Stargazer: Automated Regression-Based GPU Design Space Exploration

Wenhao Jia
Princeton University
wjia@princeton.edu

Kelly A. Shaw
University of Richmond
kshaw@richmond.edu

Margaret Martonosi
Princeton University
mrm@princeton.edu

Abstract

Graphics processing units (GPUs) are of increasing interest because they offer massive parallelism for high-throughput computing. While GPUs promise high peak performance, their challenge is a less-familiar programming model with more complex and irregular performance trade-offs than traditional CPUs or CMPs. In particular, modest changes in software or hardware characteristics can lead to large or unpredictable changes in performance. In response to these challenges, our work proposes, evaluates, and offers usage examples of Stargazer[1], an automated GPU performance exploration framework based on stepwise regression modeling. Stargazer sparsely and randomly samples parameter values from a full GPU design space and simulates these designs. Then, our automated stepwise algorithm uses these sampled simulations to build a performance estimator that identifies the most significant architectural parameters and their interactions. The result is an application-specific performance model which can accurately predict program runtime for any point in the design space. Because very few initial performance samples are required relative to the extremely large design space, our method can drastically reduce simulation time in GPU studies. For example, we used Stargazer to explore a design space of nearly 1 million possibilities by sampling only 300 designs. For 11 GPU applications, we were able to estimate their runtime with less than 1.1% average error. In addition, we demonstrate several usage scenarios of Stargazer.

1. Introduction

GPUs are increasingly being used as general-purpose parallel computing platforms. While originally targeted at graphics applications, these massively-parallel, throughput-oriented systems offer the potential for very high performance over a range of applications. However, unlike traditional CPUs or CMPs, GPUs offer a less-familiar programming model with more complex and irregular performance trade-offs. Moderate changes in software attributes (such as data layout and memory reference patterns) or hardware characteristics (such as size of memory structures and interconnect bandwidth) can lead to large or unpredictable performance changes [4, 18, 24, 28].

From a software perspective, the lack of resource abstraction and virtualization forces GPU programmers—unlike programmers for general-purpose CPUs—to take hardware characteristics into consideration even when writing GPU programs just for correctness. Moreover, to optimize performance, programs have to be tailored for specific GPU instances. It is usually very challenging to create a single version of code that performs well on several GPU implementations. Good GPU performance estimation models have the potential to overcome these difficulties by helping developers to predict and optimize program performance and to port existing applications.

From a hardware perspective, the many dimensions of the GPU design space and their intricate interactions make it difficult or time-consuming to predict and optimize the performance of future GPUs under design. With architectural simulations running five to six *orders of magnitude* more slowly than programs on real hardware, each possible design point requires days of simulation time for full-sized data sets. Exploring the full cross-product of many design parameters requires a large number of long-running simulations. Thus, like their software counterparts, hardware designers can also benefit from GPU performance estimators that accurately survey the design space and reduce the number of points for which detailed evaluations are required.

While previous computer architecture research has explored using regression methods [11, 15–17] and machine learning techniques [10, 12] to prune and better understand the design space of general-purpose CPUs, their methods have limited applicability due to the complexity of the GPU design space. In particular, GPU architectural parameters and their interactions exhibit strong and highly nonlinear influence on program runtime, and the relative importance of these parameters is difficult to predict due to unfamiliarity with the platform in the relatively new GPU field. In response to these challenges, our work proposes, evaluates, and offers usage examples of Stargazer, an automated GPU design space exploration framework based on stepwise regression modeling and tailored for the complex GPU design space. Our work has several key novelties and makes significant contributions.

First, we are the first to offer an automated design space explorer geared toward the complex and nuanced GPU arena.

1. Stargazer stands for STAtistical Regression-based GPU Architecture analyZER. The name is inspired by how only a few stars can be used to represent an entire constellation. This is similar to how our regression models offer accurate estimates from a small sample of points in the design space.

Our tool handles stronger and more complex application performance dependence on GPU architectural parameters than work geared at general-purpose CPUs. Second, unlike many previous regression methods, our work is fully automated. By this we mean that our stepwise regression method incrementally evaluates the absolute and adjusted R^2 values of design parameters to determine which should be included into the regression model. Furthermore, we also automate the use of pairwise interactions between parameters. These interactions are much stronger in the GPU design space than those in the CPU field. This helps overcome insufficient design intuition in the relatively new GPU field.

For the platforms evaluated in detail in this paper, we show that GPU performance can be accurately modeled using a small fraction of the entire simulation space. Starting from a 10-dimensional design space of over 933K possible design points, Stargazer can use 300 randomly-chosen points to build an estimator equation that predicts performance with 1.1% average error. From as few as 30–60 data points, it offers less than 5% error for all but one application. This reduces simulation requirements by 4–5 *orders of magnitude* compared to exhaustive design exploration and saves considerably over more targeted, non-exhaustive approaches as well. Our method also offers GPU hardware and software designers useful intuition about the systems they develop.

The rest of the paper is organized as follows. Section 2 reviews prior work. Section 3 and Section 4 introduce background material on regression theory and GPU architectures respectively. Our automated regression-based method is presented in Section 5. Section 6 describes our experimental methodology, and Section 7 evaluates the method. Section 8 offers a few scenarios in which Stargazer can be used. Finally, Section 9 concludes the paper.

2. Prior Work

2.1. GPU Performance and Power Analysis

As GPUs have become more widely used, several simulators have been introduced to perform parameterized studies of the GPU design space. These include GPGPU-Sim, the simulator we use in this research [1]. Another earlier simulator, Qsilver, also addressed basic GPU performance issues [26].

Some research aims to build analytical models for estimating GPU power and performance [6, 7]. If their models are parameterized, they can be used to explore the GPU hardware designs similarly as GPU simulators. However, neither GPU simulators nor these analytical models offer a way to navigate and interpret the complex parameterized GPU design space.

Where GPUs exist to be studied in real systems, performance counters offer another approach for some evaluations. In particular, hardware performance counters for some NVIDIA GPUs are available through their Parallel NSight tool [19]. While these are useful for characterizing points from the GPU design space that have already been instantiated as real GPU systems, they do not allow one to change hardware parameters to consider design points that have not yet been built.

2.2. CPU Design Space Exploration

While design space exploration for GPUs is fairly nascent, prior work considering general-purpose CPUs is relevant. A wide range of methods including machine learning and regression have been used to assist CPU design space exploration. However, the differences between CPU and GPU design spaces (Section 4.3) and/or limitations in these methods make them inefficient or unsuitable for GPUs.

For example, Ipek et al. proposed a method based on artificial neural networks [10]. While this method offers good speedup and accuracy and can be automated, it has an intrinsic feedback loop in which subsequent simulations are selected based on results observed from prior design points. As such, this method is difficult to parallelize, significantly increasing the training time. In contrast, our method can gather a random sampling of simulation points *in parallel* and then perform regression on them to create an accurate model.

Lee and Brooks proposed a regression-based modeling approach to study the CMP design space [15]. Their approach is similar to ours in using regression, and it also achieves good model accuracy with a small number of sampled simulations. However, it differs from ours in that it does not use *stepwise* regression to select relevant parameters. Instead, users must rely on domain-specific knowledge and considerable statistics experience to select parameters and interactions that are included in the model. In the newer and less familiar GPU design space, fewer users can depend on reliable domain knowledge.

Joseph et al. proposed a stepwise regression method to automatically explore a CPU design space [11]. However, each of their parameters can take only one of two possible values. As such, their model is strictly linear in the parameters, taking into account only first-order linear factors. It is unclear whether their method will apply well to design spaces broader than their chosen limited uniprocessor design space. In contrast, our method is still linear, but uses splines so that each parameter can take an arbitrary number of values expressing more complex relationships.

3. Regression Theory

Statistical regression analysis is the study of techniques that relate one dependent variable to a number of independent variables. Among various forms of regression, linear regression is the most commonly used due to its well-behaved and well-studied properties [14].

$$y = \beta_0 + \sum_{i=1}^{n} \beta_i x_i + \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \beta_{i,j} x_i x_j + \epsilon \qquad (1)$$

Equation 1 shows a linear regression model. The x_i terms are the independent variables. Their changing values cause the dependent variable, y, to vary as a response. The model error ϵ is added to describe an individual sample's deviation from what the model predicts. Linear regression theory requires the dependent variable y to be a linear combination of the regression coefficients β_i. In particular, though Equation 1 has

terms that are quadratic in independent variables x_i, the model remains linear as long as it is linear in β_i.

The task of the regression is to determine the coefficients β_i from observed measurements of y and x_i. Least square solvers are usually used to accomplish this. In this paper, we focus on GPU performance modeling, so y represents program runtime and x_i are architectural parameters such as GPU computing resources and memory bandwidth (Section 6.1). Our method is, however, flexible enough to handle more general studies. For example, y can be any measurable hardware metric, such as chip power consumption, while x_i can represent algorithmic choices in addition to architectural parameters.

To account for interplay between basic terms x_i, we can include pairwise first-order interactions as the double summation in Equation 1. The product $x_i x_j$ reflects the interaction between x_i and x_j. However, the number of interaction terms grows much faster than the number of basic terms. Automatically determining which important pairwise interactions to include is a key part of our stepwise regression process presented in Section 5.

Variables in real problems often have nonlinear dependence that cannot be accurately captured by Equation 1. To handle these cases, transformations (f_i and f_j in Equation 2) are applied to x_i. Equation 2 is still a linear regression model because y is still linear in the coefficients β_i. As a result, techniques in linear regression theory can still be applied to the problem while nonlinearity in independent variables is handled.

$$y = \beta_0 + \sum_{i=1}^{n} \beta_i f_i(x_i) + \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \beta_{i,j} f_i(x_i) f_j(x_j) + \epsilon \quad (2)$$

A common choice of f is to use piecewise polynomial functions (splines) due to their flexibility in characterizing any continuous function [20]. In addition to the choice of spline types (e.g. cubic), there is also a choice regarding the number and positions of knots for the spline. Section 5.3 has more details on the particular splines we have chosen and their arguments.

After a model and its β_i are proposed, the coefficient of determination (R^2) is a commonly-used statistical metric for measuring the quality of a model [14]. R^2 reflects how much of the variance of the dependent variable can be explained by changes in the independent variables. However, because R^2 always increases when a new term is included in the model, it does not fully show the accuracy benefit of that term. For this reason, *adjusted* R^2 is often used. Adjusted R^2 measures whether adding an additional parameter in the model brings more benefit than a random variable would have brought [14].

Finally, when there are a large number of independent variables, as is the case in the GPU design space, Equation 2 may have a large number of terms. A stepwise regression method coupled with some measure of fit, such as Akaike Information Criterion (AIC) [25] or adjusted R^2, can be used to judiciously reduce the number of terms by only selecting those that are most useful for model accuracy. Section 5 gives more details on our specific stepwise method tailored for GPU design space exploration. In formal terminology, our method is a forward selection (i.e. start with an empty model and add parameters to it), adjusted R^2-based, stepwise linear regression process which includes first-order interactions.

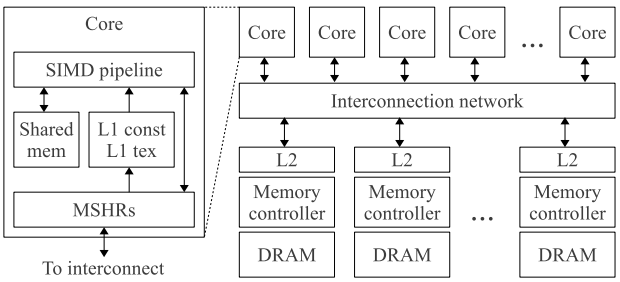


Figure 1: GPU architecture overview

4. GPU Architectures and Programming

4.1. GPU Hardware Architecture Overview

GPU architectures obtain high computational throughput from executing many identical operations simultaneously on different data, typically in a SIMD fashion. Modern GPUs further extend this functionality by replicating the basic SIMD hardware so that multiple, independent SIMD calculations may occur concurrently on different groups of data. While GPUs from different vendors may vary in specifics, our approach is generally applicable. For clarity and specificity, we use NVIDIA terminology in this paper, but the approach can be applied to GPUs from other vendors as well. Figure 1 shows a typical GPU architecture based on NVIDIA GPUs.

The basic SIMD hardware component is called a *core*. Cores contain multiple execution units, where each unit executes the code for a single thread in parallel with other units. The number of such units in a core is referred to as the core's *SIMD width*. The hardware groups threads executing the same code into sets called *warps* and schedules the threads within a warp to execute concurrently using all of the core's SIMD units. Threads all execute the same instruction using different data values from different memory addresses, though cores contain mechanisms to allow branch divergence within the warp [5].

A core contains resources shared by all threads executing concurrently on it. Each core contains a limited number of thread slots which limit the maximum number of concurrent threads. Threads executing the same code are grouped by the programmer into thread blocks; cores support a finite number of these thread blocks. Each thread block contains one or multiple warps. The hardware overlaps the computation and communication phases of different warps to hide memory latency. Cores also contain a variety of storage resources: a register set shared by all threads and software-managed *shared memory* shared among the core's threads to enable fast data access within a thread block. Other resources like caches (instruction, data, constant, texture) and texture units may also be shared by all threads scheduled on a core. The finite amount of shared resources limits the number of threads that can execute simultaneously on a core. In particular, the

smallest of four core resources—the number of thread slots, the number of thread block slots, the register set size, and the shared memory size—dictates the maximum number of thread blocks that can execute concurrently on a core; we call this a core's *(thread) block concurrency*.

Each core also contains a set of Miss Status Holding Registers (MSHRs) for tracking outstanding memory requests. MSHRs and the associate mechanisms reduce the number of memory requests sent out of the core in two ways. First, memory requests from individual threads in the same warp can be joined together into fewer wide memory requests via *intra-warp memory coalescing*. Second, overlapping memory read requests made by multiple warps executing on the same core can be joined together via *inter-warp memory coalescing*.

As shown in Figure 1, modern GPUs contain an array of these cores sharing other chip resources. An interconnection network connects cores to a distributed set of second-level caches and memory modules. Fixed groups of memory modules share one memory controller that sits in front of them and schedules memory requests. The DRAM scheduler queue size in each memory module impacts the capacity to hold outstanding memory requests.

4.2. Programming GPUs

As GPU use has increased, programming environments and abstractions have been introduced. NVIDIA's CUDA is a common method for programming NVIDIA GPUs [23], and OpenCL is an industry effort towards a more portable programming environment [13]. Nonetheless, GPU programming still requires understanding the underlying target hardware. In particular, the code typically indicates how to group computation together on hardware and how to place and access data in the system. In our work, the benchmarks are written using CUDA, but the ideas apply to other environments.

To specify computation, programmers write *kernels* which are invoked from the main program running on a host processor and which execute on the GPU. The kernel specifies operations to be performed from a single thread's point of view. When launching a kernel on a GPU, the code must specify the total number of threads executing the kernel and how to group these threads together. The code must specify the number of threads in each *thread block* and the total number of thread blocks. Threads within the same thread block execute together on a core, sharing the core's resources. They can share data via the core's shared memory and perform barrier synchronization among themselves. Thread blocks from the same kernel can execute on different cores, and multiple thread blocks may execute on a single core. Programmers decide how many threads should be contained in a thread block; this decision, along with the number of registers and the amount of shared memory needed by each thread block, determines the block concurrency.

4.3. GPU Design Space Complexity

While design space explorers are generally useful, this is particularly true in the GPU design space due to its unique complexity. We use a simple program, matrix multiply[2], to highlight some aspects of this complexity.

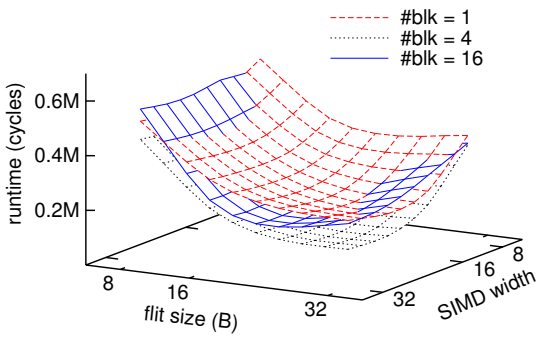


Figure 2: Matrix multiply's runtime variations in a $3 \times 3 \times 3$ design space. Different block concurrency values significantly alter the nonlinear surfaces formed by the runtime.

Figure 2 shows a small $3 \times 3 \times 3$ design space for the matrix multiply program. Each of the 3 chosen parameters may take 3 possible values, and the runtime (z-axis) of the matrix multiply program is plotted as interpolated smooth surfaces. The 3 chosen parameters are the SIMD width representing a GPU's computational power (x-axis), the interconnect flit size representing a GPU's memory bandwidth (y-axis), and the block concurrency (`#blk`) representing a GPU's amount of on-chip resources (surface layers). The figure shows that even for such a small design space and a simple program, the runtime has a nonlinear and non-monotonic dependence on the varying SIMD width and flit size when block concurrency is fixed. Furthermore, changes in block concurrency cause the runtime to respond even more unpredictably. Low block concurrency (e.g. 1) results in higher runtime because there is not enough work to keep SIMD units busy. But high block concurrency (e.g. 16) leads to even worse runtime in some parts of the design space, which may be a result of memory congestion caused by the high number of active threads. The optimal block concurrency highly depends on the values of SIMD width and flit size, indicating strong interactions between these 3 parameters.

There are two unique aspects of the complex GPU design space. First, the lack of resource abstraction and virtualization on GPUs results in highly hardware-dependent, nonlinear program performance variations. GPU programmers are often required to write and optimize programs for specific GPU hardware. However, after a program is written and optimized, it may not run well or even run at all on other GPUs in the same design space. For example, in Figure 2, GPUs with different SIMD width and flit size parameters demand the programmers to carefully adjust matrix multiply's block concurrency. However, the block concurrency is at the same time limited by available on-chip resources. Such hardware-dependent program tuning is much more common in GPUs than CPUs, which are often helped by the use of cache and memory hierarchy. A GPU

2. We modified matrix multiply from the NVIDIA CUDA SDK [22] to use an 8×8 thread block size instead of the default 16×16 thread block size to achieve a high block concurrency.

design space explorer must efficiently handle such sensitive performance variations, and our tool leverages the expressive power of splines to achieve that.

The second aspect of GPU design space complexity comes from unfamiliarity with the platform. Whereas CPUs have been under extensive study for decades, and many effects of their architectural parameters are well known, it is often unclear to GPU programmers which chip resources may be limiting the performance of a particular application. Meanwhile, some programs in our experiment suite also show unusual reliance on some unexpected resources (e.g. `nw`'s runtime heavily depends on the number of shared memory ports, as shown in Section 7.1). Furthermore, the matrix multiply example shows that GPU architectural parameters often have strong interactions. It is non-trivial for GPU programmers to predict and estimate such unexpected architectural bottlenecks. For this reason, unlike some prior work which asked users to select important parameters and interactions, our tool is fully automated.

5. Stargazer: A Stepwise Regression Method for GPU Design Space Exploration

5.1. Phase 1: Sampled Simulation Runs

Algorithm 1 Pseudocode of the stepwise regression method

Require: Parameters P_1, P_2, ..., P_n.
Require: Ranges of values S_1, S_2, ..., S_n.
Require: The design space with $|S_1| \times |S_2| \times \ldots \times |S_n|$ points.
  Randomly sample k points from the design space
  Simulate or measure each of these k samples
  Model $M = \emptyset$, the remaining variable set $T = \{P_1, P_2, \ldots, P_n\}$
  For each P_i in T: Form tentative model M_i with a single term P_i
  Among all M_i: Find the one with the highest R^2, assign it to M, and remove corresponding P_i from T
  M now stores the initial single-term model
  while T is not empty **do**
    For each P_j in T: Form tentative model M_j equal to M with an additional basic term P_j
    Among all M_j: Find the one with the highest adjusted R^2
    if (this model's adjusted R^2) $-$ (R^2 of M) $> \theta$ **then**
      Assign this model to M, remove corresponding P_j from T
      $R = \{P_m | P_m$ is already in M and $P_m \neq P_j\}$
      while R is not empty **do**
        For each P_k in R: Form tentative model M_k equal to M with an additional interaction term $P_j : P_k$
        Among all M_k: find the one with the highest adjusted R^2
        if (this model's adjusted R^2) $-$ (R^2 of M) $> \phi$ **then**
          Assign M_k to M, remove corresponding P_k from R
        else
          Break the while loop
        end if
      end while
    else
      Break the while loop
    end if
  end while
  Return M as the final model

Algorithm 1 gives an overview of our stepwise regression method for GPU performance estimation. We begin by identifying the parameter space of interest for this design. Each design issue or parameter of interest (P_i) can be considered one dimension in the design space. For each dimension, a set of possible parameter value settings (S_i) is identified. These choices could be evenly-spaced points between minimum and maximum values, progressions that rely on doubling values, user-specified values, etc.

All the possible parameter values on all the possible dimensions can be thought of as a multi-dimensional enumeration of possible design points. For the GPUs we have considered, it is common for ten or more parameters or dimensions to be of interest, each of which may have many possible value settings. As a result, an *exhaustive* simulation-based design study might require millions of points to be simulated.

Stargazer instead samples points from the full design space. We sample randomly and uniformly from the full space until a desired number of points has been collected. The user of our method can specify this either as a total number of points to simulate, as a fraction of the total design space to simulate, or as a fraction of each dimension to simulate. In our experiments discussed in Section 7, fewer than 300 simulations (0.03% of the space) are enough for excellent accuracy.

For each sampled point, the user runs a simulation or measurement through their evaluation tool. In our case, we use GPGPU-Sim [1], but any simulator is appropriate. In addition, if the parameters under study can be modified during real-system runs (e.g. configurable memory sizes, numbers of cores used, or other algorithmic choices), then Stargazer can use results from real-machine runs instead of simulations. Our method is orthogonal to this choice. The performance measurements resulting from the simulation or measurement runs are stored for use by the subsequent steps of the regression method.

5.2. Phase 2: Applying Stepwise Algorithm

Once a set of performance measurements is available for a very lightly-sampled view of the design space, the statistical analysis determines which design parameters are most influential to program/system performance and how they interact. This process repetitively selects the next most significant parameter, adjusts the model to include it, and then considers whether its "interaction terms" are useful.

In particular, we start with a number of models, each comprising only one of the studied parameters. We compute the R^2 of these models as a gauge of each individual parameter's contribution to application performance variations. Parameters that have bigger impact on application performance will produce single-term models that have higher R^2. We choose the parameter offering the largest R^2 and use it as the initial parameter for the regression model. Using this single parameter, we develop a performance estimation equation based on natural cubic splines (Section 5.3).

Once the first parameter has been chosen, we consider the next most important parameter based on *adjusted* R^2. Specifically, we augment the currently-obtained intermediate regression model with each of the remaining parameters and

calculate the adjusted R^2 of each of these proposed models. The highest adjusted R^2 indicates the most useful next parameter among those yet unchosen. If the difference between this parameter's adjusted R^2 and the R^2 of the pre-augmentation model is above a specified threshold θ, we pick this parameter as the next basic term to be incorporated into the cubic spline regression model. The currently-obtained intermediate model is changed to this augmented model and R^2 is updated.

Every time a new parameter is added to the model, we must also consider its interaction with parameters that have been chosen earlier. In particular, we consider *interaction factors* that indicate how the newly added parameter cross-correlates with each of the already chosen parameters to contribute to performance. In our approach, we consider only pairwise interactions. While three-way (and beyond) interaction factors are also possible, they have not been significant in our experience, and consequently our default method does not include them. For each interaction factor, we compare its adjusted R^2 to a threshold value ϕ to determine whether to include it. Every time an interaction is added, the R^2 and adjusted R^2 of the enhanced models are recomputed until no more interactions are above-threshold.

This process of adding parameters is applied repeatedly until either all parameters are included in the model or no remaining parameter can improve the model quality by more than the threshold θ. At this point, the regression model is returned as the result of this algorithm.

5.3. Use of Cubic Splines

Our regression-based performance estimation method develops parameterized performance estimation equations, as given in Equation 2, based on natural cubic splines. Cubic splines are flexible enough to express common functions encountered in practical problems, and natural cubic splines have improved behavior at the edges of the explored design space [20].

A cubic spline needs two arguments: the number of knots and the knot positions [27]. Using more knots strengthens the spline's ability to approximate complex functions, but also increases the risk of overfitting. We found that 1 to 3 knots work well for our studied value ranges. For knot positions, one common practice is to place knots at quartiles of the observed data. In our experiments, we make knots evenly spaced between the minimum and maximum value of any parameter. Because our design points are randomly and uniformly sampled, our approach is asymptotically equivalent to the fixed quartile approach when the number of samples is large enough (roughly 20). For very small sample sizes, our approach is more numerically stable because it prevents two knots from being placed at the same location.

5.4. Variations of the Stepwise Method

Users can customize our regression method in multiple ways to make it better suit their applications. Two important variations of the method are the metric by which additional terms are chosen and the θ and ϕ threshold values. We used adjusted R^2 as our metric to measure the quality of proposed augmented models. Adjusted R^2 is simple to compute and understand, and it works well in our studies. AIC and t-tests are two other commonly used metrics [14, 25]. AIC strikes a balance between model complexity and model fit, while the t-test uses null hypothesis to test additional terms. The choice of metrics is decoupled from the Stargazer framework itself, so users can choose metrics specific to their applications while keeping the framework unchanged.

Mathematically, θ and ϕ could always be zero because any non-zero adjusted R^2 suggests that the additional term is helpful and should be included in the model. However, in reality, we often want to exclude terms that are only marginally useful, to limit the final model's complexity. We can end the regression process when the number of terms has reached a pre-specified value, but it is difficult to pick one such value for all programs because complex programs usually intrinsically need more terms than simpler programs. Setting a target R^2 value to reach has a similar problem, because programs have varying final R^2 values based on how amenable they are to regression modeling. Our solution to this problem is to set θ and ϕ to a small nonzero value (e.g. 0.01 is experimentally found to be a reasonable choice) such that the algorithm terminates when the R^2 value converges to its final value. At that point, either no terms remain, or none of the remaining terms can bring significant improvements to the model.

5.5. Example of Stepwise Regression: `matMul`

As an example to help readers understand how our algorithm works, the regression method is applied to the matrix multiply sample program mentioned in Section 4.3 when the entire design space of Section 6.1 is evaluated. The intermediate and final results are presented in Table 1, which demonstrates how each parameter or interaction term is picked or discarded as the regression process proceeds through an actual use of the algorithm. For brevity, we used $\theta = 0.003$ and $\phi = 0$ in this example to control the number of terms in the final model.

In Table 1, the regression process proceeds from top to bottom. Bold numbers and terms indicate additions to the model. A pairwise interaction term between two factors a and b is shown as a`:`b. On the first row, we compute R^2 of models with single parameters. Because `#blk` has the highest R^2 (0.719), it is chosen as the first parameter. We then compute adjusted R^2 of the models that combine `#blk` with each remaining parameter. Because `simd` shows the largest adjusted R^2 (0.851), it is chosen as the next parameter. The R^2 of the current model is recalculated (0.853). At this point, we evaluate adding the interaction of `simd` with each parameter already existing in the model; in this case, that is only the interaction `simd:#blk`. Since the interaction term's adjusted R^2 (0.967) increases more than ϕ compared to R^2 of the current model, that term is included and R^2 of the current model is recalculated (0.968). The process continues until no more terms can bring an increase of more than θ to the R^2 of the model. The final model (last row in Table 1) includes 5 splines, 3 of which are basic terms (`#blk`, `simd`, and `intra`), and the other 2 are pairwise interactions (`simd:#blk` and `intra:#blk`).

	#blk	c$	t$	smp	ccp	simd	mshr	dramq	intra	inter
R^2	**0.719**	0.023	0.004	0.004	0.004	0.154	0.001	0.009	0.014	0.002
	Current model includes: **#blk**, $R^2 = 0.719$									
Adjusted R^2		0.717	0.718	0.719	0.716	**0.851**	0.716	0.719	0.718	0.718
	Current model includes: #blk, **simd**, $R^2 = 0.853$									
Adjusted R^2 of simd:	**0.967**									
	Current model includes: #blk, simd, **simd:#blk**, $R^2 = 0.968$									
Adjusted R^2		0.967	0.967	0.971	0.967		0.967	0.969	**0.972**	0.967
	Current model includes: #blk, simd, simd:#blk, **intra**, $R^2 = 0.9727$									
Adjusted R^2 of intra:	**0.9730**					0.9728				
	Current model includes: #blk, simd, simd:#blk, intra, **intra:#blk**, $R^2 = 0.9745$									
Adjusted R^2 of intra:						0.9737				
	Current model includes: #blk, simd, simd:#blk, intra, intra:#blk, $R^2 = 0.9745$									
Adjusted R^2		0.9729	0.9728	0.9769	0.9729		0.9728	0.9748		0.9729
	Final model includes: #blk, simd, simd:#blk, intra, intra:#blk, $R^2 = 0.9745$									

Table 1: Regression analysis of the matrix multiply program

6. Experimental Methodology

6.1. Simulator and Design Space Parameters

A cycle-level GPU simulator, GPGPU-Sim [1], is chosen for the experiments. However, because our regression method does not depend on specific features of the simulator, Stargazer is applicable to any chosen GPU simulator or measurement approach. As our studied design space, Table 2 lists 10 architectural parameters and their value ranges, for a total of 933,120 possible points. Parameters that are held unchanged in the experiments are listed in Table 3, taken from an NVIDIA Quadro FX 5800 GPU. It is worth pointing out that each of the 933,120 points is one unique GPU design, so our experiments have covered a very large number of GPUs, most of which differ drastically from the baseline configuration.

The #blk parameter is the block concurrency, i.e. the number of thread blocks concurrently running on one core. We include it in our experiments because when a GPU program is given, #blk is limited only by a few on-chip resources, and hence it is a parameter reflecting the combined effect of these architectural resources. In GPGPU-Sim, #blk is determined by the smallest of four per-core resources: the shared memory size, the register file size, the maximum number of concurrent threads, and the maximum number of concurrent thread blocks. To achieve a given #blk for a design point, we choose values for these four parameters accordingly. We are able to combine these four resources into block concurrency and simplify the regression model because in this paper, we are optimizing application runtime, which does not distinguish GPU design points that result in the same block concurrency for a given application. In future studies, when other cost models such as power and area are adopted, we may need to use the original resources instead of block concurrency. In GPGPU-Sim, both shared memory and constant caches are multi-ported, and hence we also vary the number of ports of these memories. More ports generally result in fewer stalled cycles when SIMD units access memory content with bank conflicts. Finally, intra and inter represent the simulated GPU's capabilities to perform intra- and inter-warp memory request coalescing. The values indicate how many requests may be coalesced into one request.

6.2. Benchmarks

Programs from both the GPGPU-Sim benchmark suite [1] and the Rodinia benchmark suite [3] are used for the experiments. Due to the lengthy simulation time on a cycle-level simulator, we only pick programs that can finish in a reasonable amount of time on GPGPU-Sim (Table 4). We use 300 samples from each application to build the regression model and use a different set of 200 samples to verify the accuracy of the model. When a program has multiple kernel functions (such as backprop, bfs, and nw), we total the runtime of all kernel executions. Note that each of the two benchmark suites has a distinct breadth-first search program. We use capitalization to distinguish them (i.e. BFS versus bfs).

7. Accuracy of the Stepwise Regression Method

7.1. Evaluating Regression Steps by Model R^2

R^2 reflects how well a regression model explains the variation in the dependent variable. As such, it is a useful metric for measuring the quality of the models we create. As each term is added to the model, one can recompute R^2 to discern improvements in quality of the model. For all the studied benchmarks, Figure 3 plots the changes in R^2 for each added parameter, representing the contributions to the model quality from each factor. In most applications, SIMD width is the dominant factor, contributing on average 0.78 to R^2. This is consistent with the fact that GPU applications are usually computationally intensive. Thus the amount of on-chip computing capability (i.e. SIMD width) is likely to be crucial for most GPU programs.

While SIMD width is often dominant, this is not always the case, and our regression approach is effective at finding other scenarios. For example, consider nw and matMul. nw is included in the Rodinia benchmark suite to represent programs which cannot fully utilize the wide SIMD pipelines of GPUs [3]; each thread block of nw has only 16 threads. Stargazer correctly highlights this program property and determines that nw is affected more by the number of shared memory ports; most other parameters are irrelevant. For matMul, the implementation we use has a small thread block size (8×8 threads), and consequently the program is severely memory-bound when block concurrency is low. Thus, simd only shows

Param	Unit	Values	#Points	Comments
#blk	blocks/core	1, 2, 4, 8, 16	5	Block concurrency
c$	KB	1, 2, 4, 8, 16, 32	6	Constant cache size
t$	KB	1, 2, 4, 8, 16, 32	6	Texture cache size
smp	count	1, 2, 4, 8	4	# shared memory ports
ccp	count	1, 2, 4, 8	4	# constant cache ports
simd	count	8, 16, 32	3	SIMD width
mshr	count/thread	1, 2, 3	3	# MSHRs
dramq	count	16, 32, 64	3	DRAM scheduler queue size
intra	count	1, 2, 4, 8	4	Intra-warp memory coalesce
inter	count	2, 4, 6	3	Inter-warp memory coalesce
Total			933,120	

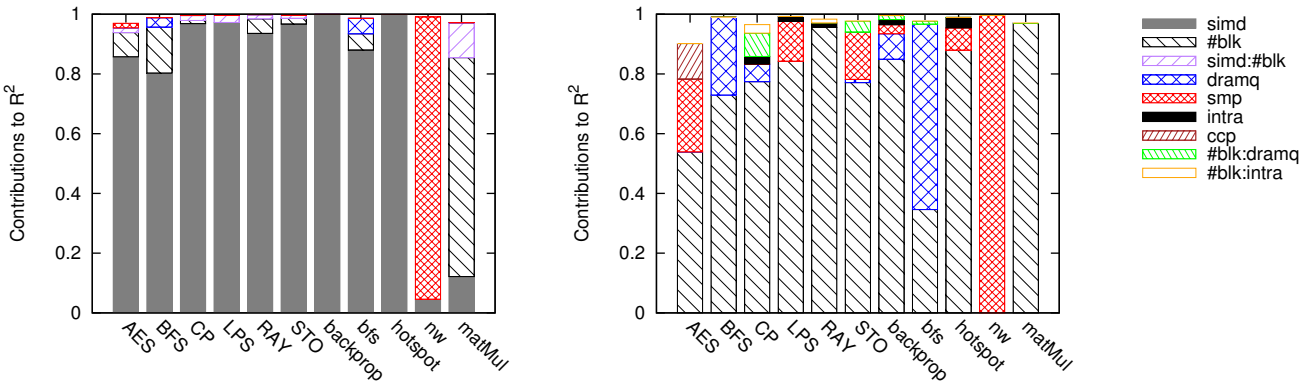
Table 2: Variable parameter values of the studied design space

Parameters	Value
# Cores	30
Core clock frequency	325 MHz
L1 data cache	None
L2 data cache	None
Interconnect topology	butterfly
Interconnect flit size	32 bytes
# DRAM controllers	8
# DRAM chips per controller	2
DRAM clock frequency	800 MHz
DRAM type	GDDR3

Table 3: Unchanged parameters

Suite	Applications	Problem Size	Threads per Block	Inst Count	Description
GPGPU-Sim	AES	256KB image	256	28M	128-bit AES encryption algorithm
	BFS	65536 nodes	512	17M	Breadth-first search on a graph
	CP	256×256 grid	128	126M	Calculate Coulombic potential in molecular dynamics
	LPS	100×100×100 grid	128	82M	3D Laplace equation solver
	RAY	256x256 image	128	71M	Graphics rendering of lighting effects
	STO	192KB input file	128	134M	Sliding-window-based MD5 calculation
Rodinia	backprop	65536 nodes	512	193M	Training weights in a layered neural network
	bfs	65536 nodes	512	28M	Breadth-first search on a graph
	hotspot	500×500 points	256	80M	Processor thermal simulation on a 2D grid
	nw	256×256 points	16	3.4M	Parallel Needleman-Wunsch algorithm for DNA sequencing
Example	matMul	256×256 matrices	64	78M	Matrix multiply sample in CUDA SDK

Table 4: Benchmark programs and their characteristics



(a) simd is the dominant factor.

(b) Zoom in on other factors when fixing simd to 32.

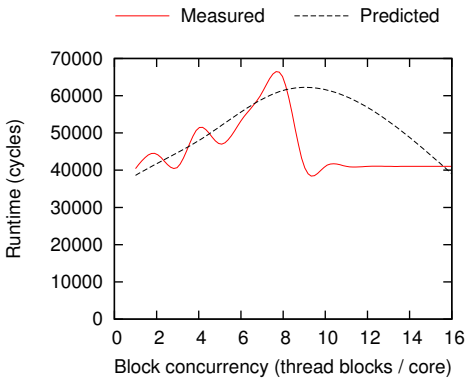
Figure 3: Each factor's contribution to R^2 of the model as regression proceeds

up as the second most important factor.

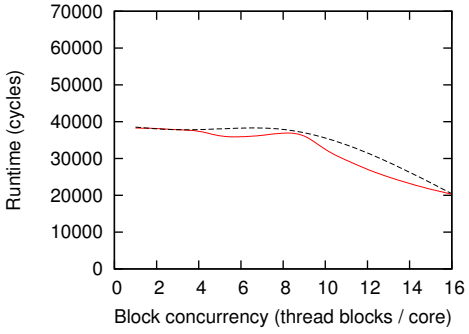
Besides SIMD width, block concurrency is often the next important factor for programs. In addition to matMul, BFS is highly dependent on #blk. Like matMul, it has high memory bandwidth demands [1]. Hence it needs more thread blocks to hide global memory load latency.

To further examine how application runtime depends on other parameters, we take a subset of the simulation data with SIMD width fixed at 32, and apply the same regression method. Figure 3b shows that different applications have diverse additional performance factors. Although block concurrency is important to most programs (contributing 0.696 to R^2 on average), AES and nw strongly depend on smp (0.244 and 0.996 respectively) while BFS and bfs are affected by dramq (0.261 and 0.620 respectively) due to high global memory traffic.

Finally, we note that the interaction factors identified by our method in Figure 3 all involve #blk. This is because the specific resource demands of each application affect runtime more prominently when block concurrency is low, hence the strong interactions between #blk and these parameters. Furthermore, examining these interactions helps reveal the particular resources needed by each application (such as intra for CP and dramq for a few benchmarks) when block concurrency cannot hide global memory latency well. In summary, our automated regression method helps build useful intuition about important parameters and their interactions.



(a) The original AES runtime



(b) The adjusted AES runtime

Figure 4: AES's runtime dependence on block concurrency before and after simulation tail handling is applied

7.2. Handling Short Simulation Runs

Although over all applications the average R^2 is 0.976, Figure 3 shows that AES's R^2 is the lowest. To investigate the reason, we did a separate set of experiments and found that this lack of fit is mainly due to the smaller input data sets leading to shorter simulation runs of this program.

Figure 4 presents the simulation results of AES, with #blk varying from 1 to 16. (Other parameters are fixed at the default values.) The solid line in Figure 4a plots the original simulated runtime of the AES program. The runtime increases in a non-monotonic fashion from 1 to 8, before dropping down and leveling off. This is contrary to the intuition that increasing block concurrency should decrease runtime. This behavior is due to the work scheduling algorithm used in GPGPU-Sim, and has been detailed in prior work [1]. In short, at the end of each simulation run, with little work left to process, only a fraction of cores are still doing computation, but the reported simulation time is the point at which all cores finish their work. The small data sets often used for simulation runs are particularly vulnerable to this "tail effect" because so little steady-state execution time exists before the kernel ends. Furthermore, the work distribution at the end of simulation runs is also highly sensitive to changes in block concurrency, which explains the observed runtime curve.

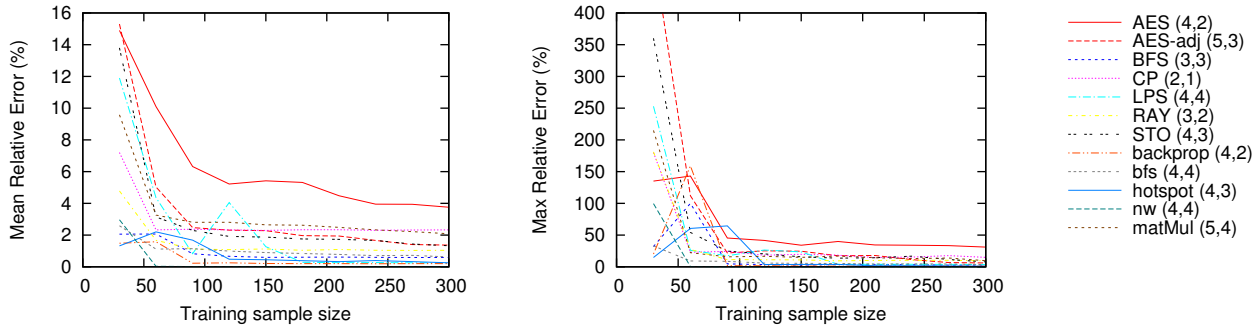
The dashed line in Figure 4a shows our regression model's prediction of the runtime for this case. One can see a large error near where #blk is 8. Our approach is useful in highlighting when such effects are occurring, but we can also reduce these effects using the technique below.

To reduce the work scheduling "tail effect"—particularly on small data sets—we process AES's simulation data to "trim the tail". Essentially, instead of using the full simulation time until the last of the work is finished, we weight the tail time by the number of cores that the tail work actually employs. The bottom graph in Figure 4 shows the result of this adjustment. The solid line indicates simulation results adjusted in this way, and the dashed line represents our regression model's predictions when it is given the adjusted input training data. With this adjustment, the runtime curve is nearly level from #blk values 1 to 8, before the tail-adjusted runtime decreases for #blk beyond 8. After the adjustment, the regression model much more closely reflects the measured data. As a result, the R^2 of AES increases from 0.902 to 0.999. For the remainder of the paper, we will refer to the adjusted AES data as AES-adj and report its results separately where relevant. The lesson learned is that accuracy metrics such as R^2 can be effective signals of issues like this. Meanwhile, even with the relatively short simulation runs of some current GPU benchmarks on GPGPU-Sim, in general our regression method still models program performance accurately, including for unprocessed AES (Section 7.3). Our method is adaptive enough to handle both short and long simulation runs.

7.3. Relative Runtime Prediction Error

A regression model estimates and interpolates unmeasured data points based on the collected sample set. Statistically, R^2 represents how well a regression model fits the observed data, but it does not directly report how accurate the predictions of unseen samples will likely be. Thus, for some users, relative prediction error may be a more intuitive accuracy metric. We evaluate our method with respect to this metric by answering the following questions: 1) How well does the runtime estimated by our model equation match the actual runtime given by simulation or measurements? 2) How many simulation samples are needed to properly train the model?

To analyze the relative error of predictions made by the regression model, we use the following approach. For each application, its simulation data repository has 500 randomly chosen design points and their simulation results. We first establish the model by randomly choosing N samples ($N = 30, 60, 90, ..., 300$) from the repository and running stepwise regression to generate a model equation. We then use the resulting model equation to predict the runtime of 200 design points also randomly chosen from the repository. These 200 samples are guaranteed to be different from those used for building the model. Estimated runtimes for these 200 test points are compared with measured simulated runtimes for the same points and relative errors are calculated. The sample-train-test process is repeated 5 times for each application, though the same 500-sample repository is used. The mean error we collect is the mean across all runs. The maximum error we report is the true maximum, not a maximum after averaging.



(a) (b)

Figure 5: Prediction accuracy vs. training sample size. Numbers in the parentheses indicate the number of splines used for basic terms and interaction terms in the 300-sample models.

Figure 5 plots the runtime prediction errors our model experienced for each of the applications in our benchmark suite. The left graph shows the mean error, and the right graph shows the maximum observed error for the model across the 200×5 test points. The x-axis varies from 30 to 300 and represents the number of randomly-chosen simulation points in our training sample set.

We start by considering the accuracies achieved with 300 sample points. Recall that a training set of 300 points is already a considerable reduction from the original 933 K points in the exhaustive design space. For this approach with 300 random training samples, the average prediction error of our method on the 200 unseen samples is remarkable: below 3.8%. `AES` has the highest mean relative error of 3.8%; its 98th percentile error is 21.7% and the maximum error is 31.1%. (However, `AES-adj` is much lower due to the tail adjustment.) Other benchmarks generally show mean errors below 2.3%, 75th percentile errors below 4.4%, and maximum errors below 15%.

Prediction accuracy depends heavily on program characteristics. Programs that exhibit more complex behavior and use more features of the GPU, such as `AES`, are inherently more difficult to predict than programs that rely on fewer architectural parameters, such as `nw`. In addition to these intrinsic structural differences, we note that the relatively short simulation times affect prediction accuracy adversely and significantly in certain benchmark programs including `AES` and `matMul`.

Having established that 300 sample points offer excellent accuracy for our application suite, the remaining question is how few samples the regression method should use to still achieve acceptable accuracy. Figure 5 shows how average error (5a) and maximum error (5b) are affected by the number of simulations used to train the model. For all applications except for the original `AES`, as few as 30–60 sample points are sufficient for mean accuracy of 5% or better. While maximum errors spike upward for small sample sets, it quickly drops to acceptable levels at 100 sample points and beyond.

To some degree, the question of how many sample points is sufficient is related to the questions of how complex the application is and how many terms the stepwise regression model must include. For this reason, each application is labeled in the legend of Figure 5 with an annotation of the form $(param, pair)$. In this annotation, $param$ is the number of splines used for individual parameter terms in each appliction's regression model and $pair$ is the number of additional terms used to reflect pairwise interactions between parameters. The total number of terms in the regression equation is equal to the sum of these two. We report these $(param, pair)$ results for the maximum accuracy case (300 sample points) with θ and ϕ set to 0. (The earlier matrix multiply example in Table 1 used a larger θ value which leads to fewer terms to simplify the presentation.)

Applications such as `CP` have very few included terms in the final regression model. For these, a small number of randomly-sampled designs nicely cover possible parameter value combinations of interest. As a result, the error is quite low even for very small sample counts. On the other hand, applications such as `AES` depend on more parameters and therefore require substantially more samples to cover enough area of the design space the parameters span. The other benchmarks lie between these two extremes. One can use trends from Figure 5 to obtain a trade-off between simulation time and model accuracy. Section 8.1 explains this trade-off further.

7.4. Comparison Against a Fixed-Factor Method

Finally, we compare our method's accuracy against simpler methods that use a fixed number of factors. These methods might be similar to the intuition and educated guesses that experienced designers use in more familiar design spaces. The simpler, fixed-factor method uses the top three factors observed in most of the program regression results in Section 7.1: `simd`, `#blk`, and `simd:#blk`. Figure 6 shows the average relative error comparison between a fixed-factor approach and our automated approach. The figure shows that the automated method is never worse than the fixed-factor method. This is because the former usually includes more terms, except for programs (such as `CP`) which have no more than three terms in their regression results. In these cases the automated method is at least as good as the fixed-factor method.

Note that we are giving an advantage to the fixed-factor

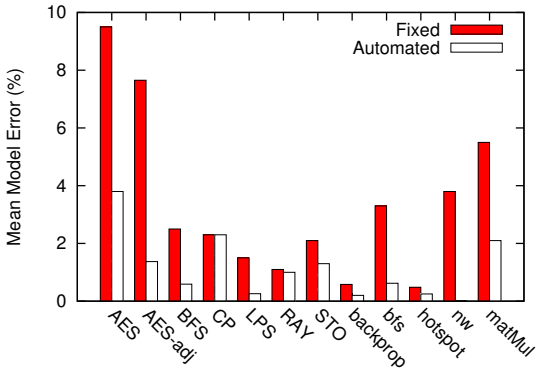


Figure 6: Automated method vs. fixed 3-factor method

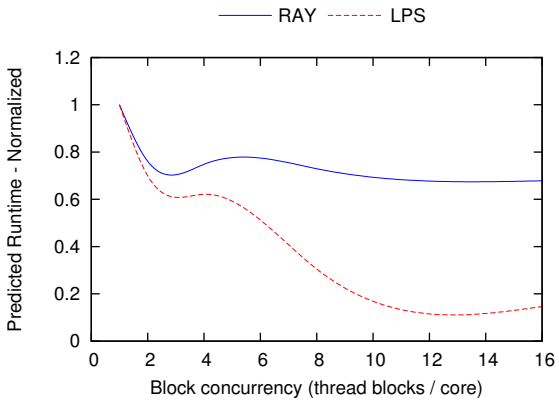


Figure 7: `LPS`'s runtime is more sensitive to block concurrency than `RAY`'s runtime, and thus `LPS` is more memory-bound than `RAY`.

method because we use the *top* three factors in Figure 6. This essentially assumes the hardware designer always knows what the top factors are. In the case of more complex design spaces or less experienced designers, an automated approach is more likely to lead to effective and accurate design space pruning.

8. Case Studies of Using Stargazer

8.1. Reducing Simulation Time by Pruning Design Space

The primary usage scenario of Stargazer is to reduce simulation requirements while also guiding architects towards accurate and informed pruning of the GPU design space. For example, assume a GPU architect is tasked with exploring the design space in Table 2. Exhaustively simulating nearly 1 million points in the design space is undoubtedly impractical. Instead, the architect can use our tool to rapidly capture the general structure of the design space by sampling and simulating a much smaller number of designs. In particular, Figure 5 shows the trade-off between accuracy and training time (simulation samples). If the architect chooses an expected mean error of 5% or less, all but one of our applications require only 30 or 60 design points to be simulated. (`AES` is the outlier, though `AES` with tail reduction has much lower errors.) This is a significant reduction (roughly 15000×) in potential simulation requirements. Furthermore, because there is no learning/feedback loop that guides sample selection in our method, simulated points can be chosen *at random* from the space of possibilities, and the simulations can all be run in parallel. Once the regression model is obtained, one can evaluate other design points using just lookups into the regression equation.

Most researchers do not exhaustively simulate all possible points, but instead simulate explorations along "important axes" while centering their evaluations around a likely design point. Selecting a design's "center point" and "important" axes, however, requires intuition and often some amount of trial-and-error. Our approach improves the automation by which such architectural traits can be determined.

If equation-based evaluations of unsimulated points is insufficient, one could still use the regression model to identify the important axes and "interesting" parts of the design space. After that, further simulations can be targeted at the regions highlighted by the regression model to give the most insight and accuracy from a limited number of simulation cycles.

8.2. Application Characteristics and Benchmark Diversity

Designers or users of a benchmark suite often want to assess the diversity of the included benchmark programs [2,8,9]. One way to characterize benchmark diversity is to compare regression models obtained from each application across the whole suite. Figure 3 shows how each application's performance is affected by various GPU architectural parameters when SIMD width is fixed. The unique characteristics of programs like `nw` are immediately noticeable. Benchmark designers often include a few such programs to enhance benchmark diversity. In addition, both breadth-first search programs heavily depend on the `dramq` factor, suggesting that these two programs have high global memory needs. Meanwhile, `AES`, `LPS`, `STO`, and `hotspot` all have noticeable dependence on `smp`, while `AES` additionally relies on `ccp`. These components are consistent with each application's known behavior. In summary, regression methods make a benchmark suite's attributes easier to compare.

As another example, consider how GPU programs rely on high block concurrency to hide memory latency [21]. By analyzing the slope of an application's runtime versus `#blk` curve, one can detect how severely an application is limited by global memory bandwidth. A compute-bound application needs fewer thread blocks to achieve peak performance, while a memory-bound application needs more. Stargazer can estimate the runtime versus `#blk` curve for this use.

Using the regression models of `LPS` and `RAY` as examples, we vary `#blk` from 1 to 16 and keep the remaining parameters fixed. This curve, shown in Figure 7, is essentially the spline generated for the `#blk` term of the model. The graph shows that `LPS`'s runtime depends more on block concurrency because `LPS` does very little computation while streaming the 3-layer cube over the global memory. `RAY`'s runtime depends less on block concurrency because it does heavy computation to calculate light reflection and shadows. The model clearly highlights `LPS` is more memory-bound than `RAY`.

9. Conclusions

This paper has proposed and evaluated an automated stepwise regression method for design space exploration in GPUs. Relative to prior work in CPU design space pruning, our work offers useful novelties in terms of handling a large number of variables and interactions. This allows us to automatically and efficiently explore the complex characteristics of the GPU design space. Relative to prior work in GPU performance evaluation, this work offers experiences from our regression method that show how it could be used to understand benchmark diversity, hardware bottlenecks and trade-offs, and other important scenarios. Extremely sparse samples of the design space (300 out of 933K points) can offer performance estimation equations with very good accuracy (1.1% average error). This can lead to 15000× reduction in simulation time requirements relative to exhaustive approaches and also improves significantly compared to more tailored approaches involving parameter variations around a possible design point.

As GPU designs incorporate new and diverse features, it will be important for hardware and software designers to quickly develop intuition regarding how they influence performance. Considerable challenges lie in anticipating inflection points and parameter interactions in complex designs. We see our automated regression method as an important step in facilitating such design space exploration.

Acknowledgment

We thank Daniel Lustig, Carole-Jean Wu and the anonymous reviewers for their useful feedback and insights related to this work. This material is based upon work supported in part by the National Science Foundation under Grant No. CCF-0916971. The authors also acknowledge the support of the Gigascale Systems Research Center, one of six centers funded under the Focus Center Research Program (FCRP), a Semiconductor Research Corporation entity. Finally, we thank NVIDIA for donating some of the equipment used in this research. We are making Stargazer regression models and tools openly available; please contact the authors for access.

References

[1] A. Bakhoda *et al.*, "Analyzing CUDA workloads using a detailed GPU simulator," in *IEEE Int. Symp. Performance Analysis of Systems and Software*, 2009.

[2] C. Bienia *et al.*, "PARSEC vs. SPLASH-2: A quantitative comparison of two multithreaded benchmark suites on Chip-Multiprocessors," in *IEEE Int. Symp. Workload Characterization*, 2008.

[3] S. Che *et al.*, "Rodinia: A benchmark suite for heterogeneous computing," in *IEEE Int. Symp. Workload Characterization*, 2009.

[4] J. W. Choi *et al.*, "Model-driven autotuning of sparse matrix-vector multiply on GPUs," in *Proc. 15th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming*, 2010.

[5] W. W. L. Fung *et al.*, "Dynamic warp formation and scheduling for efficient GPU control flow," in *40th Ann. IEEE/ACM Int. Symp. Microarchitecture*, 2007.

[6] S. Hong and H. Kim, "An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness," in *Proc. 36th Ann. Int. Symp. Computer Architecture*, 2009.

[7] S. Hong and H. Kim, "An integrated GPU power and performance model," in *Proc. 37th Ann. Int. Symp. Computer Architecture*, 2010.

[8] K. Hoste and L. Eeckhout, "Microarchitecture-independent workload characterization," *IEEE Micro*, vol. 27, no. 3, pp. 63–72, 2007.

[9] K. Hoste *et al.*, "Performance prediction based on inherent program similarity," in *Proc. 15th Int. Conf. Parallel Architectures and Compilation Techniques*, 2006.

[10] E. Ipek *et al.*, "Efficiently exploring architectural design spaces via predictive modeling," in *Proc. 12th Int. Conf. Architectural Support for Programming Languages and Operating Systems*, 2006.

[11] P. J. Joseph *et al.*, "Construction and use of linear regression models for processor performance analysis," in *IEEE 12th Int. Symp. High Performance Computer Architecture*, 2006.

[12] P. J. Joseph *et al.*, "A predictive performance model for superscalar processors," in *39th Ann. IEEE/ACM Int. Symp. Microarchitecture*, 2006.

[13] "OpenCL: The open standard for parallel programming of heterogeneous systems," The Khronos Group. [Online]. Available: http://www.khronos.org/opencl/

[14] M. Kutner *et al.*, *Applied Linear Regression Models*, 4th ed. McGraw-Hill/Irwin, 2004.

[15] B. C. Lee and D. M. Brooks, "Accurate and efficient regression modeling for microarchitectural performance and power prediction," in *Proc. 12th Int. Conf. Architectural Support for Programming Languages and Operating Systems*, 2006.

[16] B. C. Lee and D. M. Brooks, "Illustrative design space studies with microarchitectural regression models," in *IEEE 13th Int. Symp. High Performance Computer Architecture*, 2007.

[17] B. C. Lee and D. M. Brooks, "Applied inference: Case studies in microarchitectural design," *ACM Trans. Architecture and Code Optimization*, vol. 7, no. 2, pp. 8:1–8:37, Oct. 2010.

[18] V. W. Lee *et al.*, "Debunking the 100X GPU vs. CPU myth: An evaluation of throughput computing on CPU and GPU," in *Proc. 37th Ann. Int. Symp. Computer Architecture*, 2010.

[19] E. Lindholm *et al.*, "NVIDIA Tesla: A unified graphics and computing architecture," *IEEE Micro*, vol. 28, no. 2, pp. 39–55, Mar. 2008.

[20] D. C. Montgomery *et al.*, *Introduction to Linear Regression Analysis*, 3rd ed. Wiley-Interscience, 2001.

[21] *CUDA C Best Practices Guide*, 2nd ed., NVIDIA Corporation.

[22] "CUDA C/C++ SDK CODE Samples," NVIDIA Corporation. [Online]. Available: http://developer.nvidia.com/cuda-cc-sdk-code-samples/

[23] *NVIDIA CUDA Programming Guide*, 2nd ed., NVIDIA Corporation.

[24] S. Ryoo *et al.*, "Program optimization space pruning for a multithreaded GPU," in *Proc. 6th Ann. IEEE/ACM Int. Symp. Code Generation and Optimization*, 2008.

[25] Y. Sakamoto *et al.*, *Akaike Information Criterion Statistics*. D. Reidel Publishing Company, 1986.

[26] J. W. Sheaffer *et al.*, "A flexible simulation framework for graphics architectures," in *Proc. ACM SIGGRAPH/EUROGRAPHICS Conf. Graphics Hardware*, 2004.

[27] C. Stone and C. Koo, "Additive splines in statistics," in *Proc. of the Statistical Computer Section*, 1986, pp. 45–48.

[28] H. Wong *et al.*, "Demystifying GPU microarchitecture through microbenchmarking," in *IEEE Int. Symp. Performance Analysis of Systems and Software*, 2010.