GPU Performance and Power Tuning Using Regression Trees

WENHAO JIA and ELBA GARZA, Princeton University
KELLY A. SHAW, University of Richmond
MARGARET MARTONOSI, Princeton University

GPU performance and power tuning is difficult, requiring extensive user expertise and time-consuming trial and error. To accelerate design tuning, statistical design space exploration methods have been proposed. This article presents Starchart, a novel design space partitioning tool that uses regression trees to approach GPU tuning problems. Improving on prior work, Starchart offers more automation in identifying key design trade-offs and models design subspaces with distinctly different behaviors. Starchart achieves good model accuracy using very few random samples: less than 0.3% of a given design space; iterative sampling can more quickly target subspaces of interest.

Categories and Subject Descriptors: C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors)

General Terms: Design, Measurement, Performance

Additional Key Words and Phrases: Design space exploration, GPGPU, statistical modeling, decision tree

ACM Reference Format:
Wenhao Jia, Elba Garza, Kelly A. Shaw, and Margaret Martonosi. 2015. GPU performance and power tuning using regression trees. ACM Trans. Architec. Code Optim. 12, 2, Article 13 (May 2015), 26 pages.
DOI: http://dx.doi.org/10.1145/2736287

1. INTRODUCTION

As a popular heterogeneous computing paradigm, CPU-GPU pairs offer high performance and power efficiency for suitable workloads. However, careful performance and power tuning is essential for fully utilizing GPU capabilities. This is because poor GPU software and hardware design choices often have severe consequences. For example, the performance of Section 2's example kernel can differ by 100× depending on the value of a single program parameter. Clearly, finding the "right" parameter settings can be very beneficial; the key is to do this *design space exploration* efficiently.

Traditionally, a GPU software or hardware design space is explored manually, relying on the intuition and experience of the people doing the exploration to navigate the space [Datta et al. 2008; Li et al. 2009; Choi et al. 2010; Dotsenko et al. 2011; Torres and Gonzales-Escribano 2011; NVIDIA 2011, 2014c]. However, such intuition is difficult to establish in the GPU realm due to numerous challenges, such as complex memory

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. This work was also supported by C-FAR, one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA.

Authors' addresses: W. Jia, E. Garza, and M. Martonosi, Princeton University, Princeton, NJ; emails: jiawenhao@gmail.com, {elba, mrm}@princeton.edu; K. A. Shaw, University of Richmond, Richmond, VA; email: kshaw@richmond.edu.

DOI: http://dx.doi.org/10.1145/2736287

hierarchies [NVIDIA 2009, 2012], runtime hardware settings (Section 8.3), and subtle software–hardware interactions [Jia et al. 2012]. These challenges make GPU design spaces especially unfamiliar and complex, making traditional, human intuition–based tuning techniques likely to miss important parts of a given space.

In the past, automated tuning tools have been proposed to replace or complement manual design space exploration. Among them, some linear regression–based approaches have been particularly simple and effective [Lee and Brooks 2006; Joseph et al. 2006; Jia et al. 2012]. However, the use of linear regression means that these methods can only evaluate parameter importance *globally* across the entire design space rather than *locally* within particular subregions. This severely limits their applicability and accuracy for complex design spaces (Section 7.2).

This article proposes *Starchart*, a fully automated, regression tree–based design tuning tool. Starchart uses a small set of design samples to recursively partition a given design space at key inflection points of a design parameter values, capturing a parameter's global performance and power impact as well as its local influence within design subspaces. Due to its ability to model design subspaces, Starchart can model highly complex, real-system software tuning spaces that prior linear regression–based methods cannot model well. In addition, the subspace modeling ability lets Starchart automate some important design tasks by representing them as decision trees; Section 8 demonstrates how to use Starchart in various realistic design scenarios.

We evaluate Starchart using two sampling approaches. In Baseline Starchart, random sampling is used to collect design samples. Only a small portion of a design space (less than 0.3%) needs to be sampled to achieve median prediction errors of 4% to 8%, resulting in a 300× or more tuning productivity improvement. In Iterative Starchart, a more intelligent, iterative sampling approach is proposed and evaluated. It improves accuracy within design space regions of interest by 21% relative to Baseline Starchart.

Overall, this work makes the following contributions:

(1) Starchart's novel partitioning tree approach provides accurate performance or power estimates for complex GPU design spaces, especially for real-system software tuning spaces that have proven challenging to prior linear regression-based methods.
(2) We present an iterative sampling approach that can automatically sample more designs from design space regions that are of particular interest to the user. This improves modeling accuracy and efficiency for these regions under a limited sampling budget.
(3) We present case studies that offer useful GPU insights as well as demonstrate how to use Starchart to automate design tasks.
(4) Although Starchart is evaluated using GPU software tuning problems in this article, it is general enough to handle non-GPU and hardware tuning problems.[1]

In the rest of the article, Section 2 motivates the use of regression trees. Section 3 reviews prior work. Section 4 presents Baseline Starchart, and Section 5 presents Iterative Starchart. Then, using Section 6's methodology, Section 7 evaluates Baseline and Iterative Starchart. Section 8 offers case studies. Section 9 concludes the article.

2. MOTIVATION: DESIGN SPACE PARTITIONING

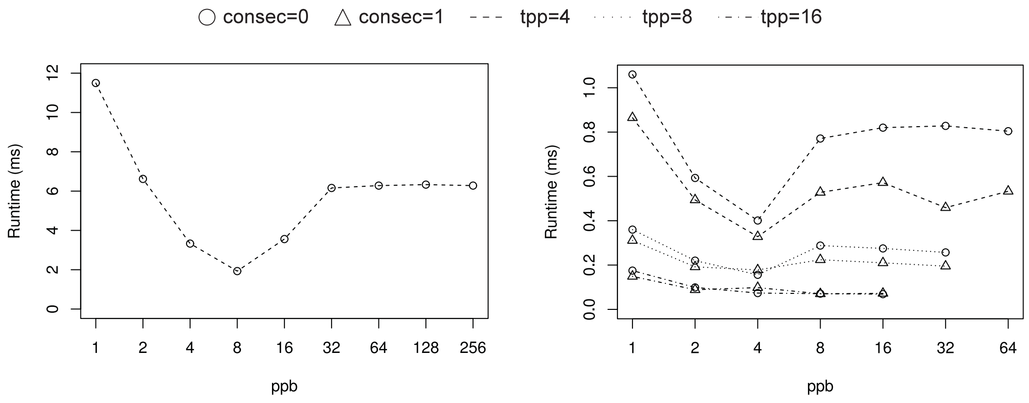
To motivate our work, we present a walk-through example based on the `swap` kernel in the `kmeans` program from the Rodinia benchmark suite, using data collected via real-system measurements on the AMD platform described in Section 6.

Figure 1 shows the kernel code, with parameterizations added to allow possible optimizations. (This code template approach is often used by autotuning tools.) The

[1]Starchart is released under an open-source license at http://www.princeton.edu/~mrm/ for public use.

```
// 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];
}
```

Fig. 1. Parameterized code of the `swap` kernel from Rodinia's `kmeans`.



(a) `tpp` $= 1$, `consec` $= 1$ (default Rodinia kernel) (b) `ppb`–`tpp`–`consec` interactions

Fig. 2. 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.

code transposes 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 (1) `tpp`, the number of parallel threads per point; (2) `ppb`, the number of points per block; and (3) `consec`, a binary flag controlling thread organization and memory access striding. When `consec` is 1, all threads in a block issue consecutive and coalesced accesses while executing the same instruction.

Even in this example with only three parameters, poor parameter settings can lead to 100× performance penalties. However, finding good parameter settings is difficult because the parameters interact in complex ways to affect parallelism, memory coalescing, and—ultimately—performance. For example, `tpp` and `ppb` represent orthogonal forms of parallelism, but their product is limited by the maximum thread count of the GPU platform. Without automation, design space exploration must either require near-exhaustive experimentation or rely on the error-prone manual tuning process.

To begin the example, Figure 2(a) plots the kernel runtime for varying `ppb` values, with `tpp` and `consec` set to 1. (This is the default Rodinia code.) If `tpp` must remain as 1 (e.g., to simplify code), then Figure 2(a) shows that selecting a good value of `ppb` matters. A `ppb` of 8 performs nearly 6× better than other options. However, Figure 2(b) shows that `tpp` > 1 is usually better. When `tpp` can take values higher than 1 (indicating that some threads-per-point parallelism is being exploited), the runtimes are consistently faster than those in Figure 2(a). Furthermore, for larger `tpp` values, runtime varies very little with increasing `ppb`, indicating that `ppb` is not the preferred form of parallelism

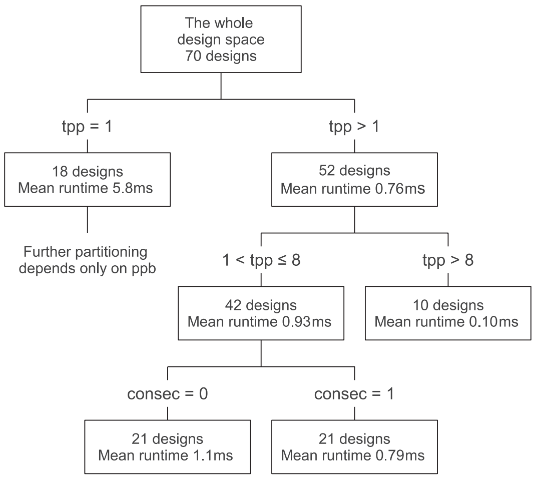

The whole design space 70 designs
tpp = 1
tpp > 1
18 designs Mean runtime 5.8ms
52 designs Mean runtime 0.76ms
Further partitioning depends only on ppb
1 < tpp ≤ 8
tpp > 8
42 designs Mean runtime 0.93ms
10 designs Mean runtime 0.10ms
consec = 0
consec = 1
21 designs Mean runtime 1.1ms
21 designs Mean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

Fig. 3. Design space partitions represented as a tree efficiently summarize performance trends.

in these situations. Finally, Figure 2(b) also shows that although the `consec` parameter does affect performance, this is primarily for small `tpp` values.

Clearly, even what seems to be a fairly simple three-parameter tuning space displays considerable subtlety. Parallelism and memory striding parameters interact differently in different parts of the tuning space. Our work proposes and evaluates automated statistical techniques to determine the relative importance of parameters both globally across the entire design space and locally in particular subspaces. The ability to capture parameters' local interactions makes Starchart more powerful and useful than regression-based approaches.

Figure 3 illustrates a regression tree automatically generated by Starchart for this kernel. Each tree branch 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 $1 < \texttt{tpp} \leq 8$, because $\texttt{tpp} > 8$ is the optimal global solution and `consec` is irrelevant under that subspace. Furthermore, Starchart can easily capture and express arbitrarily high levels of interactions (e.g., the three-way interaction between `tpp`, `ppb`, and `consec`) simply by adding levels to the tree. Existing linear regression techniques only consider either manually specified parameter interactions [Lee and Brooks 2006] or just pairwise interactions [Jia et al. 2012].

3. RELATED WORK

Specialized GPU autotuning frameworks exist for specific algorithms [Datta et al. 2008; Li et al. 2009; Choi et al. 2010; Dotsenko et al. 2011]. However, they tend to focus more on identifying and presenting key algorithmic choices than on automatically pruning them. Considerable user experience or exhaustive experimentation is often needed to search the resulting design spaces. In contrast, Starchart can automatically detect the important design parameters and their interactions among a large number of unpruned parameters, easing the creation of a new autotuner. Meanwhile, Starchart can also replace or aid the heuristic search algorithms used in several more general autotuning frameworks [Fursin and Temam 2010; Ansel et al. 2014].

Various automation methods have been proposed to search and navigate software tuning spaces, such as CPU kernel genetic tuning [Moilanen and Williams 2005], compiler optimization pruning [Triantafyllis et al. 2003], and optimal GPU code searching [Bergstra et al. 2012; Ganapathi et al. 2009]. They differ from Starchart in several aspects. First, in terms of problem specification, Starchart builds application-specific performance and power models, whereas some existing work only assesses compiler optimizations common across all applications [Triantafyllis et al. 2003]. Second, in terms of scope, Starchart is validated for multiple applications on two GPU platforms using both performance and power as metrics, whereas some existing work is evaluated only for one particular application [Bergstra et al. 2012; Ganapathi et al. 2009]. Finally, and perhaps most importantly, prior machine learning–based work mostly aims to find the globally optimal program configuration [Moilanen and Williams 2005; Bergstra et al. 2012; Ganapathi et al. 2009]. In contrast, because statistical models are generally more transparent and easier to interpret than machine learning models, Starchart not only finds the global optimum but also reveals local optima, parameter interactions, and performance trends. In other words, Starchart offers more useful design information to its users than conventional optimal design search algorithms can.

Due to the simplicity and effectiveness of statistical methods, linear regression has become the core of several design space exploration tools [Lee and Brooks 2006; Joseph et al. 2006; Jia et al. 2012]. However, the use of linear regression in these tools means that they cannot distinguish a parameter's effects in local regions of a design space. This not only severely reduces their modeling accuracy for complex design spaces, as shown in Section 7.2, but also prevents them from being used to handle many useful subspace-based design problems, such as the Starchart case studies in Section 8. One study [Magni et al. 2013] uses a similar regression tree approach as Starchart, but it deals with a specific program optimization—thread coarsening—instead of proposing a general tuning framework like Starchart does.

Other GPU performance models (e.g., those based on program analysis [Ryoo et al. 2008; Hong and Kim 2009; Sim et al. 2012]) and power models (e.g., those based on performance counters [Nagasaka et al. 2010; Chen et al. 2011]) exist. The former usually impose strong limits on the types of programs that can be analyzed and are difficult to automate. 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 [Kutner et al. 2005], which is studied extensively in statistics and machine learning fields. Some current studies [Loh 2008] improve on the classic method we use. They can be easily applied to our core algorithm to improve its accuracy, efficiency, or both.

4. STARCHART: AUTOMATED DESIGN SPACE PARTITIONING

This section describes the various steps in the Starchart workflow.

4.1. Design Sample Collection

Parameter selection. To use Starchart on an application, a set of parameters that define the dimensions of the tuning space must first be identified. Parameters may relate to software (e.g., blocking factor, thread count) or to hardware (e.g., active core count, system clock rates). Parameters can either be binary in nature (e.g., whether to turn on or off an optimization) or can take numeric values across a possible range. Parameters can even be choices, such as including "NVIDIA vs. AMD" as a design option [Jia et al.

2013]. Because statistical partitioning is effective at automatically pruning design spaces and identifying *interesting* subspaces, one can be generous in including any tuning parameters that *may* matter.

Sampling program designs. In traditional autotuning, 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 baseline approach uses as input only a modest set of designs sampled from the space uniformly at random (UAR). Section 7.1 shows that less than 0.3% of all possible designs are needed to accurately characterize the whole space, resulting in a more than 300× productivity improvement. In some design scenarios, we are more interested in particular parts of a given design space, such as the best performance or lowest power usage regions. We may wish to sample more from these regions to get a higher model accuracy in these regions. Section 5 describes an iterative sampling approach that can achieve this goal.

Evaluations. Each randomly selected sample point represents a specific design that must be evaluated. This could be either performance or power measurements, and it could be either real-system measurements or simulation. In this work, we use real-system power and performance measurements (Section 6). Thus, the input to Starchart's partitioning algorithm is a set of UAR sampled design points (i.e., their parameter settings) and the resulting performance or power measurements.

4.2. Automated Partitioning Algorithm

Algorithm 1 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.

Developing intuition about relative parameter importance requires analyzing how performance 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 3, 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 find split points where the samples in the two subsets fall into distinct, mostly nonoverlapping groups. If two distinct subsets exist, this implies that this parameter setting clearly influences performance; otherwise, the parameter setting does not have a clear influence on performance. Starchart uses the sum of squared errors (SSE), which is the sum of squared differences between performance of samples in a set and the average performance of all samples in the set, to compare splits, as explained later, although other metrics could be used as well.

In Figure 3, we see that the subdivision of the original set creates two subsets with widely disparate mean runtimes: 5.8ms versus 0.76ms. Clearly, 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

ALGORITHM 1: Recursively partition a program design space. The metric here is performance; power can be modeled similarly.

```
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
1  List of pending partitions $C = \{S\}$;
2  Partitioning history $H = \{\}$;
3  repeat
4      foreach partition $c$ in $C$ do
5          $\bar{r}$ = the average performance of all points $s_j$ in $c$;
6          $\bar{r}$ is used as the modeled performance of all points in $c$;
7          $r_j$ is the performance of $s_j$;
8          sum of squared errors $SSE_0 = \sum_j (r_j - \bar{r})^2$;
9          forall the parameter $P_k$ do
10             forall the possible parameter value $v_l$ in $V_k$ do
11                 split $c$ into two partitions $c_1$ and $c_2$;
12                 $c_1$ has the samples from $c$ with $P_k \leq v_l$;
13                 $c_2$ has the samples from $c$ with $P_k > v_l$;
14                 $\bar{r_1}$ = the average performance of points in $c_1$;
15                 $\bar{r_2}$ = the average performance of points in $c_2$;
16                 $SSE_1 = \sum_j (r_j - \bar{r_1})^2$ for $s_j \in c_1$;
17                 $SSE_2 = \sum_j (r_j - \bar{r_2})^2$ for $s_j \in c_2$;
18                 $SSE_{12} = SSE_1 + SSE_2$;
19             end
20         end
21         remove $c$ from $C$;
22         find the smallest $SSE_{12}$ across all $(P_k, v_l)$ pairs;
23         $SSE_{min}$ = this smallest $SSE_{12}$;
24         $(P_{min}, v_{min})$ = the param-value pair that produces $SSE_{min}$;
25         $SSE_0 - SSE_{min}$ is the reduction in error;
26         if $SSE_0 - SSE_{min} > threshold$ then
27             split $c$ into two partitions $c_l$ and $c_h$;
28             $c_l$ has the samples from $c$ with $P_{min} \leq v_{min}$;
29             $c_h$ has the samples from $c$ with $P_{min} > v_{min}$;
30             add $c_l$ and $c_h$ to $C$;
31             add $((P_{min}, v_{min}), (c, c_l, c_h))$ to $H$ as part of the final tree;
32         end
33     end
34 until $C$ is empty;
output: Partitioning history stored in $H$ represented as a tree
```

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 Algorithm 1, 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 through 20). It then sorts all of the different sample splitting choices by the sum of 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. In other words, 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 through 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 through 34).

Starchart's end product is a tree where nodes are design space partitions, and the edges out of them specify the value ranges for a particular parameter in each of the two subtrees. These trees have two key characteristics. First, parameter splits highest in the tree have the highest overall influence on performance. This is because Starchart greedily orders splits based on parameter values that reduce SSE the most. Second, for each partitioned subspace, independent choices can be made regarding subsequent further partitions. This is key; it means that 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 technique's efficacy and differentiates it from prior work that only recognizes globally important parameters [Lee and Brooks 2006; Jia et al. 2012].

4.3. Algorithm Options

Tree node models. We use average and SSE values in each tree node in our implementation of Starchart because 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 [Jia et al. 2012] 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 Algorithm 1, splitting might continue until there is only one point per set. This level of detail is rarely necessary. We set the *threshold* value to 0 for performance trees, meaning that 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 being split, it means that the natural variation in samples in this partition cannot be attributed to any particular parameter's choice of value. For power trees, we set *threshold* to 3W, the instrument error, as explained in Section 6.

In our experiments, 200 samples result in 40 to 50 final partitions with the preceding stopping criterion. Section 7.1 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 of those splits. They can focus on the top or the subtrees 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.

5. AN ITERATIVE STARCHART APPROACH

The UAR sampling used in the Baseline Starchart algorithm described so far is easy to implement and ensures good coverage of the whole design space. However, in many practical GPU tuning problems, depending on their design goals, users are often more interested in some parts of a design space than in other parts. With this in mind, this section answers the following question: under a limited sampling "budget," can we modify the Baseline Starchart algorithm to specifically emphasize regions of interest without sacrificing model accuracy over the whole design space?

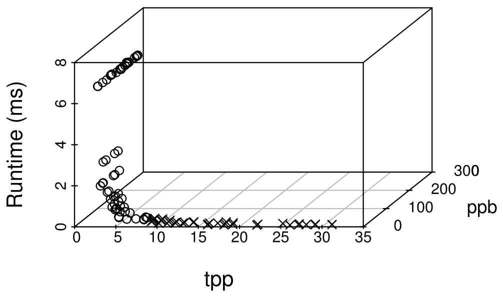


Fig. 4. For swap in kmeans, the part of the design space with higher tpp values ($\times$ symbols) contains designs with lower runtimes and thus deserves more samples.

5.1. Motivation: Focusing on Areas of Interest

The bulk of time required to use Starchart is devoted to gathering performance (or power) data for the sample points. Therefore, when certain regions in a design space are of more interest to a designer, the designer may want to make the best use of a limited sampling budget to improve prediction accuracy for those regions. (Whereas we define regions of interest as the best-performing regions in the rest of the article, many other criteria can be used, such as lowest power usage, performance-power Pareto frontiers, etc.) In the rest of this section, the same kernel from Section 2 is used to explain how an iterative sampling approach can help to achieve this goal.

Figure 4 shows how tpp and ppb values determine this kernel's performance (z-axis) for 100 UAR samples. (The parameter consec can take either 0 or 1, not distinguished in the figure.) The performance trends revealed in Section 2 are more clearly visualized in Figure 4: high tpp values generally result in designs with lower runtimes. Hence, if the goal is to find high-performance kmeans designs, a designer may want to focus on detailed parameter choices in the region of the design space with high tpp values—shown as the part occupied by "$\times$" symbols in Figure 4. This can be accomplished by assigning more of the sampling budget to that region.

We use an iterative sampling approach to achieve such emphasis. In other words, after an initial sampling round that gives us information about the locations of the regions of interest, we strategically take more samples from those regions in a second sampling round. In Figure 4, this means that we will sample more points from the high tpp region. Then, samples from both rounds are used together to build a Starchart tree. Because first-round samples, which cover the entire design space, are still used in the tree-building process, the generated tree maintains good accuracy over the whole space.

Note that our goal is to include more high-performance samples in our model without increasing total sample counts. This differs from many existing approaches, such as some active learning methods [Settles 2010], which increase sample counts to improve model accuracy. The next section describes our iterative sampling approach in detail.

5.2. The Iterative Starchart Algorithm

From here on, *Baseline Starchart* refers to the Starchart algorithm with the default UAR sampling approach described in Algorithm 1, and *Iterative Starchart* refers to the modified algorithm with iterative sampling.

As mentioned in Section 4, the input to Baseline Starchart's regression analysis is a small sampled set of possible designs from the parameter space. This set is sampled UAR from the global parameter space and constitutes at most 0.3% of the design space (Section 7.1). In Iterative Starchart, we explore the possibility of devoting fewer of the "budget" of sample points to the regions of the design space that are far from optimal. Since gathering runtime data for each sampled point is the most time-consuming part of Starchart, this allows the tool to focus its time on the most interesting parts of the design space.

Rather than using a single level of UAR sampling to collect samples for the regression analysis, Iterative Starchart performs a preliminary phase that identifies a "region of interest." Then, additional design samples from that region are collected. We then perform regression analysis and create the Starchart tree with the additional points *along with* points initially sampled UAR (to maintain good global performance prediction for the entire space). Algorithm 2 shows the workflow of Iterative Starchart. Note that in Algorithm 2, the iterative sampling technique (lines 1 through 4) selects the points to be used as the input S to Algorithm 1 on line 5.

ALGORITHM 2: Create a refined Starchart training set using an iterative sampling method.

```
  input: $D$, a design space specified with $n$ design parameters $P_1, P_2, \ldots, P_n$
  input: $V_i$, the set of values that parameter $P_i$ may take
  input: $I$, the number of points to sample in the first round to form the initial sample set $S_I$
  input: $t$, the percentage of best-performing points taken from $S_I$ to form a bounding box
  input: $A$, the number of points to sample in the second round to form the additional sample
         set $S_A$
1 Sample $I$ designs from $D$ UAR and label the set $S_I$;
2 Sort designs in $S_I$ by decreasing performance and take the top $t\%$ of these designs to form
  set $S_t$ ($S_t \subseteq S_I$);
3 Create a bounding box, $B$, by tracking the upper and lower limits of values seen for each
  parameter, $P_i$, across all samples in $S_t$;
4 Sample additional $A$ designs from $D$ such that these designs fall within $B$ and form a set $S_A$;
5 Run Algorithm 1 with $S_I \cup S_A$ as its input sample set $S$.
  output: a Starchart tree that emphasizes the region of interest while having acceptable
          overall accuracy
```

Algorithm 2 starts with I designs sampled UAR, forming a set S_I. The algorithm then sorts these points by the metric of interest (e.g., runtime) and selects the top $t\%$ of these points to form a set S_t. This cutoff value t can be made more or less selective depending on the degree of emphasis that we seek. The algorithm defines a specific region of interest by analyzing these top $t\%$ designs. In particular, it identifies the minimum and maximum values seen for each parameter, P_i, across all designs in S_t. These upper and lower limits of all n parameters form an enclosed n-dimensional area (i.e., a bounding box B) around points in S_t. B contains samples that perform well under our chosen metric of interest, and we proceed to sample A additional points from D that fall within B and form a sample set S_A. S_A reflects our guided sampling emphasis on a region of interest, whereas S_I ensures our global sample coverage. Finally, the union of S_I and S_A are fed into Algorithm 1 as its input sample set S to train an Iterative Starchart tree.

6. METHODOLOGY

The methodology described in this section is used for all results in Section 7.

Benchmarks studied. As shown in Table I, we use six GPU kernels from the Rodinia suite [Che et al. 2009] and NVIDIA GPU SDK [NVIDIA 2014a]. 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. Some of these parameters already exist in

Table I. Configurable GPU Kernel Software Parameters Common on Both NVIDIA and AMD Platforms

Kernel	Parameter	Value	Category	Comment
`bfs`	`block`	32–1024	Thread block size	Num of threads in each thread block
	`npt`	1–64	Work allocation	Num of nodes processed by each thread
	`consec`	0/1	Coalescing	Each thread processes consecutive nodes?
	`pa-attrib`	0/1	Data layout	Store node attributes in parallel arrays?
	`pa-status`	0/1	Data layout	Store node status flags in parallel arrays?
`hotspot`	`x`	1–1024	Thread block size	X dimension size of each thread block
	`y`	1–1024	Thread block size	Y dimension size of each thread block
	`t-temp`	0/1	Data layout	Transpose temperature array in shared mem?
	`t-power`	0/1	Data layout	Transpose power array in shared mem?
	`t-buffer`	0/1	Data layout	Transpose buffer array in shared mem?
`kmeans`	`ppb`	32–1024	Thread block size	Num of points processed by each thread block
	`tpp`	1–32	Work allocation	Num of threads cooperating on each point
	`consec`	0/1	Coalescing	Neighbor threads process consecutive features?
`matrix`	`x`	1–1024	Thread block size	X dimension size of each thread block
	`y`	1–1024	Thread block size	Y dimension size of each thread block
	`tiling`	1–1024	Work allocation	Num of elements processed by one thread
	`unroll`	1–16	Compiler option	Degree of unrolling of innermost loop
	`t-a`	0/1	Data layout	Transpose `A` array in shared mem?
	`t-b`	0/1	Data layout	Transpose `B` array in shared mem?
	`use-smem`	0/1	Shared memory	Buffer matrix tiles in shared mem?
`nbody`	`x`	1–1024	Thread block size	X dimension size of each thread block
	`y`	1–1024	Thread block size	Y dimension size of each thread block
	`unroll`	1–8	Compiler option	Degree of unrolling of innermost loop
	`n`	1024–32768	Input size	Num of bodies in the input data
`stream-cluster`	`block`	32–1024	Thread block size	Num of threads in each thread block
	`ppt`	1–16	Work allocation	Num of points processed by each thread
	`pa-point`	0/1	Data layout	Store point structures in parallel arrays?
	`use-smem`	0/1	Shared memory	Buffer point structures in shared mem?

Note: The listed value ranges are the union of both platforms.

Table II. Hardware Parameters and Values Available Only on the NVIDIA Platform

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 48KB L1 caches and 16KB shared memory (vs. 16KB L1 caches and 48KB shared memory)

the kernels; others are our own addition to explore further optimizations. The size (i.e., the total number of possible program designs) of each design space and the measured runtime distributions of sampled designs are shown later in Table III. (When a UAR-sampled configuration cannot be run on a particular platform, it is not counted in the sample set.) 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, etc.), the parameter values listed in Table I are the union of all possible values on both platforms. For example, in actual experiments, the `block` parameter of `bfs` starts at 32 on the NVIDIA platform and 64 on the AMD platform, but Table I lists the start value as 32 for simplicity. For every benchmark, either `block` or $\texttt{x} \times \texttt{y}$ indicates the total number of threads per thread block. Finally, three parameters are available only for NVIDIA, listed in Table II.

Table III. Design Space Sizes and the Runtime Distributions of Designs Sampled from these Spaces (in Milliseconds)

Benchmark	Design Space Size	Min	1st Qt.	Mean	3rd Qt.	Max
`bfs-amd`	131K	0.51	0.59	0.89	1.22	1.54
`bfs-nvidia`	1.84M	1.08	1.25	1.90	2.50	4.00
`hotspot-amd`	524K	0.31	1.71	1.74	1.81	1.94
`hotspot-nvidia`	67.1M	0.28	0.35	0.55	0.60	3.95
`kmeans-amd`	61.4K	0.046	0.12	1.99	2.69	6.31
`kmeans-nvidia`	1.05M	3.08	3.87	8.07	11.14	26.28
`matrix-amd`	75.5M	3.28	7.09	22.37	27.42	108.20
`matrix-nvidia`	47.2B	11.15	37.47	13.28	15.15	5756.00
`nbody-amd`	3.15M	0.03	0.26	6.62	4.91	204.50
`nbody-nvidia`	1.61B	0.06	0.61	28.51	24.41	634.00
`streamcluster-amd`	65.5K	0.38	0.56	1.20	1.66	3.03
`streamcluster-nvidia`	2.10M	0.90	1.63	4.42	6.44	24.05

Performance and power measurement platforms. We experiment on two systems: (1) an NVIDIA Tesla C2070 GPU and (2) an AMD Radeon HD 7970 GPU. The AMD GPU is newer than the NVIDIA GPU. Having two systems with distinct capabilities is useful for comparison, despite disparate technologies.

To collect kernel execution times, vendor-supplied profiling tools are used [NVIDIA 2014b; AMD 2012]. The runtime experiments are run separately from the power experiments. Because on a real system repeated runs of even the same kernel would produce different measured runtimes, each kernel is executed at least 10 times and the average execution time taken and used by Starchart. (However, even without averaging, since Starchart is a statistical method, it is naturally robust against measurement variance like this.) 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 8.4 and `nbody`), the runtimes are normalized against input size.

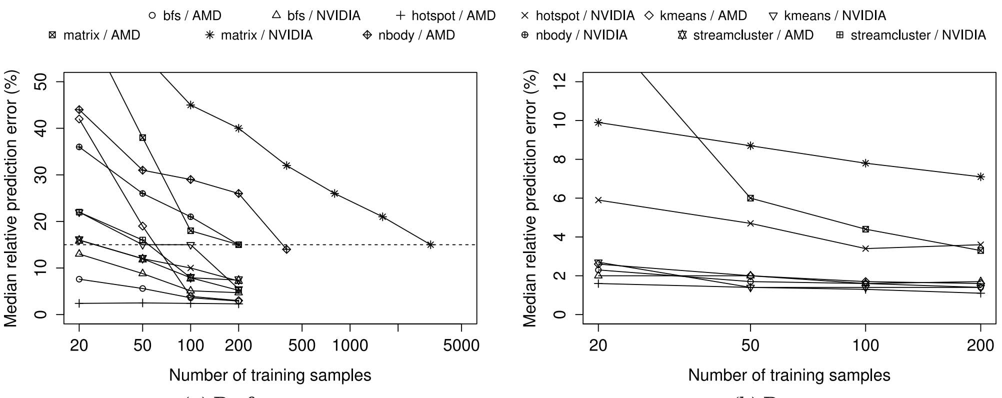
For real-system power measurements, we use the same methodology as Hong and Kim [2010]: a Wattsup Pro power meter connected to a logging machine to record whole-system power consumption at 1-second intervals. (To report GPU power alone, we subtract baseline system power: about 100W with GPUs unplugged.) Because a kernel usually takes only several milliseconds to execute (Table III), the GPU repeatedly executes the same kernel in a loop, resulting in a sustained and prominent power surge. We measure and report the power consumption level after it has stabilized over a long period of time while the GPU keeps executing the same kernel. For each kernel, this produces repeatable measurements within a 2 to 3 W range (about 1% of total power). Because the power meter also has a 1.5% measurement error, we use 3W as the stopping criterion (i.e., *threshold* in Algorithm 1) for Starchart power trees to avoid growing unnecessary nodes.

7. EVALUATING THE PARTITIONING ALGORITHM

Section 7.1 presents the accuracy of Baseline Starchart. Section 7.2 compares this with our prior work, Stargazer [Jia et al. 2012]. Sections 7.3 and 7.4 discuss algorithmic optimality and efficiency, respectively. Finally, Section 7.5 evaluates Iterative Starchart.

7.1. Baseline Starchart: Prediction Accuracy Versus Training Set Size

A detailed regression tree has many uses (see Sections 2 and 8), but here we use its function as a performance predictor to demonstrate its accuracy. To use a regression tree to predict the performance (or power) of an arbitrary design, start at the root of the



(a) Performance (b) Power

Fig. 5. Prediction accuracy versus training set size. Users can adaptively compare against the validation set to decide the number of training samples to collect.

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.

In statistics, validation samples are often used to decide how many sample points are needed to build regression trees. In this method, users or 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 and are never used in the tree-forming process. (We use 200 validation samples in our results.) Then, a small number of separately UAR-sampled designs are used to form an initial tree. (We use 20 samples as our starting point.) After forming a regression tree with these samples, one can use the validation points to test the performance prediction accuracy of the tree. If the prediction accuracy is sufficient, the process stops. Otherwise, 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 that the training samples provide good coverage of the entire space, and users can stop collecting more samples.

Figure 5 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% error for power and 8% error for performance. To achieve a target 15% prediction error, `matrix` on the NVIDIA platform needs 3,200 samples, and `nbody` on the AMD platform needs 400 samples. (Even 3,200 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 the AMD platform requires the most samples relative to its design space size: 0.3%. Unless otherwise stated, in the rest of the article, all trees are built using these sample sizes: 3,200 for NVIDIA-`matrix`'s performance, 400 for AMD-`nbody`'s performance, and 200 for all others. Finally, 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, fewer training and validation samples are needed.

7.2. Baseline Starchart Versus Stargazer

For this section's comparison, both Starchart and our previous Stargazer method use the same training and validation samples. All samples are obtained from real systems as described in Section 6. Training and validation set sizes are described in Section 7.1.

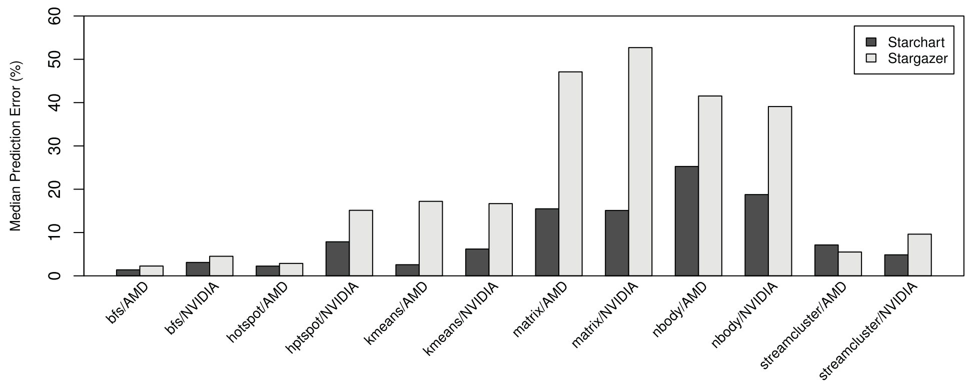


Fig. 6. Baseline Starchart models performance better than Stargazer for most kernels.

Previous statistical techniques [Lee and Brooks 2006; Joseph et al. 2006; Jia et al. 2012], including Stargazer, have only been applied to simulations, where the results are highly repeatable. In contrast, both Starchart variants handle the variability of real-system measurements. As a result, when we compare Starchart to Stargazer in terms of performance modeling accuracy in Figure 6, the former clearly outperforms the latter. In particular, Stargazer cannot model `matrix`'s and `nbody`'s performance with acceptable accuracy, resulting in high median prediction errors. One exception is `streamcluster` on AMD, whose design space is simple enough to be modeled by both Stargazer and Starchart, and the former actually slightly outperforms the latter. Overall, Starchart achieves a 9.2% median prediction error averaged over all benchmarks, which is much lower than Stargazer's 21.2% error.

7.3. Baseline Starchart Tree Optimality

Next, we use the following two steps to evaluate the *optimality* of Starchart trees—that is, how well a Starchart tree characterizes a given design space: (1) obtain a small sample set that has a good coverage of the entire design space and (2) evaluate how well the Starchart tree generated by our algorithm fits the observed samples and, in particular, whether this tree finds the "best" parameter configuration in the sample set.

Section 7.1 has explained how we use validation sets to estimate the number of training samples needed to ensure a good coverage of the whole design space. Alternatively, classic statistical metrics such as confidence interval estimation [Kutner et al. 2005] can also be used to determine the desired sample count. Either way, in our experiments, a few hundred samples are usually statistically guaranteed to cover the studied design spaces fairly well. In particular, they are statistically likely to include designs that perform similarly as the "best" designs for most of our benchmarks.

Given a good enough sample set, Starchart uses least squares partitions—in other words, a split for any partition should minimize the SSE of the two subpartitions. It has been proven that optimal least squares partitions must have contiguous elements inside each subpartition [Fisher 1958], and thus Starchart's method of testing $N-1$ possible splits for a partition with N unique numerical design parameter values, instead of testing all 2^N possibilities, is sufficient for finding a locally optimal least squares split. Namely, the trees generated by Starchart are steepest-descent greedy and stepwise optimal.

However, in general, these stepwise optimal trees are *not* globally optimal. In fact, finding a globally optimal tree that fits observed data with the minimum least squares

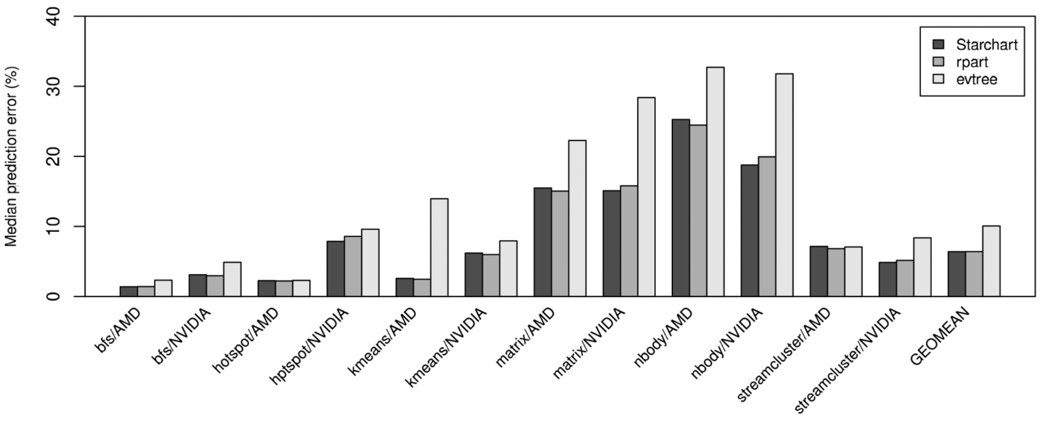


Fig. 7. Starchart achieves competitive performance compared to two R packages.

error is an NP-complete problem [Hyafil and Rivest 1976]. In practice, a greedily induced tree like Starchart is reasonably good, giving classification or regression results that are very close to those of the optimal tree [Murthy and Salzberg 1995]. To prove this, using the validation methodology in Section 7.1, Figure 7 compares three algorithms: Starchart, the `rpart` routine in R [R Core Team 2013] based on the CART method [Breiman et al. 1984], and `evtree`, a state-of-the-art evolutionary algorithm.

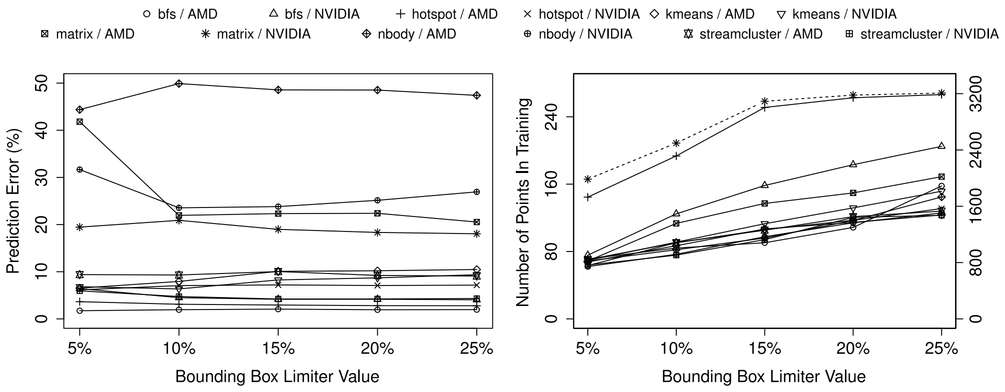
Starchart achieves a very similar prediction accuracy as `rpart`, even though the latter uses a slightly more sophisticated ANOVA [Freedman et al. 2007] split condition than Starchart's simple SSE comparison method. Compared to `rpart`, Starchart has a simpler algorithm and less code, but the simplicity does not diminish its accuracy and applicability for GPU tuning problems. The `evtree` algorithm actually gives worse results than Starchart and `rpart`, likely for two reasons. First, because of the time and space complexity of evolutionary algorithms, `evtree` must cap its tree heights and evolutionary iteration counts to fairly low values. This significantly limits the scale of the problems that `evtree` can handle, as well as the final tree's accuracy. Second, a globally optimal tree trained from sampled designs is not necessarily better at predicting unsampled designs—that is, an optimal tree search algorithm may overfit observed data.

Finally, while we want to compare the "best" parameter configurations found by Starchart with those found by exhaustive search, in practice this is difficult due to the extremely large design spaces that we explore (see Table III). However, for any benchmark, a Starchart tree—even if not globally optimal—can at least identify the partition that contains the "best" design in the sample set. This partition is likely to also contain the "best" design in the whole space and is often small enough to be exhaustively searched.

7.4. Baseline Starchart Algorithm Efficiency

Starchart is highly efficient. For 200 training samples, on a computer equipped with an Intel Core i5 processor, our implementation generates a tree with about 50 partitions in only a few seconds. This section gives a simple analysis of its algorithm efficiency.

In Algorithm 1, the main work is done inside the innermost body (lines 11 through 18) of the triple-nested loop. If we represent the number of elements (i.e., possible parameter settings) in the set V_i (i.e., the value range for the parameter P_i) as $|V_i|$, the loops on lines 9 and 10 combined together iterate $O(n \times \overline{|V|})$ times, where $\overline{|V|} = (|V_1| + |V_2| + \cdots + |V_n|)/n$ is the average number of values a parameter may take. (Note



(a) Prediction error for points within BB (b) Total training samples ($I + A$)

Fig. 8. Iterative Starchart's prediction accuracy generally improves as more total training samples are used. In Figure 8(b), only the dashed line uses the right y-axis. Subtract I (800 for dashed line and 60 for others) from Figure 8(b)'s plotted values to get A.

that $n \times \overline{|V|} = |V_1| + |V_2| + \cdots + |V_n|$ is much less than the design space size $|V_1| \times |V_2| \times \cdots \times |V_n|$.) Meanwhile, the loop on line 4 iterates as many times as the number of partitions in the final tree, bounded by the total number of design samples (i.e., $O(m)$)—the worst case happens when the final tree is fully grown and every leaf partition contains exactly one sample. Finally, lines 11 through 18 take a variable amount of time to execute depending on the number of samples in a particular partition, also bounded by m. Combining all preceding analyses gives a worst-case time complexity of $O(m^2 n\overline{|V|})$ for the whole algorithm. In other words, the algorithm's complexity is quadratic in the number of samples and linear in the number of parameters and their choices, ensuring good scalability as the design space grows in size.

Starchart can account for interactions among as many parameters as the tree height (easily 6 or more for 40 to 50 final partitions). In contrast, a linear regression approach like Stargazer 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 growth in time complexity is one reason linear regression approaches are often limited to accounting for only pairwise interactions.

7.5. Iterative Starchart: Prediction Accuracy Versus Bounding Box Size

Next, we evaluate the performance prediction accuracy of Iterative Starchart given by Algorithm 2. To reiterate, the goal of Iterative Starchart is to provide better accuracy in a region of interest (e.g., the top $t\%$ best performing designs) without unduly sacrificing accuracy over the whole design space. This is particularly valuable when we want to target certain regions of interest while being subject to a limited sampling budget.

For the results presented in this section, we use $I = 800$ sample points for NVIDIA-`matrix` and $I = 60$ for all else. These values are chosen using Figure 5(a) such that for each kernel, I is a low sample count but still achieves reasonable coverage. However, when actually using Iterative Starchart, we may not have Figure 5(a) for a given kernel. Instead, one can use Section 7.1's adaptive method to define I (e.g., the smallest sample count to reach a 30% overall error).

With I chosen, we then vary t from 5% to 25%. For a particular t value, as described in Algorithm 2, A additional samples along with the original I samples are used to build an Iterative Starchart tree. Figure 8(b) shows the value of $I + A$ for different

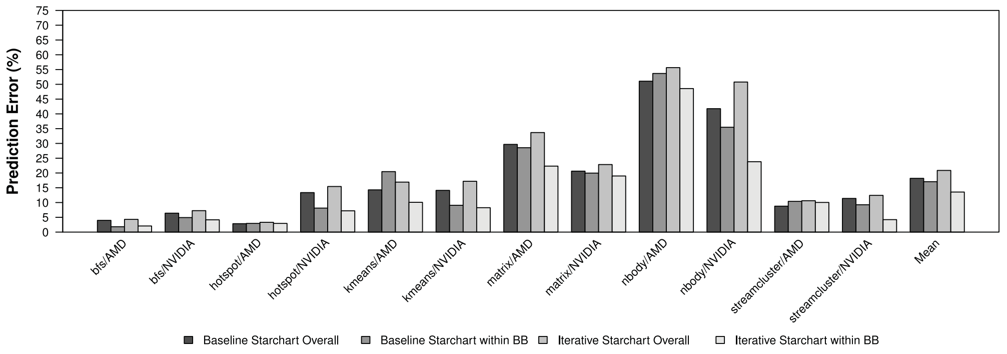


Fig. 9. Iterative Starchart has lower within-bounding box prediction error than Baseline Starchart for all applications. However, due to areas outside bounding boxes, overall error increases slightly.

t values. For experimental repeatability, A depends on t here because the additional samples are taken from a preselected, fixed-size sample repository. Thus, a higher t value generally increases the bounding box size and leads to more samples being taken from the repository. However, real users would not have a sample repository, and thus A would take a fixed value such as a fraction or multiple of I. Nevertheless, Figure 8(b) shows that for most kernels, at $t = 15$, A takes a value of 30 to 40. Notable exceptions include AMD-`hotspot` and NVIDIA-`matrix`, whose application characteristics make a 15% bounding box large enough to include most points from the sample repository.

Figure 8(a) shows the prediction error of Iterative Starchart trees trained using sample counts shown in Figure 8(b). The same validation method as in Section 7.1 is used, with 200 validation samples never used in the tree-forming process. Only prediction accuracy on those validation samples within the 15% bounding box (62 out of 200 samples on average across all applications) is shown in Figure 8(a). (The overall accuracy will be discussed later.) In the remainder of this section, we use $t = 15$ because it strikes a balance between error reduction and training set size increase.

Figure 9 compares Baseline Starchart and Iterative Starchart for overall accuracy (using all 200 validation samples) and for points within the bounding box (using on average 62 out of 200 validation samples). Both trees are trained using the same number of total samples shown in Figure 8(b). The first two bars for each application show Baseline Starchart's prediction accuracy (overall and within bounding box). The second two bars per application show corresponding results for Iterative Starchart. In general, for all applications, Iterative Starchart performs better (lower prediction error) for points from the regions of interest (within bounding boxes). In fact, compared to Baseline Starchart, Iterative Starchart has a 21% lower average prediction error for points within the bounding box. For performance tuning, this ability to use iterative sampling to quickly zoom in on the best-performing designs is very useful.

8. CASE STUDIES

As stressed in previous sections, the decision tree format used by Starchart can help to answer many subspace-based design questions. This section presents case studies that demonstrate how to formulate a few common design tuning problems as a series of decisions that can be answered using a Starchart tree.

8.1. 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

```
// A thread block has 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. 10. Our technique eases the process of selecting good values for the five parameters in the bfs kernel.

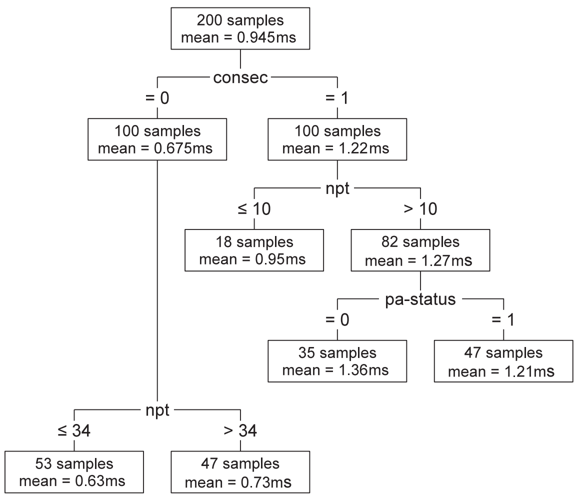


Fig. 11. The performance tree of bfs on AMD reveals that performance does not have strong dependence on block.

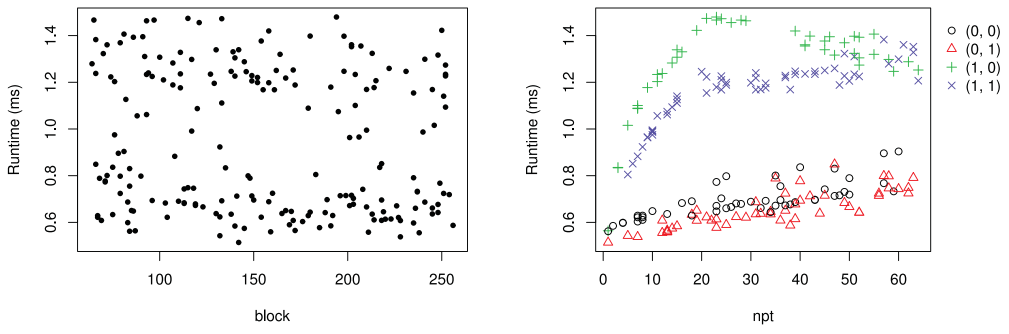
from potential optimization. For bfs, a breadth-first search graph algorithm, this case study shows how our approach differs substantially from prior work in its ability to reveal important and *local* parameter interactions to efficiently and accurately optimize parameter settings and prune uninteresting portions of the space.

As Table I shows, bfs on the AMD platform has five tunable parameters. Figure 10 shows 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—that is, 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 11 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 to hide memory latency for memory-bound kernels like bfs.
—*Reality:* Figure 11 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 12(a) 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 may 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 12(b) shows

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



(a) Performance versus `block` (b) (`consec`, `pa-status`) in parentheses

Fig. 12. Performance indeed depends on `npt`, and `consec` and `pa-status` further partition points into fairly separate groups.

performance versus varying `npt`, proving that caches are still helpful for small `npt` values.

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

—*Reality:* Contrary to the third hunch, the three parameters related to memory locality are not of equal importance. In particular, Figure 11'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 12(b) 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 overall design space, 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 [Nagasaka et al. 2010] rather than using input parameters or hardware configurations. Thus, prior clustering techniques might tie low cache miss counter values to high performance but could not directly guide designers toward the hardware or software *design decisions* to achieve this. In contrast, our approach's output *is tied directly* to controllable parameters. The tree partitions that we identify can lead directly to software optimizations or hardware design decisions.

8.2. 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 (1) optimize the power or performance of an application on each of many different platforms, (2) select which platform to run an application on according to some power or performance criteria, or (3) 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 HD 7970. Each sample is tagged with a binary parameter, `nvidia`, which takes 1 for NVIDIA runs and 0 otherwise. Our tool

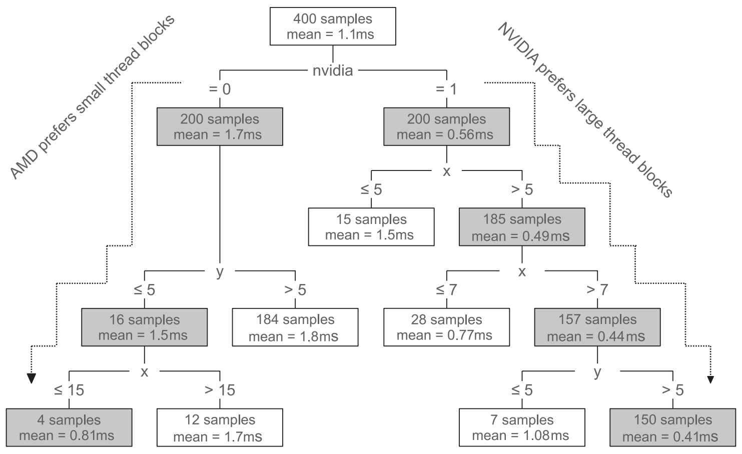


Fig. 13. The cross-platform tree of `hotspot` identifies particularly good designs for AMD and NVIDIA, respectively.

then analyzes all of these samples simultaneously, producing the regression tree in Figure 13.

In Figure 13, the highest-performing design subspaces for each platform are shaded in gray. Most notably, our partitioning approach helps a developer see that they are nonoverlapping 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 autotuner developer seeking to identify the best platform-specific settings could use Starchart to quickly guide design choices for the hardware encountered.

Cross-platform analysis becomes even more complex when trade-offs vary depending on other parameter choices. For example, Figure 13 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 the application's entire parameterized 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 to 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.

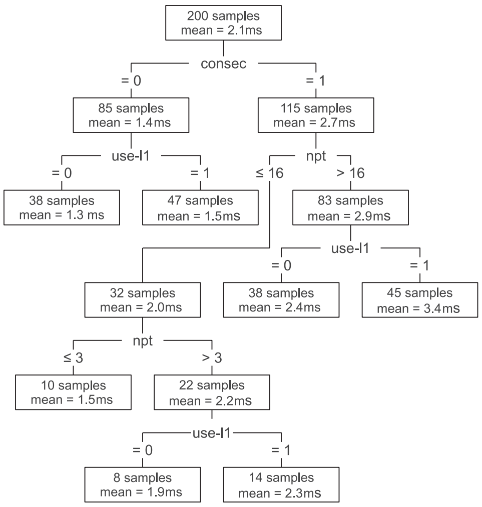


Fig. 14. The bfs performance tree on the NVIDIA platform shows that turning on caches has adverse effects on runtime.

8.3. Automating Cache Configuration

GPU developers also face hardware configuration decisions, such as the highly configurable caches on some recent NVIDIA GPUs. Users can configure on a per-instruction basis whether or not to use a cache and can trade off cache capacity against shared memory capacity. NVIDIA recommends that developers experimentally determine the best cache configuration [NVIDIA 2009]. In this case study, we show how our partitioning algorithm assists with such decisions.

Table II shows the NVIDIA cache configuration parameters to consider. Once these parameters are added to the design space, sampling and tree formation proceeds as discussed previously. Here, we create both power and performance trees.

Across our benchmark suite, caches have a widely varying impact. For example, in hotspot, the performance partition tree (not shown) shows clearly that caching has little influence on this program. This is because hotspot uses shared memory as its main workspace and relies little on global memory and, consequently, caches.

In contrast, Figure 14 shows when the cache parameter is influential for bfs and helps to clarify its effects. In the $\texttt{consec} = 1$ and $\texttt{npt} > 16$ portion of the tree, turning caches on ($\texttt{use-l1} = 1$) can hurt performance (3.4ms vs. 2.4ms) by almost 50%. This is because when caches are turned on, the NVIDIA hardware issues longer memory requests than when caches are not used. The longer memory requests created by caching increases bfs's already-high memory bandwidth demands and degrades performance. However, the use-l1 parameter is not visible in the subtree for $\texttt{npt} \leq 3$. The $\texttt{npt} \leq 3$ setting limits the impact that longer memory access requests have on the application's bandwidth demands.

Caching can also influence GPU power dissipation in similarly subtle ways. Later, Figure 16(a) shows the power tree for swap when the caching parameters are included. It shows that when $\texttt{tpp} \leq 8$, using a cache helps to reduce power consumption. However, the tree also shows that the impact of caching is unclear when $\texttt{tpp} > 8$.

This case study shows how our technique can determine when new hardware features should be used and how to configure them. Whether to use a hardware feature like caches depends heavily on the interplay between several of the software and

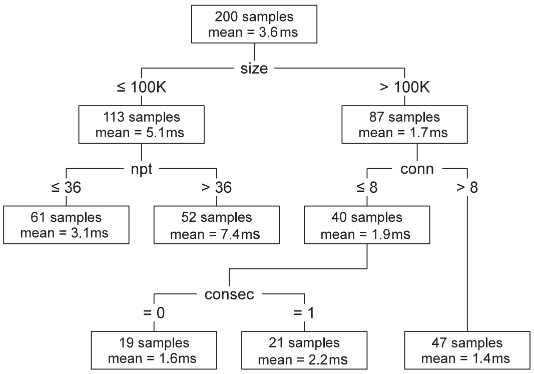


Fig. 15. The `bfs` performance tree on the AMD platform shows its input sensitivity.

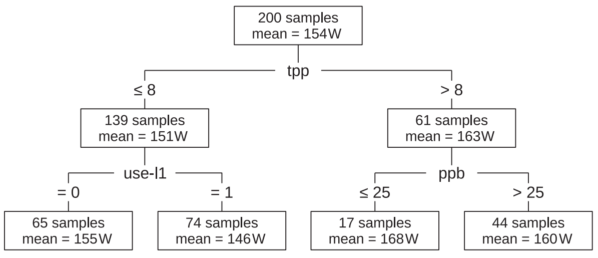
hardware parameters in the design space. Partition trees can help to visualize these issues.

8.4. Characterizing Program Input Sensitivity

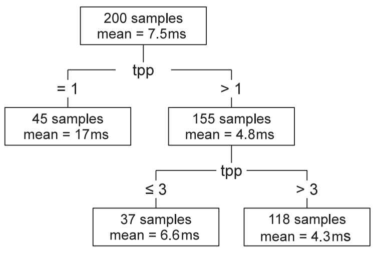
A program's runtime almost always scales with input data. For this reason, performance tuning generally requires the use of "typical" input data, and the resulting conclusions can be highly input dependent. Clearly, optimizations may vary in complex ways with input characteristics. Thus, this case study 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 respective input sizes—that is, we study computational throughput instead of absolute runtime.

For some applications, 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 $\times$400 to 3,200 $\times$3,200 elements) and 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 varying input sizes. It also indicates that optimizations based on parameter settings that *do* appear in regression trees are applicable across a wide range of inputs.

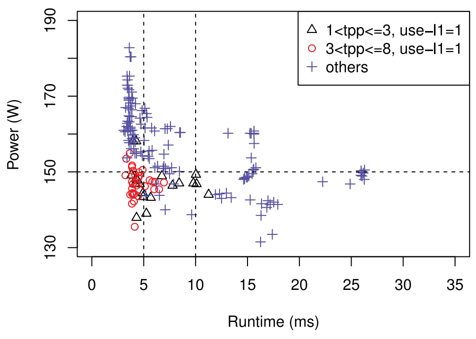
For `bfs`, we use a random input graph generator that takes in two parameters: (1) the number of nodes, `size`, which varies from 50K to 1M, and (2) the average number of edges a node connects to, `conn`, which varies from 1 to 32. The generated tree is shown in Figure 15 and leads to the following observations. 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



(a) Power tree



(b) Performance tree



(c) Power versus performance

Fig. 16. Power/performance trade-offs of `swap` in `kmeans` on the NVIDIA GPU.

subspaces helps to identify optimizations that are *conditionally* important for certain input sizes/characteristics.

8.5. Exploring Power/Performance Trade-Offs

GPU system and application performance optimization is increasingly subject to power constraints. However, few automated approaches have been proposed to tackle power-aware performance optimization. Most prior design space exploration studies [Lee and Brooks 2006; Jia et al. 2012; Hong and Kim 2009; Nagasaka et al. 2010; Chen et al. 2011] can only model one metric at a time. One study proposes modeling power and performance in conjunction [Hong and Kim 2010], but that study's 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 partitions are derived from original program parameters, users can immediately know how to set parameters to achieve the power/performance of the partitions that meet their goals.

Figure 16(c) shows power and performance of the 200 `kmeans` design samples on the NVIDIA platform. Ignoring 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 GPU power consumption to below 150W and meet a runtime target of 10ms (indicated by dotted lines), it is difficult to know which particular parameter settings would result in qualified designs.

Starchart trees help to answer these questions. Figure 16(a) shows the power tree of `kmeans`. Clearly, the design subspace with `tpp` $\leq$ 8 and `use-l1` $=$ 1 has a power consumption generally below 150W. This is due to less intense thread activities

($\texttt{tpp} \leq 8$) and the use of caches ($\texttt{use-l1} = 1$). Likewise, `kmeans`'s performance tree (Figure 16(b)) shows that the design subspace with $\texttt{tpp} > 1$ can generally meet the performance goal due to increased interthread parallelism. By intersecting these two sets and plotting them using different symbols (triangles and circles in Figure 16(c)), we see that 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 and reduced power budget. In these situations, our method can help to adaptively determine the necessary program state transitions. For example, assume that the program runtime target becomes more stringent from 10ms to 5ms, without changing the power budget. Users can look at Figure 16(b) 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 16(c) shows that designs with $\texttt{tpp} > 3$ (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 47W (or 26% of total power) with less than 10% performance slowdown.

Finally, 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`, the effective use of L1 caches can lower power usage regardless of whether runtime is improved. Additionally, the same program over the same range of optimizations 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 earlier 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 can be useful to many users.

9. CONCLUSION

This article 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. An optional iterative approach can further increase accuracy for areas of interest through guided design sampling. Finally, detailed case studies are presented to show how this algorithm can help to solve application tuning problems such as pruning program design parameters, optimizing program performance across platforms, automating cache configuration, characterizing program input sensitivity, and doing power-aware performance optimization.

REFERENCES

AMD. 2012. APP Profiler. Retrieved March 27, 2015, from http://developer.amd.com/tools-and-sdks/archive/amd-app-profiler.

Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffery Bosboom, Una-May O'Reilly, and Saman Amarasinghe. 2014. OpenTuner: An extensible framework for program autotuning. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation Techniques*. 303–316.

James Bergstra, Nicolas Pinto, and David Cox. 2012. Machine learning for predictive auto-tuning with boosted regression trees. In *Proceedings of Innovative Parallel Computing*. 1–9.

Leo Breiman, Jerome Friedman, Charles J. Stone, and Richard A. Olshen. 1984. *Classification and Regression Trees*. Chapman and Hall/CRC.

Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the IEEE International Symposium on Workload Characterization*. 44–54.

Jianmin Chen, Bin Li, Ying Zhang, Lu Peng, and Jih-Kwon Peir. 2011. Tree structured analysis on GPU power study. In *Proceedings of the IEEE 29th International Conference on Computer Design*. 57–64.

Jee W. Choi, Amik Singh, and Richard W. Vuduc. 2010. Model-driven autotuning of sparse matrix-vector multiply on GPUs. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 115–126.

Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick. 2008. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage, and Analysis*. 1–12.

Yuri Dotsenko, Sara S. Baghsorkhi, Brandon Lloyd, and Naga K. Govindaraju. 2011. Auto-tuning of fast Fourier transform on graphics processors. In *Proceedings of the 16th Symposium on Principles and Practice of Parallel Programming*. 257–266.

Walter D. Fisher. 1958. On grouping for maximum homogeneity. *Journal of the American Statistical Association* 53, 789–798.

David Freedman, Robert Pisani, and Roger Purves. 2007. *Statistics* (4th ed.). W. W. Norton and Company.

Grigori Fursin and Olivier Temam. 2010. Collective optimization: A practical collaborative approach. *ACM Transactions on Architecture and Code Optimization* 7, 4, 20:1–20:29.

Archana Ganapathi, Kaushik Datta, Armando Fox, and David Patterson. 2009. A case for machine learning to optimize multicore performance. In *Proceedings of the 1st USENIX Conference on Hot Topics in Parallelism*. 1.

Sunpyo Hong and Hyesoon Kim. 2009. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In *Proceedings of the 36th International Symposium on Computer Architecture*. 152–163.

Sunpyo Hong and Hyesoon Kim. 2010. An integrated GPU power and performance model. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*. 280–289.

Laurent Hyafil and Ronald L. Rivest. 1976. Constructing optimal binary decision trees is NP-complete. *Information Processing Letters* 5, 1, 15–17.

Wenhao Jia, Kelly A. Shaw, and Margaret Martonosi. 2012. Stargazer: Automated regression-based GPU design space exploration. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*. 2–13.

Wenhao Jia, Kelly A. Shaw, and Margaret Martonosi. 2013. Starchart: Hardware and software optimization using recursive partitioning regression trees. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*. 257–267.

P. J. Joseph, Kapil Vaswani, and Matthew J. Thazhuthaveetil. 2006. Construction and use of linear regression models for processor performance analysis. In *Proceedings of the 12th International Symposium on High Performance Computer Architecture*. 99–108.

Michael H. Kutner, Christopher J. Nachtsheim, John Neter, and William Li. 2005. *Applied Linear Regression Models* (5th ed.). McGraw-Hill.

Benjamin C. Lee and David M. Brooks. 2006. Accurate and efficient regression modeling for microarchitectural performance and power prediction. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*. 185–194.

Yinan Li, Jack Dongarra, and Stanimire Tomov. 2009. A note on auto-tuning GEMM for GPUs. In *Proceedings of the 9th International Conference on Computational Science*. 884–892.

Wei-Yin Loh. 2008. Classification and regression tree methods. In *Encyclopedia of Statistics in Quality and Reliability*, F. Ruggeri, R. S. Kenett, and F. W. Faltin (Eds.). Wiley, 315–323.

Alberto Magni, Christophe Dubach, and Michael F. P. O'Boyle. 2013. A large-scale cross-architecture evaluation of thread-coarsening. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage, and Analysis*. Article No. 11.

Jake Moilanen and Peter Williams. 2005. Using genetic algorithms to automatically tune the kernel. In *Proceedings of the Linux Symposium.*

Sreerama Murthy and Steven Salzberg. 1995. Decision tree induction: How effective is the greedy heuristic? In *Proceedings of the 1st International Conference on Knowledge Discovery and Data Mining*. 222–227.

Hitoshi Nagasaka, Naoya Maruyama, Akira Nukada, Toshio Endo, and Satoshi Matsuoka. 2010. Statistical power modeling of GPU kernels using performance counters. In *Proceedings of the International Green Computing Conference*. 115–122.

NVIDIA. 2009. NVIDIA's Next Generation CUDA Compute Architecture: Fermi. Retrieved March 27, 2015, from http://www.nvidia.com/content/pdf/fermi_white_papers/nvidia_fermi_compute_architecture_whitpaper.pdf.

NVIDIA. 2011. Tuning CUDA Applications for Fermi. Retrieved March 27, 2015, from http://hpc.oit.uci.edu/nvidia-doc/sdk-cuda-doc/C/doc/Fermi_Tuning_Guide.pdf.

NVIDIA. 2012. NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110. Retrieved March 27, 2015, from http://www.nvidia.com/content/PDF/kepler/NVIDIA-kepler-GK110-Architecture-Whitepaper.pdf.

NVIDIA. 2014a. GPU Computing SDK. Available at https://developer.nvidia.com/gpu-computing-sdk.

NVIDIA. 2014b. NVIDIA Visual Profiler. Retrieved March 27, 2015, from https://developer.nvidia.com/nvidia-visual-profiler.

NVIDIA. 2014c. Tuning CUDA Applications for Kepler. Retrieved March 27, 2015, from http://docs.nvidia.com/cuda/kepler-tuning-guide/#axzz3Ve1bH0Vr.

R Core Team. 2013. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. http://www.R-project.org/.

Shane Ryoo, Christopher I. Rodrigues, Sam S. Stone, Sara S. Baghsorkhi, Sain-Zee Ueng, John A. Stratton, and Wen Mei W. Hwu. 2008. Program optimization space pruning for a multithreaded GPU. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. 195–204.

Burr Settles. 2010. *Active Learning Literature Survey*. Technical Report. University of Wisconsin–Madison.

Jaewoong Sim, Aniruddha Dasgupta, Hyesoon Kim, and Richard Vuduc. 2012. A performance analysis framework for identifying potential benefits in GPGPU applications. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 11–22.

Yuri Torres and Arturo Gonzales-Escribano. 2011. Understanding the impact of CUDA tuning techniques for Fermi. In *Proceedings of the International Conference on High Performance Computing and Simulation*. 631–639.

Spyridon Triantafyllis, Manish Vachharajani, Neil Vachharajani, and David I. August. 2003. Compiler optimization-space exploration. In *Proceedings of the International Symposium on Code Generation and Optimization*. 204–215.

Received August 2014; revised January 2015; accepted February 2015