13.5	14.7	12.7	13.0	14.7	16.8	18.6
1075	17.9	16.4	12.7	11.8	13.9	15.8	17.2	13.7
1225	15.8	20.1	14.5	11.7	13.7	16.9	17.6	19.4
1375	26.9	12.3	14.8	12.1	12.0	14.9	16.7	18.3

Legend
10.0
15.0
20.0
25.0
30.0

Fig. 11: Validation set error heat map at 300 MHz frequency.

they were in Fig. 10. However, the entry in the bottom left corner of Fig. 11 is higher. This entry represents a model built with a base configuration of four CUs, a 300 MHz engine frequency, and a 1375 MHz memory frequency, the highest memory bandwidth to compute power configuration. In this base hardware configuration, only the most memory bound kernels will stress the memory system. This again results in less diversity in performance counter signatures similarly to the previously discussed high error base configuration, but with a large fraction of kernels limited entirely by compute throughput. This again means that the classifier will have a harder time distinguishing between kernel scaling behaviors using the base configuration's performance counter values.

We generated heat-maps similar to Fig. 10 and Fig. 11 using each of the remaining possible base hardware configuration engine frequencies. The results of these heat-maps are summarized in Fig. 12. Each box and whisker set shows the distribution of the average error of models with a common base engine frequency. The lines through the middle of the boxes represent the distribution averages. The ends of the boxes represent the standard deviation of the distributions and the ends of the whiskers represent the maximum and minimum of the errors. The best distribution is seen with a 500 MHz base engine frequency, which supports the idea that configurations in the middle of the design space make better base hardware configurations. However, the distributions also show that even for more extreme base engine frequencies, it is still possible to find a good base hardware configuration by choosing the appropriate base CU count and memory frequency to balance out the configuration.

B. *Sensitivity to the Number of Clusters*

In this section, we discuss model accuracy as we vary the number of clusters per set. We built models using 2, 4, . . . , 20 clusters per set and five different base hardware configurations for a total of 50 models. We then analyzed each model's error for the 447 possible target hardware configurations (we exclude the base configuration from the targets). Section V-B1 discusses model accuracy as cluster count is varied and Section V-B2 discusses how the major sources of modeling errors shift with cluster count.

1) Overall Error: Fig. 13 shows the average error across the validation kernel set as the number of clusters per cluster set varies for five base hardware configurations. The x-axis is labeled with the model base hardware configuration. The labels list the base CU count first, engine frequency second, and memory frequency third. The average error across the five base configurations is shown on the far right. Because the cluster centroids and neural network weights are randomly initialized, the error values include some random variance. As shown earlier, the error is higher for models generated from base hardware configurations with imbalanced compute to memory bandwidth ratios. In particular, using a 32 CU, 1000 MHz engine frequency and 475 MHz memory frequency configuration yields error values consistently higher across all cluster settings explored. However, the overall error trend across the sampled base hardware configurations is similar. Model error trends downward until around 12 clusters. For cluster counts greater than 12, error remains relatively constant.

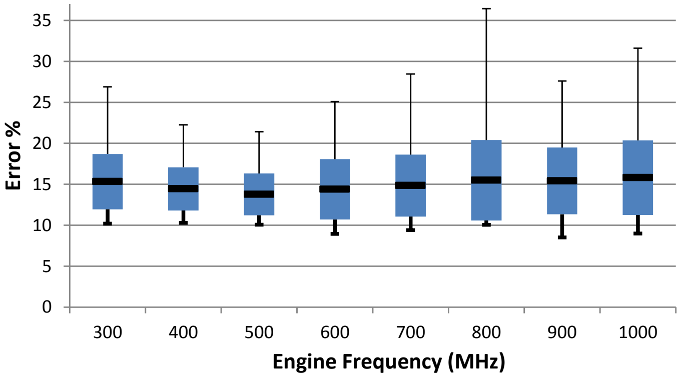


Fig. 12: Error vs. base hardware configuration.

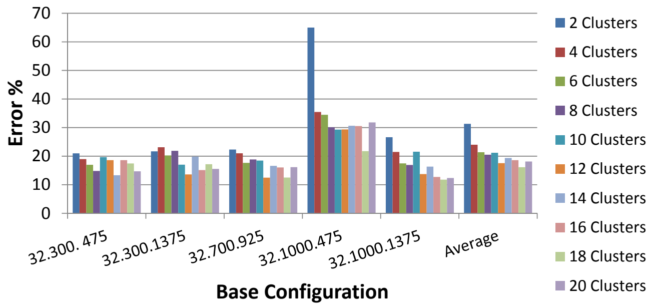


Fig. 13: Validation set error variation across cluster count.

Fig. 14 provides a more detailed breakdown of validation set error distribution as the number of clusters is varied. The distributions of two, six, and twelve cluster count models are shown as histograms. For each cluster count, we aggregated the error values across the five base configurations shown

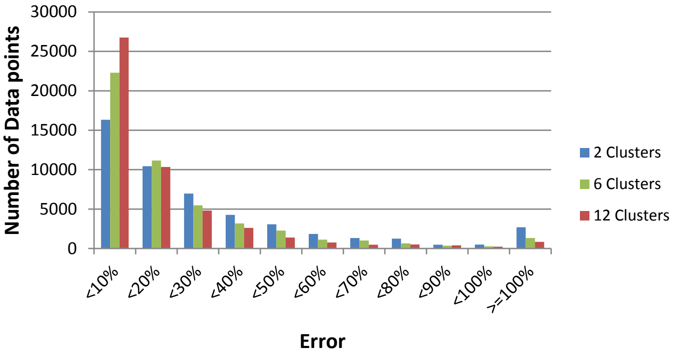


Fig. 14: Validation set error distribution.

in Fig. 13. Because these models were evaluated using 22 validation kernels and 447 target configurations, a total of 49,170 (5×22×447) error values are represented for each of the three cluster counts. The first 10 bins of the histogram represent 10 percent error ranges (e.g., the first bin contains all data points with less than 10% error, the second bin contains all data points with greater than or equal to 10% error and less than 20% error, etc.). The final bin contains all data points with greater than or equal to 100% error. Going from low to high error bins, the number of model predictions falling in each bin exhibits exponential decay (until the last bin, which corresponds to a greater range of values than the other bins). The number of data points in low error bins is higher and the rate of decay is faster for models with more clusters.

2) Sources of Errors: There are two major sources of error in the model. The first is poorly formed clusters. This can be caused by a poor training set, a typical concern encountered when applying machine learning techniques that require a training set. The number of clusters created by the clustering algorithm will also have a large impact on model accuracy. Using too few clusters will fail to capture the diverse range of scaling behaviors. Using too many clusters introduces the possibility of over-fitting the model to the training set. The second source of modeling error comes from the classification step. Even with a set of well-formed clusters, there will be times the neural network classifier has difficulty classifying a kernel to its appropriate cluster. Misclassifying a kernel will often result in large performance prediction error.

To gain insight into how these two sources of error behave, we construct some models with an oracle classifier in place of the neural network and compare them to our original model described in Section III. The oracle models can only be applied to the training data set it was constructed with. For any kernel in the training data set, an oracle classifier chooses the exact cluster that the k-means cluster algorithm assigned it to, while the original, unmodified model uses the neural network classifier to pick an appropriate cluster based off performance counter values. When applying a model with an oracle classifier to its own training set, all error can be attributed to clustering. Comparing the oracle and unmodified models allows us to gain insight into the interaction between the clustering and classification errors. Because the models are only being applied to the training data set in this test, we can ignore the possibilities of a non-representative training set and of over-fitting.

Both types are constructed using 2, 4, . . . , 20 clusters for the five base hardware configurations shown in Fig. 13. We use the models to predict the execution times of each kernel in the training data set for 447 different target hardware configurations and compare the oracle and unmodified model errors. The results are shown Fig. 15. We averaged error across base configurations, kernels, and target hardware configurations to more clearly display accuracy vs. number of clusters for unmodified, neural network models and oracle models. The oracle model error monotonically decreases as the number of clusters increases and will decrease to zero as the number of clusters approaches the number of training kernels. If each kernel has its own cluster, there will be no averaging of scaling surfaces within the clusters. This means that any kernels scaling behavior in the training set can be perfectly reproduced. The oracle classifier will never misclassify. However, the neural network classifier's task becomes more difficult as the number of clusters grows. This is demonstrated empirically in Fig. 15. Notice that the neural network classifier model error remains relatively unchanged at eight or more clusters despite the fact that the oracle model error continues to decrease. These results demonstrate that while a greater number of clusters leads to more precise clusters, it also complicates kernel classification. Eventually the potential accuracy gains offered by additional clusters is offset by increased classification errors.

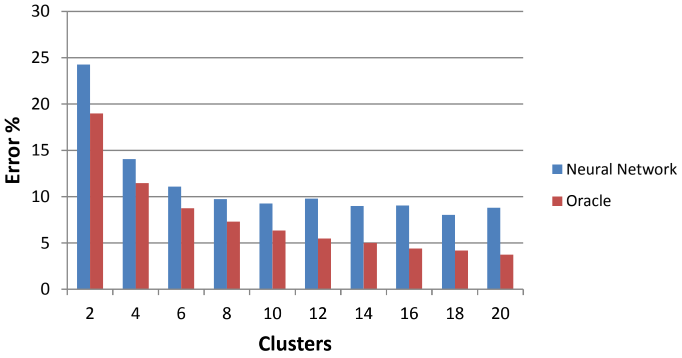


Fig. 15: Training set error variation with number of clusters.

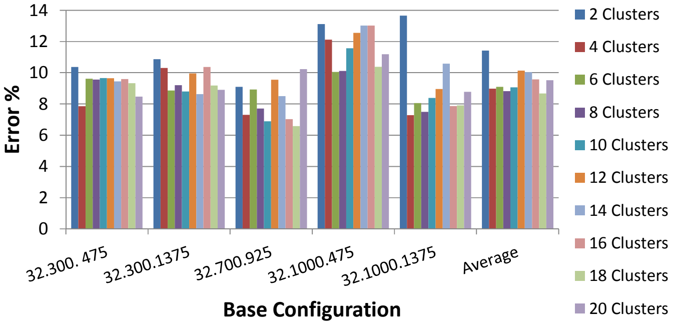


Fig. 16: Power model validation set error.

C. Power Model

In addition to performance modeling, we applied our methodology to create power models. We studied the power model accuracy versus the number of clusters using the same experimental set up used for the performance modeling experiments. We gathered the power values from hardware power monitors in our GPU, which are deterministic estimates of dynamic power based on switching activity in the cores. As such, we are training on and validating against a digital model, rather than analog measurements. This may make it easier to predict, as the values used will be less noisy.

Just like with the performance models, power models were constructed using 2, 4, . . . , 20 clusters for five base hardware configurations. The power models' errors on the validation set is presented in Fig. 16. As before, there is some randomness in the results due to the randomized initial values of the cluster centroids and neural network weights. The errors of the models are fairly similar when more than two clusters are used. This suggests that the power behaviors are less diverse than the performance behaviors. The power model validation

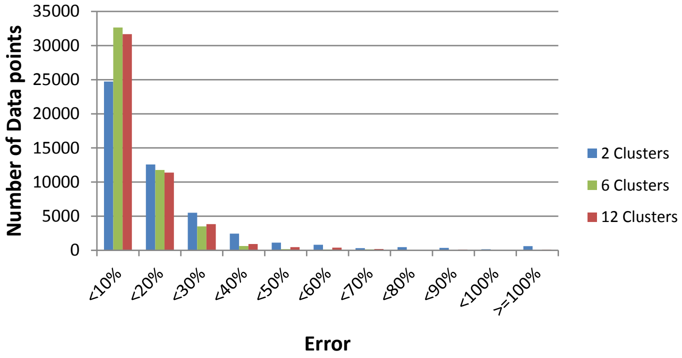


Fig. 17: Power model error distribution.

set errors, which are under 10% for most cluster configurations, are also lower than the performance model validation errors. In addition, the variability of the error across base hardware configurations is much smaller for power than it was for performance modeling.

The power model validation error distribution is shown as a histogram in Fig. 17. Compared to the distribution of errors in the performance model, the number of tests in the low error bins is much higher. There is a large improvement from two to six clusters and very little change from six to 12 clusters, which suggests that relatively few clusters are need to capture the power scaling behaviors.

VI. Related Work

Performance and power prediction are important topics, and Eeckhout gives an excellent overview of the literature in this area [16]. The bulk of these works focus on CPUs, which were traditionally a system's primary computation device. Unfortunately, CPU-focused techniques are inadequate for modeling GPUs. Mechanistic models are necessarily tied to the CPU's microarchitecture [17], [48], while more abstract models often rely on low-level data from the hardware or simulator [15], [25], [33].

GPU performance and power estimation often uses low-level tools like GPGPU-Sim [8], Multi2Sim [50], and Barra [10] for performance and GPUWattch [36] for power. As described earlier, these tools run many orders of magnitude slower than native hardware. This prevents their use for online prediction and often requires some kind of sampling mechanism in order to complete large benchmarks in a reasonable amount of time.

Low-level simulations offer excellent visibility into microarchitectural details, but our work instead focuses on faster, higher-level analyses. Besides online estimation, higher-level models can also help with design space exploration and pruning in order to reduce the number of low-level simulations that must be configured and executed. As an example, an early version of the technique detailed in this paper was used to explore processing-in-memory designs over multiple future silicon generations [53]. The usefulness of high-level models has led to many different developments which attempt to answer slightly different questions.

One common use for such models is to pinpoint software inefficiencies for programmers. Our model can estimate what the application's performance would be at other hardware configurations, but it does not directly point out bottlenecks. As an example of this type of tool, Lai and Seznec describe TEG, which uses simulation traces to pinpoint bottlenecks in GPGPU kernels [32]. Karami et al. describe a linear regression model with the same goal that uses performance counters as its input [28]. Neither is designed to predict performance across machine configurations, and as such, our model is somewhat complementary with these.

Luo and Suda describe a GPU performance and energy estimation model based on static analyses [38]. While they show good accuracy on older GPUs, their technique would not deal well with advances like multi-level caches. Baghsorkhi et al. describe a more advanced static predictor which uses symbolic analyses to estimate how regions of code would exercise the underlying hardware [6]. Both of these techniques are useful for quickly analyzing how an application would perform on specific hardware configurations, but their reliance on static analyses limits their abilities to estimate the effect of different inputs. This would further make it difficult to use these methods for online estimates.

Ma and Chamberlain describe an analytic model that focuses on memory-bound applications that are primarily dominated by hash-table accesses [40]. Their model shows good results and can be used to predict performance across microarchitectural parameters. Unfortunately, their technique is not applicable for general programs.

Hong et al. built a mechanistic performance model that views the kernel execution as a mix of memory- and compute-parallel regions [23]. This was later integrated with parameterized power models [24]. Their original model required static program analyses, limiting its ability to deal with dynamic effects like branch divergence. Sim et al. extended these models to account for such things, including cache effects [45]. They use average memory access time to account for caches, which would be unable to model the changes in cache hit rate demonstrated in Fig. 1(d). In a similar manner, the model presented by Zhang et al. [54] predicts execution times using a linear combination of time spent on different instruction types. We initially attempted to use models such as these, but we repeatedly ran into kernels that could not be accurately predicted due to irregular scaling that could not easily be taken into account with performance counters or static analyses.

In contrast, CuMAPz describes an analytic performance model that takes, as one of its inputs, a memory trace [30]. This allows it to accurately simulate cache behavior and some of the other issues that cause irregular scaling patterns (such as PCIe® bus limitations due to poor memory placement). However, memory traces are difficult and time-consuming to create, negating some of the benefit of high-level modeling. Our model tries to avoid this overhead while still taking into account non-obvious scaling patterns.

Ma et al. use linear regression models to estimate the performance and power of GPU applications in order to decide where (and how) to run GPGPU programs [39]. Bailey et al. use clustering and multivariate linear regression towards a similar goal [7]. These works are an excellent demonstration of the benefits of online performance and power prediction. Ma et al. show that their tool allows them to save a significant amount of energy over blindly assigning work to the CPU or GPU and continuously running at maximum frequency, while Bailey et al. demonstrate power capping with limited performance losses. However, the model of Ma et al., however, was only validated on a first-generation GPGPU (which is significantly simpler than modern GPGPUs), and neither model studies changing the number of active CUs.

The approaches in Stargazer [27] and Starchart [26] reduce the number of simulation points needed to explore a GPU's design space by building power and performance models from a small number of training points. This is a slightly different way of approaching design space explorations than what we present. Rather than requiring a full design space exploration (whether using a low-level or high-level model), these techniques help guide the designer towards more optimal design points. This is beneficial when performing design space explorations and could be used in conjunction with our model to further reduce the amount of time required for broad explorations. Among the other questions our model attempts to solve, however, these methods are not directly applicable for online analyses, as they still require multiple runs to gather training points for a particular run of the application.

VII. Conclusion and Future Work

In this work, we presented a high-level GPGPU performance and power predictor. Our predictor uses performance counter readings gathered at one hardware configuration to estimate the performance and power of the GPGPU kernel at multiple other hardware configurations. The approach uses K-means clustering to form a collection of representative scaling behaviors from a training set. Neural network classifiers then map performance counter values to scaling behaviors. This requires no source code or binary analysis, but it still accurately predicts application scaling trends on complex modern GPUs.

We demonstrated that this technique could estimate performance with an average error of 15% across a frequency range of 3.3×, a bandwidth range of 2.9×, and an 8× difference in number of CUs. Our dynamic power estimation model has an average error of only 10% over the same range.

There are a number of future directions for this work. While Section II gives example uses for high-level models, we focused on the accuracy of our predictions. Building an online control mechanism using our predictor is a potentially fruitful follow-on project. We would also like to add and validate more dimensions to our existing model. For instance, we could study the effects of cache size or PCIe® bus speed. In addition, our current model focuses on predicting within the configuration space of current hardware. Extrapolating outside this space is not well studied but would be useful.

Acknowledgment

AMD, the AMD Arrow logo, Radeon, Catalyst, and combinations thereof are trademarks of Advanced Micro Devices, Inc. OpenCL is a trademark of Apple, Inc. used by permission by Khronos. Xbox One is a trademark of the Microsoft Corporation. PlayStation is a trademark or registered trademark of Sony Computer Entertainment, Inc. PCIe is a registered trademark of the PCI-SIG Corporation. Other names used in this publication are for identification purposes only and may be trademarks of their respective companies.

References

[1] Advanced Micro Devices, Inc., "AMD PowerTune Technology," http://www.amd.com/Documents/amd_powertune_whitepaper.pdf, March 2012.

[2] ——, "AMD A-Series APU Processors," http://www.amd.com/en-us/products/processors/desktop/a-series-apu#2, January 2015.

[3] ——, "AMD Radeon$^{\text{TM}}$ R9 Series Graphics," http://www.amd.com/en-us/products/graphics/desktop/r9#7, January 2015.

[4] AJ KleinOsowski and D. J. Lilja, "MinneSPEC: A New SPEC Benchmark Workload for Simulation-Based Computer Architecture Research," *Computer Architecture Letters*, vol. 1, no. 1, 2002.

[5] S. Anthony, "Xbox One vs. PS4: How the Final Hardware Specs Compare," http://www.extremetech.com/gaming/156273-xbox-720-vs-ps4-vs-pc-how-the-hardware-specs-compare, November 2013.

[6] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W. W. Hwu, "An Adaptive Performance Modeling Tool for GPU Architectures," in *Proc. of the Symp. on Principles and Practice of Parallel Programming (PPoPP)*, 2010.

[7] 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 *Proc. of the Int'l Conf. on Parallel Processing (ICPP)*, 2014.

[8] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA Workloads Using a Detailed GPU Simulator," in *Proc. of the Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, 2009.

[9] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *Proc. of the IEEE Int'l Symp. on Workload Characterization (IISWC)*, 2009.

[10] S. Collange, M. Daumas, D. Defour, and D. Parello, "Barra: A Parallel Functional Simulator for GPGPU," in *Proc. of the Int'l Symp. on Modeling, Analysis and Simulation of Computer and Telecommunicatoin Systems (MASCOTS)*, 2010.

[11] M. Daga and M. Nutter, "Exploiting Coarse-grained Parallelism in B+ Tree Searches on an APU," in *Proc. of the Workshop on Irregular Applications: Architectures & Algorithms (IA3)*, 2012.

[12] M. Daga, M. Nutter, and M. Meswani, "Efficient Breadth-First Search on a Heterogeneous Processor," in *Proc. of the IEEE Int'l Conf. on Big Data (IEEE BigData)*, 2014.

[13] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter, "The Scalable HeterOgeneous Computing (SHOC) Benchmark Suite," in *Proc. of the Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU)*, 2010.

[14] H. David, C. Fallin, E. Gorbatov, U. R. Hanebutte, and O. Mutlu, "Memory Power Management via Dynamic Voltage/Frequency Scaling," in *Proc. of the Int'l Conf. on Autonomic Computing (ICAC)*, 2011.

[15] C. Dubach, T. M. Jones, and M. F. O'Boyle, "Microarchitectural Design Space Exploration Using An Architecture-Centric Approach," in *Proc. of the Int'l Symp. on Microarchitecture (MICRO)*, 2007.

[16] L. Eeckhout, *Computer Architecture Performance Evaluation Methods*. Morgan & Claypool, 2010.

[17] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith, "A Mechanistic Performance Model for Superscalar Out-of-Order Processors," *ACM Transactions on Computer Systems*, vol. 27, no. 2, pp. 3:1–3:37, May 2009.

[18] W. Feng, H. Lin, T. Scogland, and J. Zhang, "OpenCL and the 13 Dwarfs: A Work in Progress," in *Proc. of the Int'l Conf. on Performance Engineering (ICPE)*, 2012.

[19] D. Foley, "AMD's "Llano" Fusion APU," Presented at Hot Chips, 2011.

[20] J. L. Greathouse and M. Daga, "Efficient Sparse Matrix-Vector Multiplication on GPUs using the CSR Storage Format," in *Proc. of the Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2014.

[21] A. Gutierrez, J. Pusdesris, R. G. Dreslinski, T. Mudge, C. Sudanthi, C. D. Emmons, M. Hayenga, and N. Paver, "Sources of Error in Full-System Simulation," in *Proc. of the Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, 2014.

[22] K. Hinum, "AMD E-Series E1-6010 Notebook Processor," http://www.notebookcheck.net/AMD-E-Series-E1-6010-Notebook-Processor.115407.0.html, April 2014.

[23] S. Hong and H. Kim, "An Analytical Model for a GPU Architecture with Memory-level and Thread-level Parallelism Awareness," in *Proc. of the Int'l Symp. on Computer Architecture (ISCA)*, 2009.

[24] ——, "An Integrated GPU Power and Performance Model," in *Proc. of the Int'l Symp. on Computer Architecture (ISCA)*, 2010.

[25] E. Ĭpek, S. A. McKee, R. Caruana, B. R. de Supinski, and M. Schulz, "Efficiently Exploring Architectural Design Spaces via Predictive Modeling," in *Proc. of the Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2006.

[26] W. Jia, K. Shaw, and M. Martonosi, "Starchart: Hardware and Software Optimization Using Recursive Partitioning Regression Trees," in *Proc. of the Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, 2013.

[27] W. Jia, K. A. Shaw, and M. Martonosi, "Stargazer: Automated Regression-Based GPU Design Space Exploration," in *Proc. of the Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, 2013.

[28] A. Karami, S. A. Mirsoleimani, and F. Khunjush, "A Statistical Performance Prediction Model for OpenCL Kernels on NVIDIA GPUs," in *Proc. of the Int'l Symp. on Computer Architecture and Digital Systems (CADS)*, 2013.

[29] O. Kayiran, A. Jog, M. T. Kandemir, and C. R. Das, "Neither More Nor Less: Optimizing Thread-level Parallelism for GPGPUs," in *Proc. of the Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, 2013.

[30] Y. Kim and A. Shrivastava, "CuMAPz: A Tool to Analyze Memory Access Patterns in CUDA," in *Proc. of the Design Automation Conference (DAC)*, 2011.

[31] S. Kosonocky, "Practical Power Gating and Dynamic Voltage/Frequency Scaling," Presented at Hot Chips, 2011.

[32] J. Lai and A. Seznec, "Break Down GPU Execution Time with an Analytical Method," in *Proc. of the Workshop on Rapid Simulation and Performance Evaluation (RAPIDO)*, 2012.

[33] B. C. Lee and D. M. Brooks, "Accurate and Efficient Regression Modeling for Microarchitectural Performance and Power Prediction," in *Proc. of the Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2006.

[34] M. Lee, S. Song, J. Moon, J. Kim, W. Seo, Y. Cho, and S. Ryu, "Improving GPGPU Resource Utilization Through Alternative Thread Block Scheduling," in *Proc. of the Int'l Symp. on High Performance Computer Architecture (HPCA)*, 2014.

[35] S. Lee and W. W. Ro, "Parallel GPU Architecture Simulation Framework Exploiting Work Allocation Unit Parallelism," in *Proc. of the Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, 2013.

[36] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi, "GPUWattch: Enabling Energy Optimizations in GPGPUs," in *Proc. of the Int'l Symp. on Computer Architecture (ISCA)*, 2013.

[37] H. Li, S. Bhunia, Y. Chen, T. N. Vijaykumar, and K. Roy, "Deterministic Clock Gating for Microprocessor Power Reduction," in *Proc. of the Int'l Symp. on High Performance Computer Architecture (HPCA)*, 2003.

[38] C. Luo and R. Suda, "A Performance and Energy Consumption Analytical Model for GPU," in *Proc. of the Int'l Conf. on Dependable, Autonomic and Secure Computing (DASC)*, 2011.

[39] K. Ma, X. Li, W. Chen, C. Zhang, and X. Wang, "GreenGPU: A Holistic Approach to Energy Efficiency in GPU-CPU Heterogeneous Architectures," in *Proc. of the Int'l Conf. on Parallel Processing (ICPP)*, 2012.

[40] L. Ma and R. D. Chamberlain, "A Performance Model for Memory Bandwidth Constrained Applications on Graphics Engines," in *Proc. of the Int'l Conf. on Application-Specific Systems, Architectures and Processors (ASAP)*, 2012.

[41] A. McLaughlin, I. Paul, J. L. Greathouse, S. Manne, and S. Yalamanchili, "A Power Characterization and Management of GPU Graph Traversal," in *Workshop on Architectures and Systems for Big Data (ASBD)*, 2014.

[42] I. Paul, S. Manne, M. Arora, W. L. Bircher, and S. Yalamanchili, "Cooperative Boosting: Needy Versus Greedy Power Management," in *Proc. of the Int'l Symp. on Computer Architecture (ISCA)*, 2013.

[43] M. Peres, "Reverse Engineering Power Management on NVIDIA GPUs - Anatomy of an Autonomic-Ready System," in *Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*, 2013.

[44] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically Characterizing Large Scale Program Behavior," in *Proc. of the Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2002.

[45] J. Sim, A. Dasgupta, H. Kim, and R. Vuduc, "A Performance Analysis Framework for Identifying Potential Benefits in GPGPU Applications," in *Proc. of the Symp. on Principles and Practice of Parallel Programming (PPoPP)*, 2012.

[46] V. Spiliopoulos, S. Kaxiras, and G. Keramidas, "Green Governors: A Framework for Continuously Adaptive DVFS," in *Proc. of the International Green Computing Conference and Workshops (IGCC)*, 2011.

[47] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W. W. Hwu, "Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing," University of Illinois at Urbana-Champaign, Tech. Rep. IMPACT-12-01, March 2012.

[48] B. Su, J. L. Greathouse, J. Gu, M. Boyer, L. Shen, and Z. Wang, "Implementing a Leading Loads Performance Predictor on Commodity Procesors," in *Proc. of the USENIX Annual Technical Conf. (USENIX ATC)*, 2014.

[49] B. Su, J. Gu, L. Shen, W. Huang, J. L. Greathouse, and Z. Wang, "PPEP: Online Performance, Power, and Energy Prediction Framework and DVFS Space Exploration," in *Proc. of the Int'l Symp. on Microarchitecture (MICRO)*, 2014.

[50] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli, "Multi2Sim: A Simulation Framework for CPU-GPU Computing," in *Proc. of the Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, 2012.

[51] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe, "SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling," in *Proc. of the Int'l Symp. on Computer Architecture (ISCA)*, 2003.

[52] J. J. Yi, S. V. Kodakara, R. Sendag, D. J. Lilja, and D. M. Hawkins, "Characterizing and Comparing Prevailing Simulation Techniques," in *Proc. of the Int'l Symp. on High Performance Computer Architecture (HPCA)*, 2005.

[53] D. P. Zhang, N. Jayasena, A. Lyashevsky, J. L. Greathouse, L. Xu, and M. Ignatowski, "TOP-PIM: Throughput-Oriented Programmable Processing in Memory," in *Proc. of the Int'l Symp. on High Performance Parallel and Distributed Computing (HPDC)*, 2014.

[54] Y. Zhang and J. D. Owens, "A Quantitative Performance Analysis Model for GPU Architectures," in *Proc. of the Int'l Symp. on High Performance Computer Architecture (HPCA)*, 2011.

[55] Y. Zhang, Y. Hu, B. Li, and L. Peng, "Performance and Power Analysis of ATI GPU: A Statistical Approach," in *Proc. of the Int'l Conf. on Networking, Architecture and Storage (NAS)*, 2011.