Starchart: Hardware and Software Optimization Using Recursive Partitioning Regression Trees

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 in increasingly wide use, but significant hurdles lie in selecting the appropriate algorithms, runtime parameter settings, and hardware configurations to achieve power and performance goals with them. Exploring hardware and software choices requires time-consuming simulations or extensive real-system measurements. While some auto-tuning support has been proposed, it is often narrow in scope and heuristic in operation.**

This paper proposes and evaluates a statistical analysis technique, Starchart, that partitions the GPU hardware/software tuning space by automatically discerning important inflection points in design parameter values. Unlike prior methods, Starchart can identify the best parameter choices within different regions of the space. Our tool is efficient—evaluating at most 0.3% of the tuning space, and often much less—and is robust enough to analyze highly variable real-system measurements, not just simulation. In one case study, we use it to automatically find platform-specific parameter settings that are $6.3\times$ faster (for AMD) and $1.3\times$ faster (for NVIDIA) than a single general setting. We also show how power-optimized parameter settings can save $47\,\mathrm{W}$ (26% of total GPU power) with little performance loss. Overall, Starchart can serve as a foundation for a range of GPU compiler optimizations, auto-tuners, and programmer tools. Furthermore, because Starchart does not rely on specific GPU features, we expect it to be useful for broader CPU/GPU studies as well.

***Keywords*—GPU, auto-tuning, design space exploration, regression tree, decision tree.**

I. Introduction

Modern processors employ increasing amounts of heterogeneity and specialization to reach their power-performance targets. With CPU and GPU cores available, along with accelerators and complex, configurable memory hierarchies, the multitude of resource combinations and subtle software-hardware interactions create challenges to those optimizing either software performance on a given platform or hardware design choices for a future one. Traditional manual exploration techniques such as exhaustive search or center design point approaches are either not scalable or are prone to missing important parts of the increasingly large design spaces.

As one example, in recent computing systems, CPU-GPU pairs have emerged as a popular form of heterogeneous parallelism. While GPUs offer extremely high performance at good performance-per-watt, vexing challenges exist in mapping applications onto a GPU and in finding the best configurations and parameter choices. Unfortunately, little support exists for GPU performance or power tuning. Extensive programmer intuition and hours if not days of simulation or experimentation are often required, given the large number of possible hardware-software combinations and interactions.

For GPU applications, poor configuration choices have severe consequences. For example, the choice between optimal and off-optimal parameter settings for the `swap` kernel in the Rodinia `kmeans` benchmark changes performance by $100\times$ depending on a single parameter value (Section II). In another case, judicious choice of parameter settings can save application power usage by nearly $50\,\mathrm{W}$ with almost no performance loss (Figure 10c).

Clearly, finding the "right" parameter settings can be very beneficial; the key is to do this *design space exploration* efficiently. While some linear regression-based automatic design space exploration methods have been proposed for simulation-based experiments [11, 12, 14], a major issue is that they evaluate parameters based on their importance *globally* across the entire design space, rather than *locally* within particular subregions. This reduces both their accuracy and their applicability in real-system environments (Section VI-B).

This paper proposes and evaluates *Starchart*[1], a fully-automated design space optimizer using a statistical tree-based partitioning approach. It analyzes parameter influences in local subspaces in addition to across the entire parameter space. The subspace analysis capability allows Starchart to accurately model complex design spaces and thus expands its usage scenarios. We have applied our approach to several metrics including power and performance. Starchart is robust enough to be used with real-system measurements, despite their higher variability than simulations. For design spaces with thousands to millions of possible designs, much less than 0.3% of the points need to be sampled, resulting in more than $300\times$ productivity improvement. We present results from real-system measurements on two distinct GPU platforms (NVIDIA Tesla C2070 and AMD Radeon HD 7970) and via multiple metrics (performance, power, and combinations of the two). This paper makes the following contributions.

First, Starchart's novel partitioned tree approach can provide accurate performance or power estimates for complex hardware or software design spaces, even with the variations of real-system measurements. It only uses the original application design parameter values, without the need for auxiliary variables such as performance counter measurements.

[1]Starchart stands for Statistical Tuning via Automatically- and Recursively-Constructed, Hierarchically-Applied Regression Trees, a follow-up tool to our prior work Stargazer [11].

Second, we present case studies that offer useful GPU insights as well as demonstrate the value of subspace-based exploration. For GPU designers and users, Starchart enables cross-platform and cross-input program optimizations, improving performance by up to $6.3\times$. Starchart can also be used to do power-guided performance optimizations; for our benchmarks, it identifies a parameter setting that saves up to 47 W with little performance loss. Our examples show how Starchart can be used to identify influential design parameters near a particular power budget, to select between different hardware platforms (e.g. NVIDIA vs. AMD), and to estimate performance across a range of input data sizes.

Third, even though this paper focuses on GPU scenarios, Starchart does not rely on any GPU-specific features. Thus, it is applicable to a broader range of architectures including CPU-only and CPU-GPU systems.

The rest of the paper is organized as follows. Section II uses an example to motivate our tool. Section III introduces prior related work. The main algorithm of Starchart is presented in Section IV. Section V describes our experimental methodology, and Section VI evaluates the accuracy of Starchart. A number of case studies of using Starchart are presented in Section VII. Finally, Section VIII concludes the paper.

II. Motivation: Design Space Partitioning

To motivate our work, we present a walkthrough example based on the `swap` kernel in the `kmeans` program from the Rodinia benchmark suite, using data collected via real-system measurements on the AMD platform described in Section V.

Figure 1a shows the `swap` kernel from `kmeans`, but with additional parameterizations added to allow possible optimizations. (This code template approach is frequently used by auto-tuning optimizers.) The code is a matrix transpose operation from an array F of M multidimensional points, each having N features, into an array Ft of N lists, each having M features. Key optimization parameters include: (i) `tpp`, the number of parallel threads per point; (ii) `ppb`, the number of points per block; and (iii) `consec`, which is a binary flag controlling thread organization and memory access striding. When `consec` is 0, features accessed by a single thread are numbered consecutively, but from the point of view of the whole thread block, they are issuing strided accesses. When `consec` is 1, features accessed by a single thread are in strides, but from the point of view of the whole thread block, all threads are issuing consecutive and coalesced accesses. Although there are only three parameters discussed in this example, selecting parameter values is difficult and subtle. Selecting the wrong parameter values can lead to $100\times$ performance penalties. Selecting the correct parameter settings is difficult, however, because the parameters interact in complex ways to affect parallelism, memory bandwidth and coalescing, and—ultimately—performance. For example, `tpp` and `ppb` represent orthogonal forms of parallelism, but together their product is limited by the thread parallelism of the GPU platform. Without automation, design space exploration would either require near-exhaustive experimental runs or rely on the error-prone selection of a "center point" design.

To begin the walkthrough example, Figure 1b plots the kernel runtime for varying selections of `ppb` values, when `tpp` and `consec` are both set to 1 (the default Rodinia code). If, for some reason, the program cannot use `tpp` parallelism, then Figure 1b shows that selecting a good value of `ppb` will make a large difference. A `ppb` of 8 is a sweet spot, with performance nearly $6\times$ better than some other options. However, Figure 1c shows that `tpp` is usually a more effective form of parallelism. When `tpp` can take values higher than 1 (indicating that some threads-per-point parallelism is being exploited), the runtimes are consistently better than those in Figure 1b. Furthermore, for larger `tpp` values, runtime varies very little with increasing `ppb`, indicating `ppb` is not the preferred form of parallelism in these situations. Finally, Figure 1c also shows that while the `consec` parameter does affect performance, this is primarily for small `tpp` values.

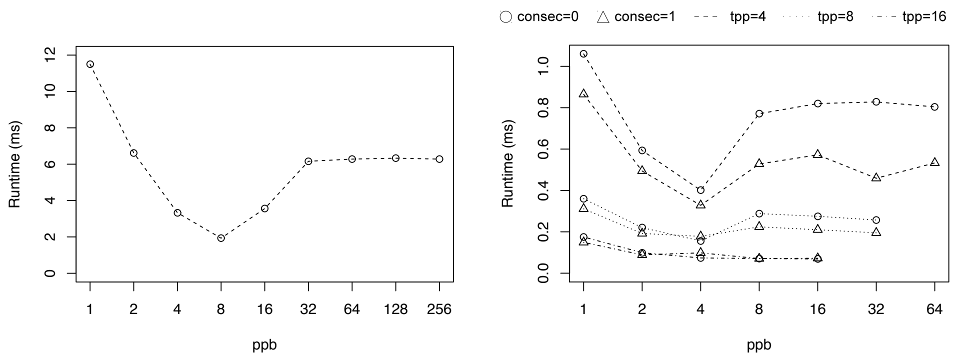
Clearly, even what seems to be a fairly simple 3-parameter tuning space displays considerable subtlety. Parallelism and memory striding parameters interact differently in different parts of the tuning space, making it a nontrivial task to determine the best parameter settings under varying program runtime environments and constraints. Our paper proposes and evaluates Starchart, a statistical technique that automatically determines the relative importance of parameters both globally across the entire design space, as well as locally to particular subspaces. Starchart's ability to explore subspaces enables a wide range of use cases (Section VII) beyond those handled well by previous regression approaches.

Figure 2 illustrates a regression tree automatically generated by our statistical regression process for the same kernel. Each branch in the tree divides the design space where a particular parameter value has been shown to be an important determinant for program performance (or any other metric). The topmost branch point indicates the most important parameter value. Each subsequent level of the tree shows the next most important parameter decision, contingent on the ones above it. Furthermore, each rectangle gives the number of experimental sample points in that particular design space partition and their average performance. Reading the automatically-generated tree, a GPU application developer or tuning tool can identify which parameters are most influential on a given metric and can see when parameters may be important within a local subregion of the tuning space. In contrast, existing approaches would not distinguish between the different `consec` values when `tpp` ≤ 8 because `tpp` > 8 is the optimal global solution and `consec` is irrelevant under that subspace. Starchart can easily capture and express arbitrarily high levels of interactions (such as the three-way interaction between `tpp`, `ppb`, and `consec`) simply by adding levels to the tree. Existing design explorers based on linear regression miss many such interactions, because they either only consider manually specified parameter interactions [14] or limit themselves to pairwise interactions [11].

III. Related Work

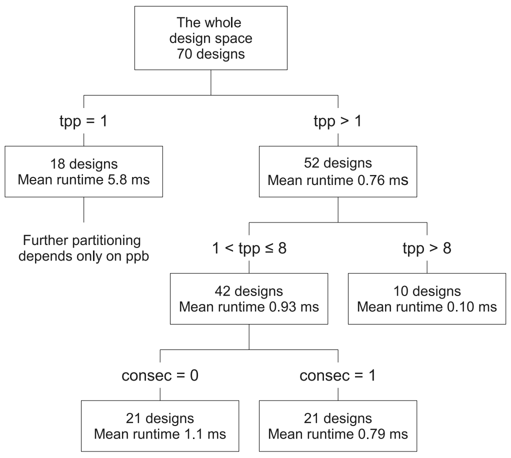
GPU auto-tuning work [5–7, 15] typically optimizes the performance of a particular application or algorithm. Considerable human intuition and experience are, however, still needed to select tuning parameters, prune their values, and build performance models, in order to find a small enough region of the design space that contains the optimal design and on which exhaustive experiments are feasible. Our work can

```
// Launch M/ppb thread blocks,
// each having tpp*ppb threads
tid = global thread ID;
pid = tid / tpp;
for (i = 0; i < N/tpp; i++) {
  if (consec > 0)
    k = tpp*i + tid%tpp;
  else
    k = N/tpp * (tid%tpp) + i;
  Ft[k*M + pid] = F[pid*N + k];
}
```



(a) Parameterized code (b) `tpp` = 1, `consec` = 1 (original kernel) (c) `ppb`–`tpp`–`consec` interactions

Fig. 1: For the `swap` kernel in `kmeans`, `ppb` is an important performance lever for the original kernel. Higher `tpp` improves performance while reducing `ppb`'s influence. `consec` also affects performance, but only for smaller `tpp` values.



The whole design space 70 designs
tpp = 1
tpp > 1
18 designs Mean runtime 5.8 ms
52 designs Mean runtime 0.76 ms
Further partitioning depends only on ppb
1 < tpp ≤ 8
tpp > 8
42 designs Mean runtime 0.93 ms
10 designs Mean runtime 0.10 ms
consec = 0
consec = 1
21 designs Mean runtime 1.1 ms
21 designs Mean runtime 0.79 ms

Fig. 2: Design space partitions represented as a tree efficiently summarize performance trends.

automate this part of creating a new auto-tuner. With Starchart, auto-tuners can simply use a large number of unpruned tuning parameters, and our method will automatically detect the important ones and their interactions, guiding users directly to the optimal design or the region that contains it.

Various statistical and machine learning methods have been applied to prune the usually vast auto-tuning or compiler optimization space of a program. Triantafyllis et al. propose a compiler framework that uses heuristics learned offline from a set of representative code segments and a compile-time performance estimator to iteratively prune program optimizations [22]. However, their goal is to assess a set of common compiler optimizations identical across all applications, while our tool aims to build performance models generated from application-specific design parameters. In a few other more focused case studies, Bergstra et al. use a timing model built with boosted regression trees to predictively auto-tune a single kernel, filterbank [2], and Ganapathi et al. tune stencil code using kernel canonical-correlation analysis [8]. One key distinction between these prior studies and our work is that their main purpose is to find the globally optimal program configuration, while Starchart not only finds the global optimum but also reveals subspace structures and local optima.

Apart from the methods mentioned above, some linear regression-based design space explorers [11, 12, 14] are worth particular attention because they adopt a similar workflow as our tool. These design space explorers offer cost-effective performance-prediction alternatives to simulation. However, like the methods mentioned above, none of the linear regression model-based approaches can distinguish a parameter's effect in local regions of a design space. This means they cannot be used to solve problems that involve comparing designs in different subspaces, which is the basis of many useful tasks (e.g. nearly all case studies in Section VII). Furthermore, these prior approaches do not handle well the variability that arises in real-system measurements. In contrast, Starchart's subspace approach and its ability to account for high-order parameter interactions make it amenable to real-system data (Section VI-B).

Other GPU performance models (such as those based on program analysis [9, 20, 21]) and power models (such as those based on performance counters [4, 17]) exist. The former impose strong limits on the types of programs that can be analyzed and require users to have intimate knowledge about the programs. The latter can be used as the power measurement component of our framework when real-system power measurements are physically infeasible.

Formally, our algorithm builds a type of decision tree [13], which is studied extensively in data mining and machine learning fields. Some current studies [16] improve on the classic method we use. They can be easily applied to our core algorithm to improve its accuracy, efficiency, or both.

IV. Automated Design Space Partitioning

This section gives the overall flow of Starchart, our statistical hardware and software tuning space partitioning approach, as well as details about its design issues.

A. Design Sample Collection

Parameter Selection: For an application or a group of applications, a set of parameters must be identified which define the dimensions of the tuning space. Parameters may relate to software (e.g. blocking factor, thread count, etc.) or to hardware (e.g. active core count, system clock rates, etc.).

Parameters can either be binary in nature (e.g. whether to turn on or off an optimization), or they can take numeric values across a possible range. Parameters can even be choices, such as including "NVIDIA vs. AMD" as a design option (Section VII-B). Our statistical partitioning is effective at automatically pruning design spaces and identifying *interesting* subspaces. As a result, one can be generous in including any tuning parameters that *may* matter.

Sampling Program Designs: In traditional auto-tuning, a designer or programmer might run evaluations for a large number of possible points within the fully-enumerated set of design possibilities. Exhaustively evaluating all design points is rare and time-consuming. The other common technique—selecting a "center point" design and evaluating a range of parameter settings around it—relies heavily on designer intuition and may miss promising regions of the design space.

In contrast to exhaustive or center-point based explorations, our approach rests only on a small set of designs sampled from the space uniformly at random (UAR). UAR sampling is easy to perform and offers good coverage of the space. Section VI-A shows that less than 0.3% of all possible design points are needed to accurately characterize the whole space, resulting in a more than 300$\times$ productivity improvement.

Evaluations: Each randomly-selected sample point represents a specific design which must be evaluated. This could be either performance or power measurements, and it could be either real-system measurements or simulation. In this paper, we use real-system power and performance measurements on NVIDIA and AMD platforms (Section V). Thus, the input to our partitioning algorithm is a set of UAR sampled design points (i.e. their parameter settings) and the resulting performance or power measurements.

B. Automated Partitioning Algorithm

Figure 3 shows the pseudocode for Starchart's design space exploration algorithm. Its partitioning algorithm seeks to determine which of the many parameters, P_i, strongly influence a metric such as performance and to determine how these different parameters interact with one another. As input, Starchart takes a list of all possible value settings, V_i, for each parameter, P_i, as well as a set of randomly generated design samples whose performance has been measured. In this section, we use performance as an example metric; other metrics such as power can be modeled similarly.

In order to develop intuition about the importance of a parameter setting, we need to analyze how performance of the samples changes with respect to values of this particular parameter. We can do this by setting a threshold value for a parameter, dividing the sample set into two groups based on each sample's value for the specified parameter relative to the threshold value, and then comparing and contrasting the groups to each other. For example, in Figure 2, we can divide the initial sample space based on whether the parameter `tpp` is equal to 1 or not, resulting in sample subsets with 18 and 52 samples respectively.

The goal for this subdivision is to determine split points where the samples in the two sets fall into distinct, mostly non-overlapping groups. If two distinct subsets exist, this implies that the setting of that parameter has a clear influence on performance. If the subsets are widely overlapping, this implies that the parameter setting does not have a clear influence on performance. Starchart uses the average values of samples as well as the sum of squared errors (SSE, the squared sum of differences between samples and the average) to compare two sample sets to each other, though other metrics could also be used. The average provides an overview for the set of samples while the SSE describes how much variance the set has.

In Figure 2, we see that the subdivision of the original set creates two subsets with widely disparate mean runtimes, 5.8 ms vs. 0.76 ms. This implies that the `tpp` setting can strongly influence performance. However, to ensure that these two numbers do a better job describing the samples than the average of the entire set, we compare the SSE of the initial samples to the combined SSE values of the two subsets. If the initial set's SSE is larger than the sum of the SSEs of the subsets, we can conclude that this parameter value is a good distinguisher of performance.

As shown in Figure 3, Starchart tentatively divides the initial sample space into two groups for every possible parameter value setting. For each chosen parameter setting, it calculates the average value and sum of the SSEs for each of the two tentative sample subsets created (lines 9-20). It then sorts all of the different sample splitting choices by the sum of the SSEs of their respective subsets. The parameter setting that results in the smallest sum of the SSEs divides the initial sample into the most distinct and internally similar subsets. That is, the choice of this parameter setting creates the greatest distinction in performance and, therefore, has the highest impact on performance. Starchart thus splits the initial set based on this parameter setting as long as the reduction in error is greater than some threshold value (lines 21-32).

After the split of the initial set into two distinct subsets, each of the two sets may still have samples that vary greatly internally. Consequently, Starchart recursively applies its algorithm to each of the two subsets which results in a total of four subsets. Further precision can be gained by continuing to apply Starchart to these four subsets and so on until the appropriate level of error is reached (lines 3-34).

Starchart's end product is a tree where nodes are design space partitions, and the edges out of them specify the values assigned to a particular parameter in each of the two subtrees. There are two key characteristics about these trees. First, parameter splits highest in the tree have the highest overall influence on performance. This is because Starchart greedily selects the parameter setting that reduces SSE the most first. Second, for each partitioned subspace, independent choices can be made regarding subsequent further partitions. This is key, because it means we can find parameters of crucial importance within a particular subregion, even if they are only of modest global importance across the full design space. The ability to find key parameters *within a subregion* is central to the efficacy of our technique and differentiates it from existing work which can only recognize globally important parameters [11, 14].

C. Algorithm Options

Tree Node Models: We use average and SSE values in each tree node in our implementation of Starchart because

```
Input: $D$: design space specified with $n$ design parameters $P_1, P_2, \ldots, P_n$
Input: $V_i$: set of values that may be taken by each parameter $P_i$
Input: Set $S$ composed of $m$ sampled designs in the space, $s_1, s_2, \ldots, s_m$,
    with their parameter values known and resulting performance, $r_1, r_2, \ldots,$
    $r_m$, measured

    List of pending partitions $C = \{S\}$
    Partitioning history $H = \{\}$
    repeat
        for all partition $c$ in $C$ do
            $\bar{r}$ = the average performance of all points $s_j$ in $c$
            $\bar{r}$ is used as the modeled performance of all points in $c$
            $r_j$ is the performance of $s_j$
            sum of squared errors $SSE_0 = \sum_j (r_j - \bar{r})^2$
            for all parameter $P_k$ do
                for all possible parameter value $v_l$ in $V_k$ do
                    split $c$ into two partitions $c_1$ and $c_2$
                    $c_1$ has the samples from $c$ with $P_k \le v_l$
                    $c_2$ has the samples from $c$ with $P_k > v_l$
                    $\bar{r_1}$ = the average performance of points in $c_1$
                    $\bar{r_2}$ = the average performance of points in $c_2$
                    $SSE_1 = \sum_j (r_j - \bar{r_1})^2$ for $s_j \in c_1$
                    $SSE_2 = \sum_j (r_j - \bar{r_2})^2$ for $s_j \in c_2$
                    $SSE_{12} = SSE_1 + SSE_2$
                end for
            end for
            remove $c$ from $C$
            find the smallest $SSE_{12}$ across all $(P_k, v_l)$ pairs
            $SSE_{min}$ = this smallest $SSE_{12}$
            $(P_{min}, v_{min})$ = the param-value pair that produces $SSE_{min}$
            $SSE_0 - SSE_{min}$ is the reduction in error
            if $SSE_0 - SSE_{min} > threshold$ then
                split $c$ into two partitions $c_l$ and $c_h$
                $c_l$ has the samples from $c$ with $P_{min} \le v_{min}$
                $c_h$ has the samples from $c$ with $P_{min} > v_{min}$
                add $c_l$ and $c_h$ to $C$
                add $((P_{min}, v_{min}), (c, c_l, c_h))$ to $H$ as part of the final tree
            end if
        end for
    until $C$ is empty

Output: Partitioning history stored in $H$ represented as a tree
```

Fig. 3: Recursively partition a program design space. The metric here is performance; power can be modeled similarly.

they are easy to compute, have clear meanings, and are widely used in traditional regression tree theories. However, different models can be used. For example, we could use linear regression models such as those in Stargazer [11] inside each tree node. Because these models are more descriptive than a simple average value, we might expect fewer tree levels before SSE stops reducing. However, these models are also much more computationally expensive to build. Exploring the trade-offs between different tree models in terms of accuracy and efficiency could be an interesting study in future work.

Stopping Criteria: In Figure 3, splitting might continue until there is only one point per set. This level of detail is rarely necessary. In general, we set the *threshold* value to 0 or a small value, meaning we stop splitting a partition when further splits do not improve the tree's capability to distinguish designs. This has a clear physical meaning: when a partition stops getting split, it means the natural variation in samples in this partition cannot be attributed to any particular parameter's choice of value. (Recall that real-systems measurements incur timing variations even in cases of repeated runs with identical parameter settings.)

In our experiments, 200 samples result in 40–50 final partitions with the above stopping criterion. Section VI-A shows this offers good model accuracy. Meanwhile, because parameters are ranked by importance top-down in the tree, users do not need to read or understand all those splits. They can focus on the top or the sub-trees of greatest interest. Other stopping criteria are also possible to further limit tree height. For example, we can put a limit on the maximum height of the tree, or we can stop splitting a partition when its data variance is below some threshold.

V. Methodology

Benchmarks Studied: As shown in Table I, we use 6 GPU kernels from the Rodinia suite [3] and NVIDIA GPU SDK [19]. The kernels exhibit diverse behavior and use various GPU functional units. For each kernel, relevant design parameters are substituted by C macros, so that our design sampler can easily modify them and generate appropriate code at compile time. Some of these parameters already exist in the kernels; others are our own addition to explore further optimizations. Every kernel is run on two platforms. The NVIDIA platform runs CUDA versions of the programs, and the AMD platform runs OpenCL versions, but they differ only syntactically. Because NVIDIA and AMD platforms have various differences (thread block size limit, shared memory size difference, etc.), the parameter values listed are the union of all possible values on both platforms. In addition, there are 3 parameter choices available only on NVIDIA (Table II). When a UAR-sampled configuration cannot be run on a particular platform, it is not counted toward its sampled design points.

Performance and Power Measurement Platforms: We experiment on two systems: (i) an NVIDIA Tesla C2070 GPU and (ii) an AMD Radeon HD 7970 GPU. Both run the Ubuntu 10.04 operating system with all unnecessary system services turned off. The AMD GPU is newer than the NVIDIA GPU, but except for Section VII-B, we use results only within the scope of each platform. In Section VII-B, one of the goals is to select the appropriate platform for a given design scenario, and having two systems with distinct capabilities is useful for comparison, despite disparate technologies.

To collect kernel execution times, vendor-supplied NVIDIA and AMD profiling tools are used [1,18]. The runtime experiments are run separately from the power experiments. Each kernel is executed at least 10 times and the average execution time taken. Only the GPU time of each kernel execution is measured and used, excluding any CPU work, data transfer, or kernel launch overhead. When the input size may change across different runs (`bfs` and `matrix` in Section VII-C and `nbody`), the runtimes are normalized against input size.

For our real-system power measurements, we use the same methodology as Hong and Kim [10]: a Wattsup Pro power meter connected to a logging machine to record whole-system power consumption at 1-second intervals. We report the power reading at the sustained and prominent power surge that indicates kernel execution. For each kernel, this produces repeatable measurements with only 2–3 watt variance (about 1% of total power). To report GPU power alone, we subtract baseline system power (about 100 W with GPUs unplugged).

Kernel	Parameter	Value	Category	Comment
bfs	block	32–1024	thread block size	The number of threads in each thread block
	npt	1–64	work allocation	The number of nodes processed by each thread (nodes per thread)
	consec	0 / 1	coalescing	Whether a single thread processes consecutive nodes
	pa-attrib	0 / 1	data layout	Whether node attributes are broken apart and stored in parallel arrays
	pa-status	0 / 1	data layout	Whether node status flags are broken apart and stored in parallel arrays
hotspot	x	1–1024	thread block size	The X dimension size of each thread block
	y	1–1024	thread block size	The Y dimension size of each thread block
	t-temp	0 / 1	data layout	Whether to transpose the temperature array in shared memory
	t-power	0 / 1	data layout	Whether to transpose the power array in shared memory
	t-buffer	0 / 1	data layout	Whether to transpose the intermediate buffer array in shared memory
kmeans	ppb	32–1024	thread block size	The number of points processed by each thread block (points per block)
	tpp	1–32	work allocation	The number of threads cooperating on each point (threads per point)
	consec	0 / 1	coalescing	Whether consecutive threads process consecutive features
matrix	x	1–1024	thread block size	The X dimension size of each thread block
	y	1–1024	thread block size	The Y dimension size of each thread block
	tiling	1–1024	work allocation	The number of elements processed by each thread in one iteration
	unroll	1–16	compiler option	The degree to which the innermost loop is unrolled
	t-a	0 / 1	data layout	Whether to transpose the A array in shared memory
	t-b	0 / 1	data layout	Whether to transpose the B array in shared memory
	use-smem	0 / 1	shared memory	Whether to buffer matrix tiles in shared memory
nbody	x	1–1024	thread block size	The X dimension size of each thread block
	y	1–1024	thread block size	The Y dimension size of each thread block
	unroll	1–8	compiler option	The degree to which the innermost loop is unrolled
	n	1024–32768	input size	The number of bodies in the input data
streamcluster	block	32–1024	thread block size	The number of threads in each thread block
	ppt	1–16	work allocation	The number of points processed by each thread (points per thread)
	pa-point	0 / 1	data layout	Whether point structures are broken apart and stored in parallel arrays
	use-smem	0 / 1	shared memory	Whether to buffer point structures in shared memory

TABLE I: Configurable GPU kernel software parameters and their possible values common on both NVIDIA and AMD platforms.

Parameter	Value	Comment
nreg	4–32	The maximum number of registers each thread can use before register spilling
use-l1	0 / 1	Whether to enable L1 caches
large-l1	0 / 1	Whether to use 48 KB L1 caches and 16 KB shared memory (versus 16 KB L1 caches and 48 KB shared memory)

TABLE II: Hardware parameters available only on the NVIDIA platform and their value ranges.

VI. Evaluating the Partitioning Algorithm

A. Training Sample Size and Prediction Accuracy

A detailed regression tree has many uses (see Sections II and VII), but here we use its function as a performance predictor to demonstrate its accuracy. With an increasing number of training samples, trees converge toward a highly accurate description of the space, proving the validity of this partition-based approach. More important, users of our method can also use the accuracy vs. training sample size relationship to adaptively determine how many training samples to collect.

To use a regression tree to predict the performance (or power) of an arbitrary design, start at the root of the tree and follow the branches based on how the parameter settings of the given design compare to the division values. When a leaf node is reached, the average power or performance of sampled designs in that partition is used to predict the queried design.

Traditionally, in statistics, validation samples are used to decide how many sample points are needed to build regression trees. In this method, users/tools initially select some number of UAR samples from the design space to be considered as the validation set. These are used to test tree accuracy, but are never used in the tree forming or partitioning process. (We use 200 validation samples in our results.) Then users select a small initial number of UAR-sampled designs from which to form an initial regression tree. (We use 20 samples as our starting point.) After forming a regression tree for these samples, one can test the performance prediction accuracy of the tree for the 200 validation points. If the prediction accuracy is sufficient, the process stops. If insufficient, additional UAR samples are collected and the regression tree is adjusted to include them. Validation against the reserved samples is repeated. When the prediction accuracy reaches a predetermined level, it means the training samples have had a good coverage of the entire space, and users can stop collecting more samples.

Figure 4 shows the median relative prediction accuracy of trees built using an increasing number of training samples. Except matrix and nbody, which have the largest design spaces, most benchmarks achieve good prediction accuracy with 200 samples: 4% average error for power and 8% average error for performance. To achieve a target 15% prediction accuracy, matrix on NVIDIA needs 3200 samples, and nbody on AMD needs 400 samples. Even 3200 and 400 samples are still much less than 0.1% of the size of matrix's and nbody's design spaces. Among all scenarios, kmeans on AMD requires the most samples relative to its design space size: 0.3%. In the remainder of the paper, all trees are built using these sample sizes (3200 for NVIDIA-matrix's performance, 400 for AMD-nbody's performance, and 200 for all else). Finally, we note that this method is highly adaptable to particular usage needs. When only the top levels of the trees are used to interpret the most salient design criteria for a space, as in Section VII, fewer samples and validation points are needed.

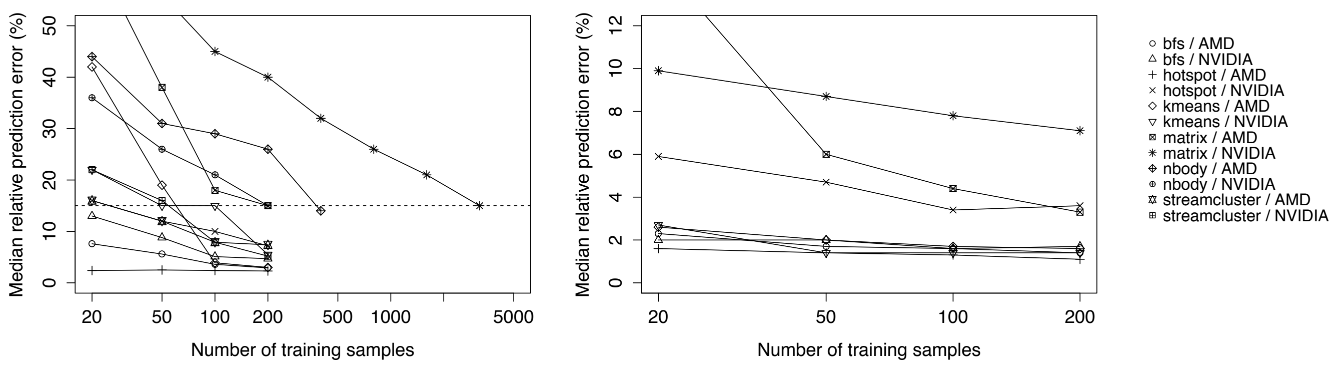


Fig. 4: Prediction accuracy vs. training set size. Users can adaptively compare against the validation set to decide the number of training samples to collect.

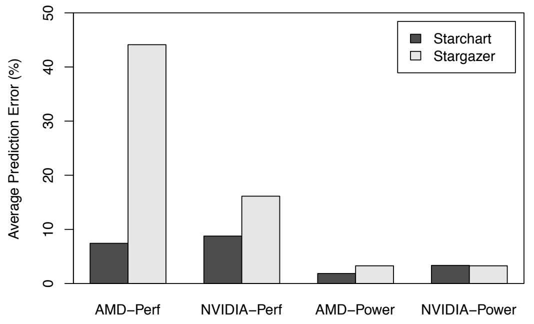


Fig. 5: Starchart outperforms prior work, particularly for modeling performance. Each bar is averaged over validation samples from all benchmarks.

B. *Comparison against Prior Work*

In this section, we compare Starchart's model accuracy to our previous work Stargazer [11]. Both methods use the same training samples and the same validation samples. All samples are obtained from real systems as described in Section V. Training and validation set sizes are described in Section VI-A.

Previous statistical techniques [11, 12, 14] including Stargazer have only been applied to simulations, where the environment is carefully controlled and results are highly repeatable. In contrast, Starchart handles complex high-order parameter interactions, which frequently arise in real-system environments. As a result, when we compare Starchart to Stargazer, the former clearly outperforms the latter in terms of prediction accuracy (Figure 5). In particular, Stargazer is unable to model `matrix`'s and `nbody`'s performance with acceptable accuracy, resulting in high average prediction error for performance models.

While we also want to compare the "best" parameter configurations found by both methods, in practice it is difficult to do so. For some programs (e.g. `matrix` and `nbody`), the subspace structures in their design spaces are very prominent. As a result, Stargazer fails to build a global, additive linear model that is statistically valid across the whole design space. Thus, the best parameter setting found by these statistically invalid models has little meaning, and it should not be used for comparison. For example, for AMD–`nbody`, the baseline runtime is $497.4\,\mu s$. The best design found by Starchart has a predicted runtime of $31.9\,\mu s$, versus an actual runtime of $29.3\,\mu s$ (a 8.8% prediction error and a $17\times$ improvement over baseline). In contrast, the best design found by Stargazer has a predicted runtime of $57.8\,\mu s$, and an actual runtime of $85.2\,\mu s$ (a 32% prediction error and a mere $5.8\times$ improvement over baseline). In summary, for the programs we tested, the "best" configuration found by Starchart always performs better or at least as well as the one found by Stargazer, even though both methods use identical training data.

Overall, while our method is able to characterize real-system evaluation results more accurately than prior work, its true strength is the ability to do subspace-based exploration. This is not offered by any prior work, and it serves as the basis for many novel and practical tool uses (Section VII).

C. *Algorithm Efficiency*

Starchart is highly efficient. For 200 training samples, on a moderately configured laptop computer, it takes only a few seconds for our unoptimized implementation to generate a tree with about 50 partitions. From Figure 3, assuming constant time cost for computing averages, our algorithm's time complexity is $O(m \times n \times \|\bar{V}\|)$, in which $\|\bar{V}\| = (\|V_1\| + \|V_2\| + ... + \|V_n\|)/n$ is the average number of values a parameter may take. Note that $n \times \|\bar{V}\|$ is much less than the design space size $\|V_1\| \times \|V_2\| \times ... \times \|V_n\|$. In summary, the algorithm's complexity is linear in the number of samples and linear in the number of parameters and their choices, ensuring good scalability as more parameters are added.

Starchart can account for interactions among as many parameters as the tree height (easily 6 or more for 40–50 final partitions). In contrast, an iterative approach such as [11] will have exponential growth in complexity when it starts exploring higher-order interactions. In those algorithms, every additional level of interactions (e.g from two-way to three-way) extends the algorithm time cost by n (the number of parameters). The multiplicative complexity growth in time complexity (instead of the linear growth like our approach) is one of the reasons why linear regression approaches are often limited to accounting for only pairwise interactions.

```
// Launch a thread block with block threads
tid = global thread ID
bid = tid / block
for (i = 0; i < npt; i++) {
  if (consec) nodeid = npt * tid + i;
    else nodeid = bid * block * npt + block * i
                + tid % block;
  if (pa-attrib) next = destinations[nodeid];
    else next = nodes[nodeid].destination;
  if (pa-status) {visited[nodeid] = 1;
      tovisit[next] = 1;}
    else {status[nodeid].visited = 1;
      status[next].tovisit = 1;}
}
```

Fig. 6: The core `bfs` kernel has five parameters. Our technique eases the process of selecting good parameter values.

VII. Case Studies

This section presents case studies that demonstrate how our statistical technique efficiently solves practical GPU performance and power optimization problems.

A. Design Space Pruning

When GPU developers optimize their applications for power or performance, they rely largely on developer intuition to select optimizations, a process called "design space pruning". Such approaches can erroneously omit important portions of the design space from potential optimization. For `bfs`, a breadth-first search graph algorithm, this case study shows how our approach differs substantially from prior work in its ability to reveal important and *local* parameter interactions in order to efficiently and accurately optimize parameter settings and prune uninteresting portions of the design space.

As Table I shows, `bfs` on the AMD platform has five tunable parameters. Figure 6 shows the pseudocode for the main kernel.[2] `block` is the number of threads per block, and `npt` controls the number of nodes processed by each thread. `consec` decides whether a single thread processes nodes consecutively, versus at even strides. The remaining parameters, `pa-attrib` and `pa-status`, control data layout, i.e. whether attributes of a node are closely packed or spread out over different arrays. `bfs` is a memory-bound application because it generates many scattered global memory requests to access node attributes and status flags, easily saturating memory bandwidth. Given the possible settings of these five parameters, 98,816 design points are possible. Using only 200 random samples, Starchart is able to gain an accurate picture of localized performance trends in different design subspaces. Figure 7a shows the resulting regression tree.

Without our method, programmers might have hunches or intuitions about how parameters will behave, but these can be quite inaccurate. For example:

Hunch 1: `block` should be the most important parameter. Increasing threads per block usually helps hide memory latency, important for memory-bound applications such as `bfs`.

[2]For simplicity, nodes in the code have only one outgoing edge. Another smaller kernel—not shown or tuned—does cleanup for this kernel.

Reality: The regression tree shows that `block` is actually almost never important anywhere in the design space. This is because for `bfs`, only 64 threads are enough to saturate memory bandwidth. Figure 7b shows this by plotting performance versus `block` for all gathered samples. Clearly there is no strong relationship between these two.

Hunch 2: `npt` should not affect performance. High `npt` can potentially increase data-reuse, but `bfs`'s large memory footprint is unlikely to benefit from the small GPU caches.

Reality: Actually the regression tree illustrates that `npt` has significant impact on performance, especially when `consec = 1`. To further confirm this, Figure 7c shows performance versus varying `npt`. It shows caches can still be helpful for some small `npt` values.

Hunch 3: The 3 memory locality parameters (`consec`, `pa-attrib`, and `pa-status`) should be equally important or equally unimportant, because they are similar optimizations.

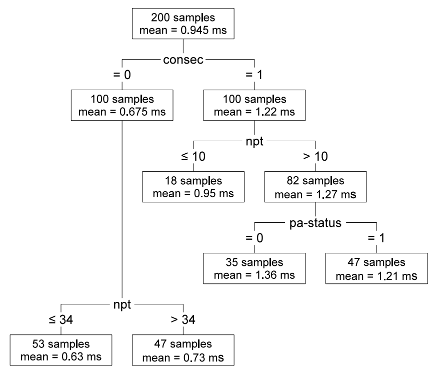
Reality: Contrary to the third hunch, the 3 parameters related to memory locality are not of equal importance. In particular, Figure 7a's regression tree shows `consec` as fairly important and `pa-status` as conditionally important based on the value of `consec`. The third locality parameter, `pa-attrib`, is not important enough to appear. Figure 7c gives the data behind these trends. The `consec` parameter is clearly important because the two clusters that correspond to `consec`=0 are mostly separated from those where `consec` = 1. Likewise, the group where `pa-status` = 1 is distinct from the group where `pa-status` = 0 for `consec = 1`. Careful code analysis shows accesses to node status arrays are more scattered than accesses to node attributes, hence the difference in parameter importance.

This example demonstrates how Starchart builds designer intuition about the design space overall, as well as about relative parameter importance. Prior work has used regression or clustering techniques to group points with similar performance results using hardware performance counter values [17], rather than using input parameters or hardware configurations. As a result, prior clustering techniques might tie low cache miss counter values to high performance, but could not directly guide the designer towards the hardware or software *design decisions* to achieve this. In contrast, our approach's output *is tied directly* to controllable parameters. The tree partitions we identify can lead directly to software optimizations or hardware design decisions.

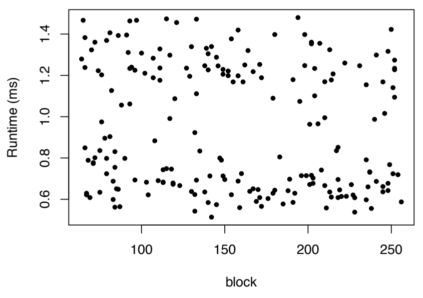
B. Cross-Platform Program Optimization

GPU developers frequently need to optimize their applications for more than one platform, but each platform's distinct power and performance characteristics influence the best configuration choices and how well the best settings can do. Developers may need to (i) optimize the power or performance of an application on each of many different platforms, (ii) select which platform to run an application on according to some power or performance criteria, or (iii) optimize the power or performance of an application simultaneously for several platforms. Starchart supports all three of these.

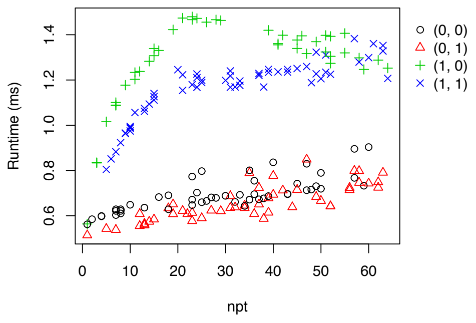
We collect 200 samples for each application on two distinct GPU platforms: NVIDIA's Tesla C2070 and AMD's Radeon



(a) Performance tree



(b) Performance vs. `block`



(c) (`consec`, `pa-status`) in parentheses

Fig. 7: The performance tree of `bfs` on AMD reveals that performance does not have strong dependence on `block`. In addition, performance indeed depends on `npt`, and `consec` and `pa-status` further partition points into fairly separate groups.

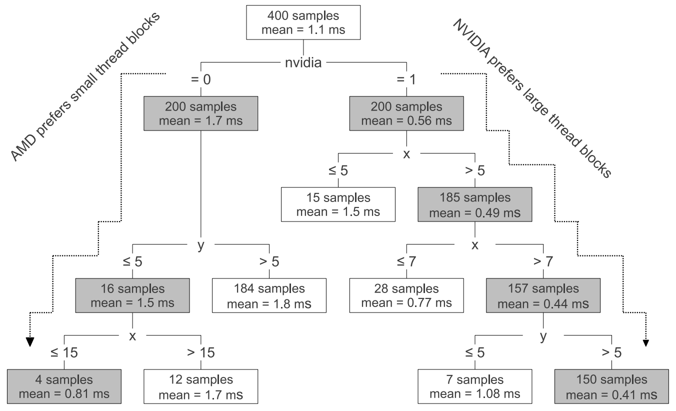


Fig. 8: The cross-platform tree of `hotspot` identifies particularly good designs for AMD and NVIDIA respectively.

HD 7970. Each sample is tagged with a binary parameter, `nvidia`, which takes 1 for NVIDIA runs and 0 otherwise. Our tool then analyzes all of these samples simultaneously, producing the regression tree in Figure 8.

In Figure 8, the highest-performing design subspaces for each platform are shaded in gray. Most notably, our partitioning approach helps a developer see that they are non-overlapping and arise from very different parameter choices on each platform. The best subspace occurs on AMD when blocking factor values are $\texttt{x} \leq 15$ and $\texttt{y} \leq 5$. On NVIDIA, the best options are $\texttt{x} > 7$ and $\texttt{y} > 5$. The reason the subspaces do not overlap is because these platforms contain different amounts of thread resources. The best configurations of `hotspot` for AMD and NVIDIA are 6.3× and 1.3× faster, respectively, than the default parameter configuration provided with the Rodinia distribution, which uses parameter values `x` and `y` that are suboptimal for both platforms. Thus, an auto-tuner developer seeking to identify the best platform-specific settings could use our regression trees to quickly guide design choices based on the hardware encountered.

Cross-platform analysis becomes even more complex when trade-offs vary depending on other parameter choices. For example, Figure 8 indicates that the best NVIDIA subspace is 1.98× faster than the best AMD subspace, but the worst NVIDIA subspace is 1.85× *slower* than the best AMD subspace. The `kmeans` design space (not shown due to space limits) exhibits a similar issue. The best subspace on the AMD platform is 8.42× faster than the best NVIDIA subspace, but the worst AMD subspace is 1.29× *slower* than the best NVIDIA subspace. Starchart helps developers consider a platform choice in concert with the entire configuration of the application's parameter design space.

Finally, developers may want to optimize applications for multiple platforms simultaneously. Our partitioning tool enables developers to recognize which software parameters are so important that they impact power or performance on multiple platforms. For example, the `kmeans` regression tree (not shown) indicates that the most important parameter for performance optimization is `tpp`. The subspace where $\texttt{tpp} > 1$ is 3.93× faster than the subspace where $\texttt{tpp} = 1$, *regardless of platform*. For developers trying to create a generically optimized application, our tool helps identify parameters that impact power or performance on multiple platforms.

In summary, this case study has shown how Starchart's regression trees give developers the insight needed to optimize an application in a platform-dependent or -independent manner or to choose the best possible platform for a given application.

C. Characterizing Program Input Sensitivity

A program's runtime almost always scales with input data. For this reason, performance tuning generally requires the use of "typical" input data, and the resulting conclusions can be highly input-dependent. Clearly, optimizations may vary in complex ways with input characteristics. Thus, this case study shows that our method cleanly handles scaling program input sizes. We treat input size/characteristics as additional design parameters in addition to parameters in Table I; our algorithm then automatically discovers the importance of these parameters and how they interact with other design parameters. Because some input characteristics (e.g. graph connectivity or matrix shape) inevitably affect input size, all performance numbers in this case study are normalized against their respec-

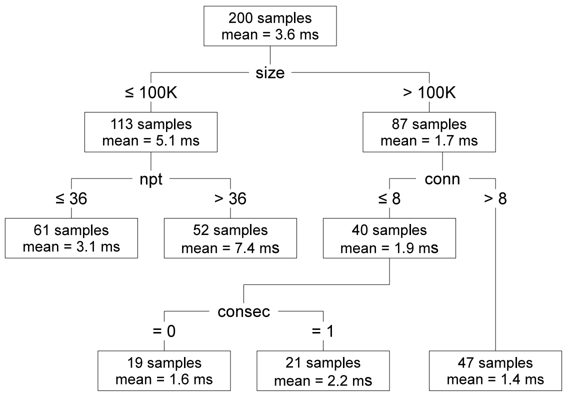


Fig. 9: The `bfs` performance tree on the AMD platform shows its input-sensitivity.

tive input sizes, i.e. we study computational throughput instead of absolute runtime.

For some programs, such as `matrix`, computational throughput is relatively unaffected by input characteristics. We run `matrix` over a range of differently sized and shaped input matrices (from 400×400 up to 3200×3200 elements), and we find that these input parameters do not appear anywhere in the top portion of the generated trees (not shown due to space constraints). This demonstrates that `matrix` scales well with a wide range of input sizes. It also indicates that optimizations based on the parameter settings that *do* appear in the regression trees will be applicable across a wide range of inputs.

For `bfs`, we use a random input graph generator that takes in two parameters: (i) the number of nodes, `size`, which varies from 50K to 1M, and (ii) the average number of edges a node connects to, `conn`, which varies from 1 to 32. The generated tree is shown in Figure 9 and leads to the following observations. First, a graph's `size` is overall a more important performance determinant than graph connectivity (`conn`). This is because graphs with more nodes require more resources (more threads) to process than graphs with more edges (no extra threads). Second, smaller graphs ($\leq$ 100K nodes) and high `npt` values significantly reduce `bfs`'s throughput. This is because when `npt` is high, each thread processes so many nodes that very few thread blocks are launched, not enough to keep all GPU multiprocessors occupied. Our method automatically and clearly spots such abnormal performance issues. Finally, `consec` improves performance for large graphs with low connectivity (`conn` $\leq$ 8). Unlike prior work, Starchart's focus on local subspaces helps identify optimizations that are *conditionally* important for certain input sizes/characteristics.

D. Exploring Power/Performance Trade-offs

GPU system and application designers are increasingly subject to power constraints when optimizing the performance of their designs. However, few systematic approaches have been proposed to tackle power-aware performance optimization. Most prior design space exploration studies [4, 9, 11, 14, 17] can only model one metric at a time. One study proposes modeling power and performance in conjunction [10], but their approach requires extensive user knowledge about the program and imposes strong limits on the types of programs that can be analyzed (e.g. they must saturate memory bandwidth).

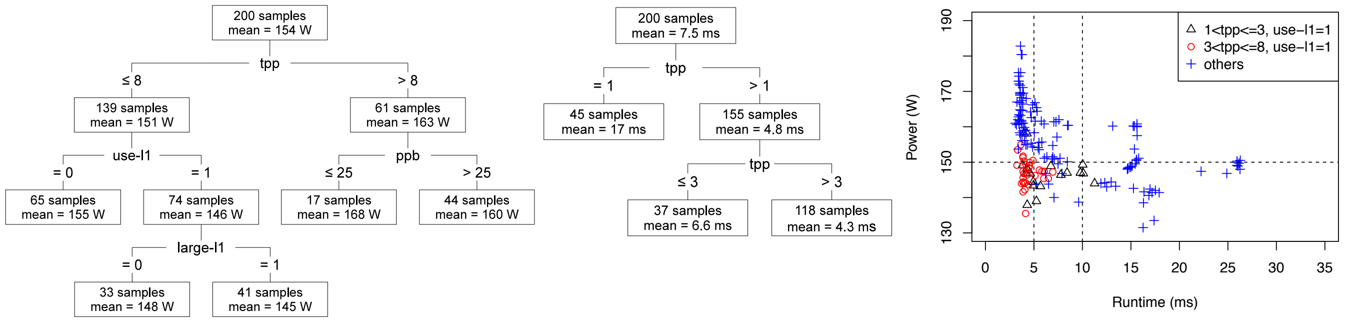
Because Starchart breaks down an application's design space into partitions with distinct power and performance characteristics, users can naturally and efficiently explore design space power/performance trade-offs. Furthermore, because our partitions are derived from the original program parameters, users can immediately know how to set parameters to achieve the power/performance of the partitions that meet their goals.

Figure 10c shows power and performance of the 200 `kmeans` design samples on the NVIDIA platform. Ignoring the different plot symbols, designers would generally have no clue which parameter settings have caused such a wide range of power and performance variation. If, for example, they must limit the GPU power consumption to below 150 W and meet the runtime target of 10 ms (indicated by dotted lines), it is very difficult to know which particular parameter settings would result in designs likely to achieve these goals.

The trees generated by Starchart can easily help answer these questions. Figure 10a shows the power tree of `kmeans`. Clearly, the design subspace with `tpp` ≤ 8 and `use-l1` $= 1$ has a power consumption generally below 150 W. This is due to less intense thread activities (`tpp` $\leq$ 8) and the use of caches (`use-l1` $= 1$). Likewise, `kmeans`'s performance tree (Figure 10b) shows that the design subspace with `tpp` > 1 can generally meet the performance goal, due to increased inter-thread parallelism. By intersecting these two sets and plotting them using different symbols (triangles and circles in Figure 10c), we see these designs occupy most of the targeted power/performance region. Deeper partitioning can better sharpen adherence to certain power or performance constraints. Additionally, per-partition power or performance summary values can be based on the "worst" value for samples in the region (e.g. max power/runtime) instead of the average.

Sometimes, the power or performance goals of a system change dynamically at runtime, due to events such as low battery state, reduced power budget, etc. In these situations, our method can help adaptively determine the necessary program state transitions. For example, assume the program runtime target becomes more stringent from 10 ms to 5 ms, without changing the power budget. Users can look at Figure 10b and decide that `tpp` must be increased to greater than 3. Because the performance target is being tightened rather than relaxed, users just need to go deeper into the tree instead of backtracking. Figure 10c shows designs with `tpp` > 3 (circles) are very likely to achieve the new power/performance goal. Compared to a high-performance high-power design, designs within this region can save up to 47 W (or 26% of total power) with less than 10% performance slowdown.

As a final note, we describe some interesting power tuning observations encountered in our experiments. For some benchmarks such as `bfs`, `kmeans`, and `matrix`, program design choices significantly influence power usage. In particular, global memory traffic is very power-consuming. For example, for `matrix`, buffering content in shared memory saves considerable power versus always fetching content from global memory; for `bfs`, effective use of L1 caches can lower power usage regardless of whether runtime is improved. Additionally, the same program over the same range of op-



(a) Power tree (b) Performance tree (c) Power vs. performance

Fig. 10: Power/performance trade-offs of `kmeans` on the NVIDIA GPU. Dashed lines mark power and performance targets.

timizations has more power variation on the AMD platform than on the NVIDIA platform, presumably because the AMD platform is newer and can better power gate various system components. All power trends described above are easy to spot in the power trees generated by Starchart. As performance-per-watt becomes an increasingly important metric, a power-aware tuning tool like Starchart has high potential to be useful to a large group of users.

VIII. Conclusion

This paper presents a novel partition-based approach for viewing GPU application tuning spaces and an automated algorithm for generating such partitions. The partition-based view is proven to be effective at visualizing and representing a GPU program's complex tuning process and can be utilized to tackle many practical application tuning tasks in a holistic fashion. Experiments show that on two different platforms, for six diverse GPU kernels, and with two different metrics, our method uses samples less than 0.3% of the entire tuning space to build accurate and compact trees representing the inherent hierarchical structure of these spaces. Detailed case studies are presented to show how this algorithm can help solve application tuning problems such as pruning program design parameters, comparing GPU platforms, characterizing program input sensitivity, and doing power-aware performance optimization.

Acknowledgements

This work was supported in part by the National Science Foundation under Grant No. CCF-0916971. We also acknowledge financial and equipment donations from NVIDIA and AMD, as well as the support of the Gigascale Systems Research Center, funded under the Focus Center Research Program (FCRP), a Semiconductor Research Corporation entity. Finally, we thank Daniel Lustig and the anonymous reviewers for their feedback.

References

[1] *AMD APP Profiler User Guide*, AMD Inc.

[2] J. Bergstra *et al.*, "Machine learning for predictive auto-tuning with boosted regression trees," in *Innovative Parallel Computing*, 2012.

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

[4] J. Chen *et al.*, "Tree structured analysis on GPU power study," in *IEEE 29th Intl. Conf. Computer Design*, 2011.

[5] J. W. Choi *et al.*, "Model-driven autotuning of sparse matrix-vector multiply on GPUs," in *Proc. 13th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming*, 2010.

[6] K. Datta *et al.*, "Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures," in *Proc. 2008 ACM/IEEE Conf. Supercomputing*, 2008.

[7] Y. Dotsenko *et al.*, "Auto-tuning of Fast Fourier Transform on Graphics Processors," in *Proc. 14th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming*, 2011.

[8] A. Ganapathi *et al.*, "A case for machine learning to optimize multicore performance," in *Proc. 1st USENIX Conf. Hot Topics in Parallelism*, 2009.

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

[10] S. Hong and H. Kim, "An integrated GPU power and performance model," in *Intl. Symp. on Computer Architecture*, 2010.

[11] W. Jia *et al.*, "Stargazer: Automated regression-based GPU design space exploration," in *Proc. IEEE Intl. Symp. Performance Analysis of Systems and Software*, 2012.

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

[13] M. H. Kutner *et al.*, *Applied Linear Statistical Models*, 5th ed. McGraw-Hill/Irwin, 2005.

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

[15] Y. Li *et al.*, "A note on auto-tuning GEMM for GPUs," in *Proc. 9th Intl. Conf. Computational Science*, 2009.

[16] W.-Y. Loh, "Classification and regression tree methods," in *Encyclopedia of Statistics in Quality and Reliability*, F. Ruggeri *et al.*, Eds. Wiley, 2008, pp. 315–323.

[17] H. Nagasaka *et al.*, "Statistical power modeling of GPU kernels using performance counters," in *2010 Intl. Green Computing Conf.*, 2010.

[18] *Compute Command Line Profiler User Guide*, NVIDIA.

[19] *GPU Computing SDK*, NVIDIA, [Online] http://developer.nvidia.com/cuda/gpu-computing-sdk.

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

[21] J. Sim *et al.*, "A performance analysis framework for identifying potential benefits in GPGPU applications," in *Proc. 15th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming*, 2012.

[22] S. Triantafyllis *et al.*, "Compiler optimization-space exploration," in *Proc. Intl. Symp. Code Generation and Optimization*, 2003.