Shrinking Sample Search Algorithm for Automatic Tuning of GPU Kernels

Xiang Li

Computer Science and Engineering
Ohio State University
Columbus OH 43210
Email: li.3880@osu.edu

Gagan Agrawal

Computer and Cyber Sciences
Augusta University
Augusta GA 30912
Email: gagrawal@augusta.edu

***Abstract*—Autotuning has been widely studied in high performance computing as a very effective mechanism for improving application performance. Such an approach has become particularly crucial for architectures like the modern GPUs, where obtaining the best performance involves a complex interaction between the architecture and the applications. Autotuning methods rely upon a *search strategy*, which is designed to search through the (potentially very large) space. A large number of search methods have been proposed in the past, and include both local and global strategies. We observe that on GPU applications, high performing configurations are likely to be spatially clustered. Based on this observation, we propose to apply a strategy we refer to as *shrinking sample*. This method searches in all areas of the entire space, looking for combinations of different parameter values, and without relying on random (initial) choices that may miss a part of the space. The efficacy and efficiency of this method has been tested against state-of-the-art local and global search algorithms on seven benchmark GPU kernels. Our experiments show that the shrinking-sample method can achieve around 99% percent of the performance from exhaustive search (on average) with orders of magnitude much less tuning time.**

I. Introduction

Graphics Processing Units (GPUs) are now an integral part of high performance computing (HPC), for a number of reasons like their cost-effectiveness and power-efficiency. Complexity of the codes run on such architectures have grown considerably because of the features of these systems. Specifically, for execution on GPUs, there are many tunable parameters like number of threads, thread block dimensions, unrolling factor, and others. Different values for each dimension may result in very different performance and furthermore, parameter settings are sensitive to both input programs and the underlying architecture. Certain parameters for program transformations such as unrolling and tiling can also have complex interactions with each other [26]. If the programmer is to choose these parameters on their own, they need to have a detailed knowledge of the specific architecture and insights into the performance characteristics. This can be extremely hard even for a single architecture, and in practice, needs to be repeated as new architectures emerge.

To free programmers from such tedious and time consuming manual tuning effort, various autotuning techniques have been developed. Autotuning broadly refers to an automated search for the combination of the tunable parameter values that yield the best performance. Autotuning software can be categorized as three groups [32]: (1) compiler-based autotuners [10], [38], [16], (2) library generators that incorporate autotuners such as FFTW [15], ATLAS [36], PhiPAC [8], SPIRAL [37] and OSKI [35]. (3) application-level autotuners [34], [24].

One common challenge for all autotuners is that the *search space*, i.e., the cartesian product of all possible values for each tuning parameter, is often too huge for an exhaustive search. Therefore, efficient search algorithms are required to examine a subset of possible parameter configurations to find (close to) optimal parameter configuration(s). There are two broad classes of searching algorithms: *model-free* and *model-based*. A model-based algorithm avoids the cost of running the code by predicting performance of a given parameter configuration [3]. Model-based methods can be further categorized into two groups: *analytical model* based algorithms and *empirical model* based algorithms. Analytical models use closed-form expressions for predictions of performance and often require great expertise in programming model, application details, and the target architecture. Moreover, such methods are often limited, especially on GPUs, where there is a complex interaction between the code, architecture, and runtime systems [3]. Empirical model based methods use a subset of parameter configurations to build a predictive model, using machine learning approaches [11], [19], [28], [31]. However, as codes and architectures become more complicated, more tuning parameters may be added to the applications. This implies very complex models, which are hard to train and/or less effective. On the other hand, model-free methods navigate the search space for high-performing configurations by using various search strategies that are independent of application and architecture. Overall, they can have better portability across architectures and do not require understanding of (or generating models for) an individual application.

In this paper, a model-free search algorithm for autotuning, *shrinking sample method*, is applied and evaluated. The application of this method is based on the hypothesis that *high performing configurations*, i.e. configurations that perform very close to optimal, tend to be spatially clustered in the space of parameter values. Suppose there are n parameters, our method ensures that search is conducted in all areas of the n-dimensional space – unlike

most other local methods, it does not use random sampling (of initial points). However, after an initial step, it works like a local method, but still looks at combination of parameter values (as opposed to optimizing each parameter independently). Thus, the cost associated with the method is similar to that of local methods. Our hypothesis allows us to search the space efficiently, and find a combination of parameters that are either optimal, or have performance very close to the optimal.

For evaluation, our method has been incorporated into a developed autotuner framework [33]. To examine the efficiency of this search algorithm, multiple GPU kernels with varying search space sizes have been used as benchmark including: median, bilateral, `stereo`, convolution, *GEMM*, raycast, and SpMV. The main parameters we focus on tuning include loop unrolling, tiling, block size in each dimension, and others. Based on experimental results, the shrinking sample search algorithm outperforms most of other search algorithms and shows robustness against different sizes and patterns of search spaces. The shrinking sample method achieves 97.25% ~ 99.97% performance of the brute-force search. At the same time, it takes only 1% ~ 10% of the corresponding exhaustive search time for all but one of the benchmarks (it takes 46% of the time for exhaustive search time for the *GEMM* kernel that has a very small search space size). In general, our method produces much better performing parameter combinations over local search methods, and is faster than most of the global search methods.

II. Related Work

In recent years, several autotuning frameworks have been built for GPU applications. Bruel *et al.* [9] implemented an auto-tuner for a CUDA compiler using the OpenTuner framework. Rasch *et al.* [27] demonstrated the efficacy of a generic directive based autotuning framework (ATF), which shows higher efficiency and flexibility compared with the state-of-the-art autotuning approaches OpenTuner [2] and CLTUne [25]. Kurzak *et al.* [23] described the implementation of a distributed autotuning engine for deploying large tuning sweeps of GPU kernels to large super computer/cluster installations. Bao *et al.* [6] presented a learning-based autotuning system to optimize the performance of big data analytics frameworks by generating enough samples for a better prediction model under certain time constraints. Werkhoven *et al.* [33] implemented a search optimizing GPU code auto-tuner for testing CUDA, OpenCL, and C kernels supporting many search optimization algorithms. Guerreiro *et al.* [17] proposed procedures to auto-tune concurrent GPU kernels to maximize the performance or minimize the energy consumption.

Various model-based search strategies have also been proposed for GPUs. Davidson *et al.* [1] demonstrate multiple techniques toward autotuning data parallel algorithms on GPU and establish a model based search method. This method identifies computational patterns between algorithms to prune the search space before tuning the algorithms' parameters. Feng *et al.* [14] proposed a sample-based GPU autotuning strategy that combines heuristic search with regression trees to prune the optimization space. Thomas *et al.* [13] used machine-learning based autotuning to address the portability issue of OpenCL.

Generic autotuners that use the model-free search algorithms can be further categorized as two groups: those using *local* and *global* search algorithms, respectively. Certain global algorithms can theoretically guarantee to find the globally optimum configuration [3], though, in reality, such algorithms will terminate when a user defined stopping criterion is met. Examples include genetic algorithm [18], particle swarm optimization [20], and simulated annealing [21]. In contrast, local search algorithms (such as window search [22], orthogonal search [30] and pattern-based search method [26]) aim to find an improving configuration by looking at values of each parameters in the neighborhood of the current value [3]. These algorithms will stop when we cannot find a neighboring configuration that improves the performance.

Extensive experimental studies on various search heuristics have been performed. Seymour *et al.* [30] examined global search methods such as random search, genetic algorithm, simulated annealing, particle swarm optimization (PSO), and local search methods including Nelder–Mead and orthogonal search. The orthogonal search is shown to suffer if there is a bad ordering of tuning parameters. Billings *et al.* [7] proposed a new adaptive orthogonal search (AOS) algorithm for model subset selection and non-linear system identification. The new AOS scheme provides an efficient tool for model term selection, model size determination, and parameter estimation [7]. PSO tends to have advantages on searching the search areas near the bounds of certain tuning parameters [30]. Random search method turned out to be robust and can compete with most of the search methods for the applications that have a large number of high performing tuning parameter configurations. Other local search methods such as pyramid search and window search have been investigated in the previous work by Kisuki *et al.* [22]. Another local algorithm, pattern-based direct search, has also been studied [26] and was shown to compete with the random search methods for multiple applications. Balaprakash *et al.* [5] made efficiency comparison between the global and local algorithms. The work showed that when the computation time is crucial, local algorithm is more efficient especially given good initial configurations. However, poor initial configurations can significantly retard both local and global search algorithms [5].

III. Motivation

One issue for autotuning is that the effectiveness of a search algorithm can be influenced by the relative performance of different *configurations* (i.e. combination of parameter values) in the search space. One question is whether certain properties can be exploited to facilitate a faster and yet effective search. One example of exploiting such a property is density of *high performing* configurations, which are the configurations that perform close to the highest performing configuration. Certain algorithms, such as random search, require or benefit from a high density of high-performing configurations [30]. However, the distribution of configurations depends on the application and more specifically, high performing configuration density varies for each application, and accordingly, random search can be limited on many applications.

To further examine what properties of search space can be used, we experimented with two applications on

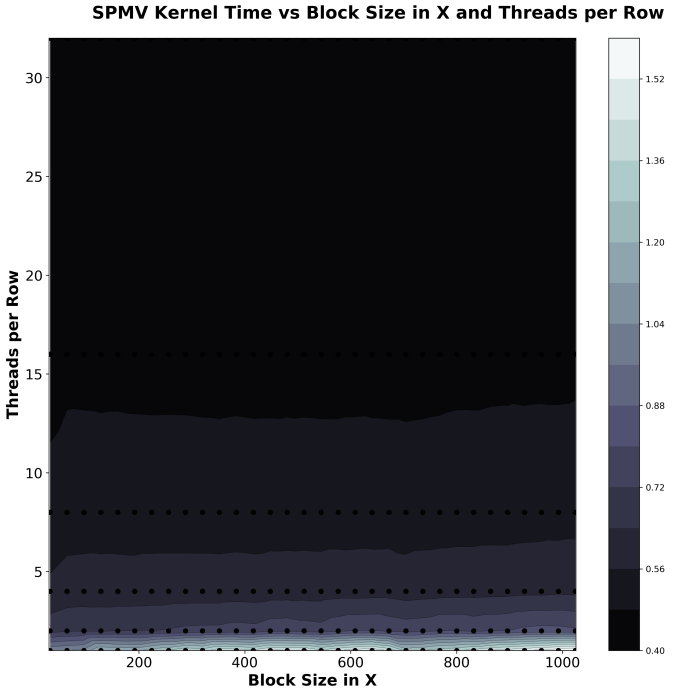

Fig. 1: Contour plots of SpMV kernel running time versus block size in x dimension and number of threads per row

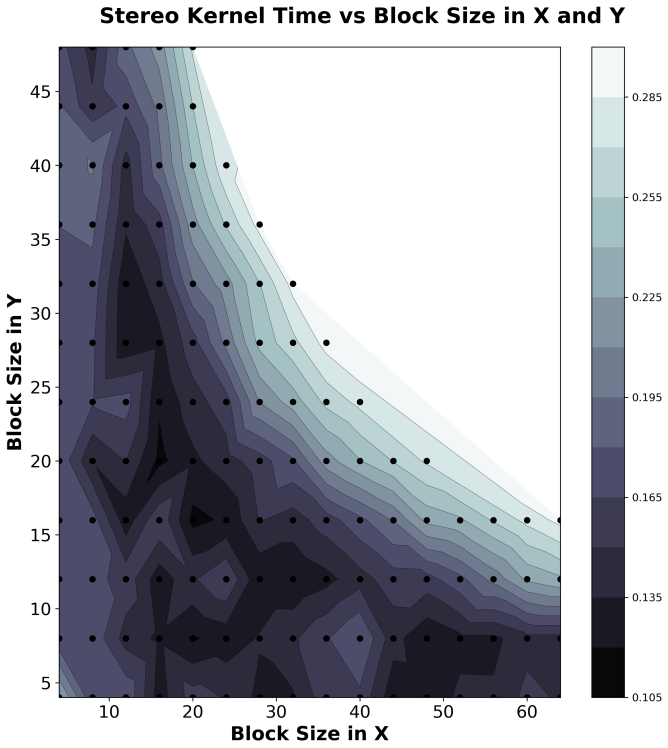

Fig. 2: Contour plots of `stereo` kernel running time versus block size in x and y dimension

GPU. Figures 1 and 2 show the relationship between the parameter values and the kernel performance of two applications: SpMV and `stereo`. The color bar shows the scale of the kernel running time and the dots shows the data point in the 2D-grid. Data points are located only in a part of the grid in Figure 2 due to the constraints between the tuning parameters (some of the grid points do not yield a valid parameter configuration).

The performance trends for SpMV show a high density of high performing points, making it easy for even the simplest of search methods to find a version with best possible performance. However, relationship between performance and parameters for `stereo` is more complex because of the existence of multiple local minima. It is well known that many search methods, especially local search methods, can suffer from the existence of the local minima [4]. Methods that are particularly vulnerable are the ones based on choosing an initial (set of) points randomly, and searching only around these points.

Overall, from results of these two applications, we notice the following. The high performing configurations tend to be spatially clustered in the space. At the same time, we find that the density of high performing configurations is not necessarily high. These observations help drive the design of our algorithm. We want to ensure that all areas within the search space are considered in the initial step, but subsequently, we focus on fine-tuning our choice within a restricted area.

IV. Shrinking Sample Search Algorithm

We introduce the following terms to describe the algorithm: n denotes the number of tuning parameters, i.e, we have an n-dimensional search space, where each dimension represents a tuning parameter such as block size or unrolling factor. $P = (P_1, P_2, \ldots, P_n)$ denotes a point inside the search space where P_i denotes the value of i_{th} tuning parameter at this configuration. Each parameter has a set of possible values which are determined by the application and the user input. f$(P_1, P_2, \ldots, P_n)$ denotes the execution time for the program with tuning parameters $(P_1, P_2, \ldots, P_n)$.

The goal of the search algorithm is to find (close to) optimum configuration point $P_{best} = (P_1, P_2, \ldots, P_n)$ such that f$(P_1, P_2, \ldots, P_n)$ is minimized. Two hyper-parameters are defined for the shrinking sample algorithm: number of partitions (k) and lower threshold for further division (V_{TH}). We first divide all possible values for each tuning parameter into k equal-sized partitions of consecutive values. The median of each partition is chosen as the representative value of the partition. Instead of going through each possible value of a tuning parameter, we save search time by only considering combinations of medians of each partition for each parameter. Our hypothesis is that such combination of medians can well represent the overall performance of its corresponding *section* in the search space. We have k^n points to search in the first step of the algorithm, which is likely much smaller than the entire search space. At the same time, unlike many other (local) algorithms, we are considering points from all parts of the entire search space.

Next, we execute the kernel with each configuration in our reduced search space. The configuration found to have the best performance represents the combination of *optimal partition* for each of the parameters. Our hypoth-

esis is that the region surrounding the best configuration from the reduced or *shrunk* search space is very likely to have the best configuration, or a configuration with similar performance to that of the best configuration. We will start next iteration of the search process by dividing the obtained optimal partition for each tuning parameter into k sections. This allows for a detailed search within the section we have identified. To avoid the issues caused by local minima, each time we combine medians for different tuning parameters from different sections, creating a larger set of candidate configurations as compared to methods that vary only one parameter at a time. The division of each partition and recombination of medians will stop when every partition size is smaller than V_{TH}. The final optimal section will have a size small enough for an exhaustive search to find the optimal parameter configuration in a reasonable time period. Though the hierarchical grid search method [29] also performs a "coarse-to-fine" tuning, it relies on the random search for specific number of combinations to find promising areas during each iteration. However, the randomly generated combinations cannot guarantee that all ranges of each parameter are checked and can miss the optimal configurations more easily.

Algorithm 1 Shrinking-sample algorithm

procedure SHRINK( )
  Divide values of each parameter into k sections:
  S^m=($[S_{11}, \ldots, S_{1k}], [S_{21}, \ldots, S_{2k}], \ldots, [S_{n1}, \ldots, S_{nk}]$)
  Construct a new search space from each section's median
  $P^m = ([P_{11}, \ldots, P_{1k}], [P_{21}, \ldots, P_{2k}], \ldots, [P_{n1}, \ldots, P_{nk}])$
  while Stopping Criteria Not Met **do**
    Execute kernel on the new search space
    Identify optimum configuration:
      $P^m_{best} = (P_{1j^1}, \ldots, P_{nj^n})$
    Find optimum section configuration:
      $S^m_{best} = (S_{1j^1}, \ldots, S_{nj^n})$
    $P^{m+1} \leftarrow P^m_{best}$
    $S^{m+1} \leftarrow S^m_{best}$
  Exhaustive search through all configurations in S^m_{best}

An illustrative example run of shrinking sample algorithm on SpMV kernel is shown in Figure 3. Three tuning parameters are introduced for SpMV kernel: block size in x dimension with 32 values, number of threads per row with 4 values and a boolean parameter read with 2 possible values. For simplicity, only two iterations are presented with the hyper-parameters taken as: k=2, and V_{TH}=1. For each iteration, the possible values of each parameters are divided into two partitions as indicated by the red dash line and the median value of each partition is shaded. The value circled by the blue rectangle indicates the extracted optimal value and the chosen partition will be further divided in the next iteration. We next describe the main steps of **Algorithm 1** below, using SpMV as a concrete example (only the first iteration will be shown for SpMV).

Step 1: We divide all possible values for each tuning parameter into k distinct sections.

S^m = ($[S_{11}, \ldots, S_{1k}], [S_{21}, \ldots, S_{2k}], \ldots, [S_{n1}, \ldots, S_{nk}]$)

where each S_{ij} indicates the j_{th} section of the i_{th} tuning parameter. For the SpMV kernel, three tuning parameters with multiple possible values are divided into two halves in the first iteration. The values for tuning parameters will

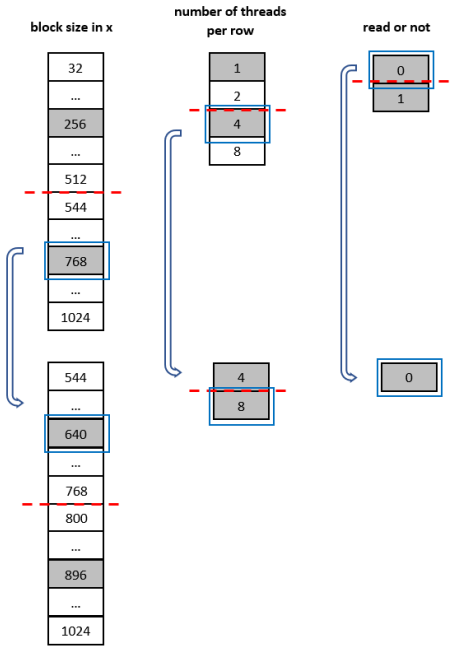

Fig. 3: Flow chart of the shrinking algorithm between different iterations on the SpMV kernel.

be listed in the following order: block size in x, number of threads per row, and read or not. So, we get

S^1 = [(32, ..., 512), (544, ..., 1024)], [(1, 2), (4, 8)], [0, 1]

Step 2: We choose the median of each section to construct a small search space.

$P^m = [P_{11}, \ldots, P_{1k}], [P_{21}, \ldots, P_{2k}], \ldots, [P_{n1}, \ldots, P_{nk}]$

where each P_{ij} implies the median of j_{th} section of i_{th} tuning parameter, i.e., P_{ij} denotes the median of S_{ij}. For the SpMV kernel, the medians of each partition are listed as P^1 = [256, 768], [1, 4], [0, 1].

Step 3: Execute the kernel on the constructed search space and identify the tuning configuration with the best performance: $P^m_{best} = (P_{1j^1}, \ldots, P_{nj^n})$ and the corresponding optimum section configuration: S^m_{best} = (S_{1j^1}, ..., S_{nj^n}). The corresponding execution time f(P^m_{best}) will be recorded as the best performance so far.

For the SpMV kernel, we have two median values for each parameter and we combine them to construct a search space with a size of 2×2×2, which is much smaller than the original search space. The best value for each parameters are circled in Figure 3 and is P^1_{best} = (768, 4, 0).

Step 4: We check the current optimum section configuration P^m_{best} and extract the corresponding partition where the median value belongs. Now, if the size of any section is greater than the lower threshold value: V_{TH}, repeat steps 1-3 by using current section configuration as the starting section configuration. However, if none of the section in the current section configuration has a larger size than V_{TH}, move to **Step 5**.

For the SpMV kernel, we extract the corresponding section based on P^m_{best}. For example, 768 belongs to the section (544, 576, ...,1024). In this way, we obtain all the optimal partitions for each tuning parameter:

S^1_{best} = [(544, 576, ...,1024)], [(4, 8)], [0]

Since the parameter read or not hits the threshold V_{TH}=1, we will continue to divide partitions of the other two tuning parameters in S^1_{best} in the next iteration.
Step 5: After we obtain a section configuration $S^m_{best} = (S_{1j^1}, \ldots, S_{nj^n})$, we can now perform a brute force search on all the tuning parameter configurations covered by this S^m_{best}.

For the SpMV kernel, suppose we reach the optimal partition at m iteration: $S^m_{best} = [576], [4], [0]$. We then perform the kernel calculation over S^m_{best}, which only includes one parameter configuration since $V_{TH} = 1$. The resulting performance will be the optimal performance. For the case where $V_{TH} > 1$, we will run kernels for multiple configurations covered by S^m_{best} and the one with the best performance will be selected.

A. Analytical Comparison with Local Search Methods

We now analytically compare our method against popular search methods, focusing on number of configurations one will need to experiment with. Our comparison is against orthogonal logarithmic search, window search, and pattern based direct search method.

In orthogonal search, one dimension is optimized while keeping the other dimensions constant. Then, each successive dimension is optimized while retaining the best values for the preceding dimensions. In our implementation, we first generate a random subset of the search space with the size: $rand_s$, and choose one configuration randomly as the starting point.

In window search, two hyper-parameters are defined: the sample size (S) and the shrinking fraction (f). Initially, the window is defined as the entire space. First, we randomly generate sample of size (S) within the window and execute the kernel to select the configuration with the best performance. Then we generate a shrinking window size around this selected configuration and do the random sampling again. Each time the windows size will shrink by a factor (f) until the window size is decreased below the sample size (S).

In Pattern-based direct search method, hyper-parameters are random sample size ($rand_s$), step size (S) and shrinking fraction of step size (f). In our implementation, we first generate a random sample with a size of $rand_s$, from which we will randomly select one configuration as the starting point. This algorithm is described as follows: **Step 1**: Starting from the base configuration $P^0 = (P_1, P_2, \ldots, P_n)$, we first record its running time as: $f_{min} = f_{P^0}$. Then we change the value of each parameter by increasing and decreasing by the step size S to generate two more configurations. For example, for i_{th} parameter we have two more configurations: $P^1 = (P_1, P_2, \ldots, P_i + S, \ldots, P_n)$ and $P^2 = (P_1, P_2, \ldots, P_i - S, \ldots, P_n)$. The configuration with the best performance will be selected. Once all the tuning parameters have been checked, we move to **Step 2**. **Step 2**: After we simulate the newly obtained configuration, if the time is less than that of the base configuration we move to **Step 1** otherwise move to **Step 3**. **Step 3**: Reduce the step size. If we reach the minimum step size (default to be 1) then we return the optimum configuration. Otherwise, we go back to **Step 1** with the reduced step size.

The evaluation of the search space size is listed in Table I. For the shrinking sample method, v_{max} is maximum number of possible values for all the tuning parameters. A compromise should be made for the value of k since a larger value of k will yield a greater size of subsets for each iteration but smaller number of iterations. Similarly, a larger V_{TH} may stop the division of a certain partition earlier but yield greater search space size for the final exhaustive search step with a time cost $O((V_{TH})^n)$. The orthogonal search method has the time complexity linear to the number of tuning parameters. However, the kernel performance depends on the random initialization. For the window search method, a larger S and a smaller fraction value f will make the algorithm stop early. However, a smaller f also makes it easier to miss the optimal configuration. In pattern-based direct search methods, a smaller step size S and a smaller fraction f will make the algorithm converge faster but make the execution more likely to miss the optimum.

TABLE I: Search Space Size of Local Search Method

Local Search Method	Hyper-parameters	Search Space Size
Shrinking Sample	k, V_{TH}	$O(k^n \cdot log_k^{v_{max}} + (V_{TH})^n)$
Orthogonal Logarithmic	$rand_s$	$O(n + rand_s)$
Window Search	S, f	$O(S \cdot log_f^{\frac{S}{N}})$
Pattern-based direct Search	$rand_s$, S, f	$O(3^n \cdot log_f^{\frac{1}{S}} + rand_s)$

V. Experimental Results and Discussion

To demonstrate the effectiveness of the shrinking-sample algorithm, seven GPU kernels have been used as the benchmarks, each with input data as listed in Table II and corresponding tuning parameters in Table III. *GEMM*, *convolution* and *SpMV* kernels are from the kernel tuner framework [33] and all the other kernels are the CUDA version of the four benchmarks used in another machine learning based auto tuning framework [13], [12]. The search space size for all these kernels varies from 384 to 147,456. To illustrate the efficiency of this searching algorithm, we perform the shrinking-sample method along with other search algorithms.

TABLE II: Input data for benchmarking GPU kernels

GPU Kernel	Input data dimension	Pre-setting
bilateral	image ($256 \times 256 \times 64$)	filter ($5 \times 5 \times 3$)
convolution	image ($4096 \times 4096 \times 64$)	filter (17×17)
GEMM	matrices (4096×4096)	
median	image (2048×2048)	filter (5×5)
raycast	image (512×512)	
SpMV	sparse matrix ($2^{17} \times 2^{16}$)	non-zero fraction 10^{-3}
stereo	image 256×256	

A. Summary of Different Global Search Methods

Four global search schemes (PSO, genetic, simulated annealing, and random sampling) have been chosen for comparison. We used the version of those methods implemented by Werkhoven in his framework [33]. Moreover, we also implemented three local search methods (orthogonal search, window search, pattern-based direct search) within this framework using CUDA.
Particle Swarm Optimization [18] originates from the simulation of flocking or swarming patterns of birds. PSO

TABLE III: Tuning Parameters for GPU kernels

GPU Kernel Tuning Parameters	Value
bilateral	
`block size in x`	$4 \cdot i \quad (i = 1, 2, \ldots, 16)$
`block size in y`	$4 \cdot i \quad (i = 1, 2, \ldots, 16)$
`block size in z`	$2^i \quad (i = 1, 2, 3)$
`Output pixels per thread in x,y,z`	$1, 2$ for x, y, z
`Use local memory`	$0, 1$
convolution	
`block size in x`	$16 \cdot i \quad (i = 1, 2, \ldots, 8)$
`block size in y`	$2^i \quad (i = 0, 1, \ldots, 6)$
`tile size in x, y`	$1, 2, \ldots, 9$ for x, y
`Add padding to image`	$0, 1$
GEMM	
`block size in x`	$16 \cdot 2^i \quad (i = 0, 1, 2\ )$
`block size in y`	$2^i \quad (i = 0, 1, \ldots, 5)$
`tile size in x, y`	$2^i \quad (i = 0, 1, 2, 3)$ for x, y
median	
`block size in x, y`	$4 \cdot i \quad (i = 1, 2, \ldots, 16)$ for x, y
`Output pixels per thread in x, y`	$1, 2$ for x, y
`Use local memory`	$0, 1$
`Use sorting(0) or histogram (1)`	$0, 1$
raycast	
`block size in x, y`	$2^i \quad (i = 0, 1, \ldots, 6)$ for x, y
`Output pixels per thread in x, y`	$1, 2, 3$ for x, y
`Unroll factor for ray traversal loop`	$1, 2, \ldots, 16$
SpMV	
`block size in x dimension`	$32 \cdot i \quad (i = 1, 2, \ldots, 32)$
`Number of threads per row`	$2^i \quad (i = 0, 1, \ldots, 5\ )$
`Read only or not`	$0, 1$
stereo	
`block size in dimension x, y`	$4 \cdot i \quad (i = 1, 2, \ldots, 16\ )$ for x, y
`Output pixels per thread in x, y`	$1, 2$ for x,y
`Unroll factor (disparity loop)`	$1, 2, 4, 8$
`Unroll factor (difference loop x, y)`	$1, 2, 4$ for x, y
`Use local memory for left/right image`	$0, 1$ for left/right image

generates a group of particles inside the hyperspace constructed by the tuning parameters. Each particle has a position and velocity flying through the hyperspace while keeping a memory of the optimal position it has visited. The particles are simultaneously drawn towards the global and local optimal locations based on the relative strength of attractions. In our case, the initial particles are generated randomly at different locations in the hyperspace. At each iteration, the performance of every particle is evaluated and velocity are updated correspondingly based on the strength calculation formula [18]. The entire procedure terminates when the number of iterations has reached a specific threshold.

Genetic Algorithm [20] is based on the evolutionary process in biology with the idea that the fittest individual will survive. In the tuning process, each parameter configuration is taken as a gene and the fittest gene in population will be the optimal configuration which gives the best kernel performance. In our case, each gene will be represented by an array of values for each tuning parameter. The initial population is generated randomly and successive generations are produced by following procedures: 1) *crossover*: merge two genes to generate an offspring gene. Two parameter arrays will be split at a random position and concatenated to form a new configuration, 2) *mutation*: produce a mutated version of a gene. With low probability, we choose one configuration and alter one of the values, and 3) *selection*: pick the fittest gene. We perform the kernels on each generation and choose the configurations with the best performance while excluding the poor-performing members. In our case, the algorithm will terminate when the number of generations hits a specific threshold.

Simulated Annealing [21] is based on the simulation of annealing process of metals during which the internal structure of metals will change when undergoing a series of cooling and heating cycles. In the simulation, a temperature setting is needed such that the system tends to mutate the state when the temperature is high and converge to a steady state when the temperature is low. In the tuning process, the parameter configuration will be likely to change at high temperature and start to converge on optimal points as the temperature is decreasing. Multiple configurable parameters can affect the performance of this algorithm: 1) annealing schedule determines how the temperature is adjusted 2) acceptance probability checks the probability of acceptance of new configuration, 3) neighbor selection function chooses the next candidate configurations.

Random Sampling: A subset of the entire search space will be selected completely at random. The kernel will be executed on this subset to choose the best configuration. The size of the random chosen subset will be determined by the user.

B. Experimental Results

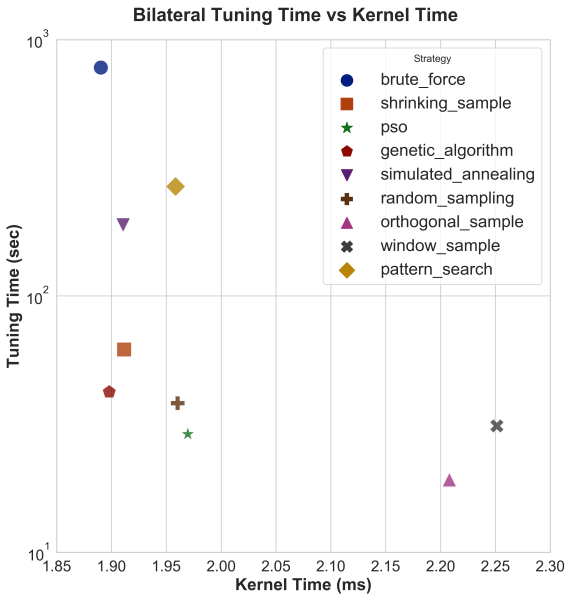

Fig. 4: The tuning time to find the best performing configuration versus the performance of Bilateral kernel, y-axis using the log scale

The experiments were performed both on a P-100 GPU and on a V-100 GPU using the auto-tuner framework *kernel tuner 0.2.0* [33]. We obtained consistent results from both architectures and only the results from P-100 are presented in this section. Each data point we present here represents the average of 14 runs of each search strategy on one specific tuning parameter configuration. The performance of each algorithm is measured with different values of their corresponding hyper-parameters and the

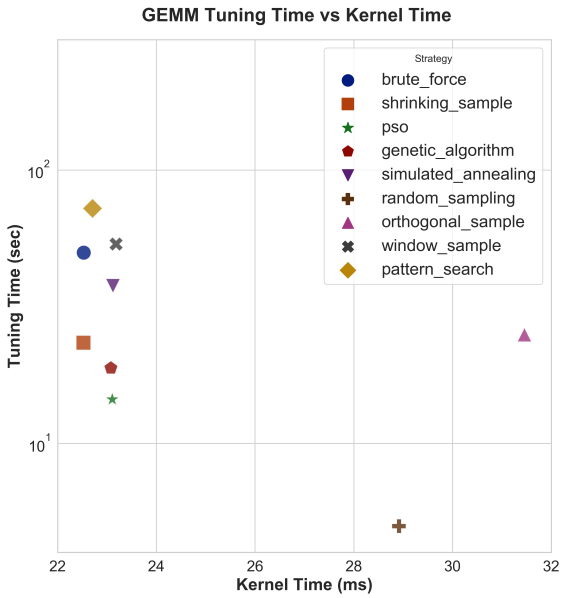

Fig. 5: The tuning time to find the best performing configuration versus the performance of *GEMM* kernel, y-axis using the log scale

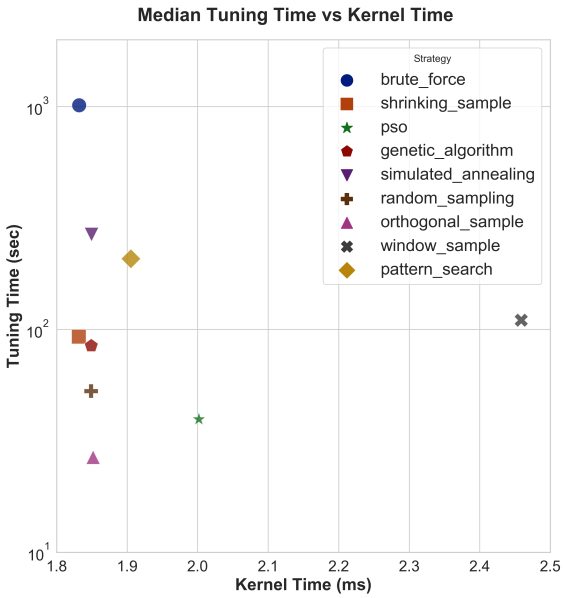

Fig. 7: The tuning time to find the best performing configuration versus the performance of Median kernel, y-axis using the log scale

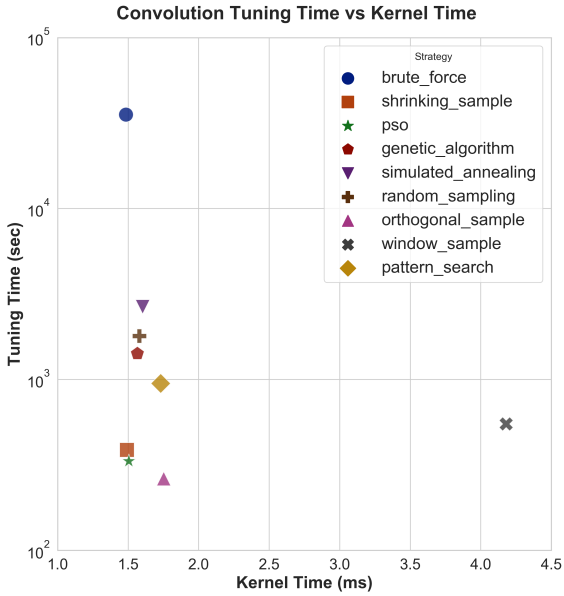

Fig. 6: The tuning time to find the best performing configuration versus the performance of Convolution kernel, y-axis using the log scale

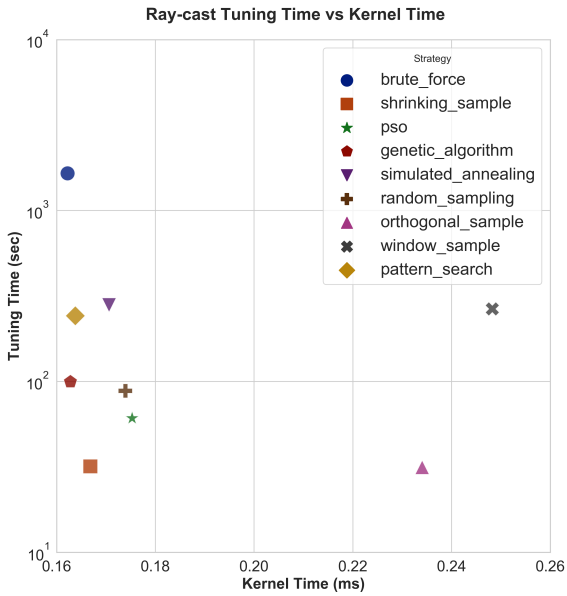

Fig. 8: The tuning time to find the best performing configuration versus the performance of Raycast kernel, y-axis using the log scale

best results are displayed in Figure 4-9. Exhaustive search is performed for each kernel to provide the optimal kernel running time as the baseline.

In Figure 4, the shrinking-sample method creates a faster version over all other search methods except the genetic algorithm. Pattern-based search has a kernel performance competing with that of random sampling method but the tuning time is higher. For *GEMM* kernel in Figure 5, the search space is relatively small and search methods like pattern-based search and window sampling have greater tuning time than exhaustive search due to the overhead in their tuning procedures. Shrinking-sample

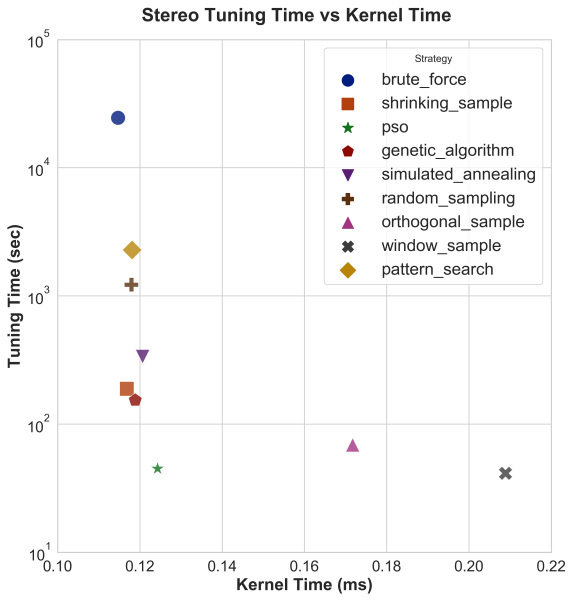

Fig. 9: The tuning time to find the best performing configuration versus the performance of Stereo kernel, y-axis using the log scale

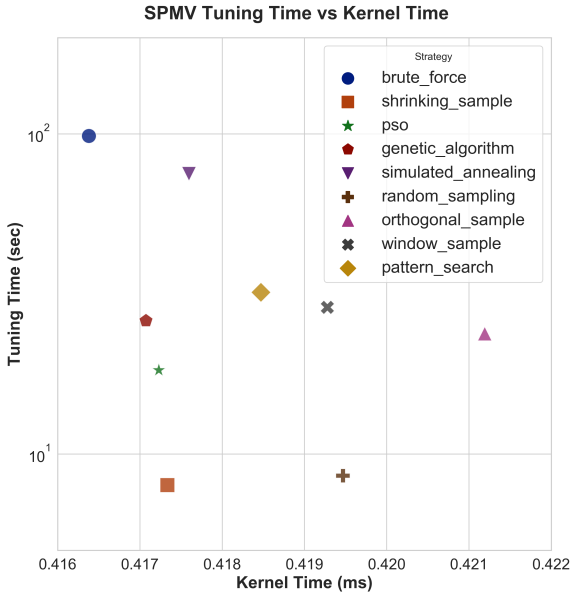

Fig. 10: The tuning time to find the best performing configuration versus the performance of SpMV kernel, y-axis using the log scale

provides the closest kernel performance to the exhaustive search though other methods such as genetic algorithm and PSO have lower tuning time.

For convolution kernel in Figure 6, shrinking-sample and PSO search methods take similar tuning time and provide competing performance. For median kernel in Figure 7, shrinking-sample provides the best performance. However, genetic algorithm, random sampling and orthogonal search also provide good kernel performance with less tuning time. For raycast kernel in Figure 8, shrinking-sample yields good kernel performance with the least tuning time though pattern-based search and genetic algorithm provide better kernel performance. In Figure 9, shrinking-sample method provides the best performance for `stereo` kernel. PSO used less tuning time and lost some kernel performance. As shown in Figure 10, all the search algorithms have similar kernel performance. The shrinking sample gained good kernel performance with the least tuning time.

Global search methods(pso, genetic algorithm, simulated annealing and random sampling) provide stable methods across seven benchmarks. Generally, shrinking-sample method outperforms most of the search strategies, however, another global search method can sometimes provide the same or even better performance than shrinking-sample (such as genetic algorithm in bilateral and pso in convolution). For the kernels with tuning parameters having a large number of possible values such as raycast and convolution, shrinking-sample can save much tuning time. The performance of local search methods (orthogonal logarithmic, window search and pattern-based search) can fluctuate across different benchmarks since their efficiency is influenced by the initial configuration. Window search method performs poorly since its strategy is prone to fall into the local optimum. The orthogonal logarithmic only works well for Median and convolution kernels. The pattern-based search also has good kernel performance for most benchmarks, however, longer tuning time is always needed compared with other methods. Shrinking-sample method still shows competing performance for the kernels with small search space size like SpMV and *GEMM*.

C. Analysis of Results

We now analyze certain aspects of our benchmarks to understand relative performance of different methods.

First, we focus on the distribution of performance across different configurations for the benchmark kernels – the data is shown in Table IV. The density of high-performing configurations is low in all kernels except *GEMM* and SpMV. Overall, shrinking-sample method is within 1-3% of the execution times of the brute force search method across all applications. In comparison, the fraction of the configurations in the search space that are within 5% higher than optimal kernel running time is 0.04% for raycast, 0.1% for `stereo`, 0.13% for convolution, and 0.17% for median. This shows that our method is very effective in finding points in the entire space that provide optimal or close to optimal performance. This is because the shrinking sample method treats each parameter independently and looks at sample points more evenly scattered inside the search space. This also shows that our hypothesis that there is a spatial correlation with respect to performance – taking median of sections of parameter values and then searching in the section where best performance is achieved is effective.

On the other hand, certain local search methods (such as orthogonal search and window search), may find a local optimal configuration with performance much worse than the global optimum. That is because the search process is always executed within the neighborhood of a single

TABLE IV: Distribution of Kernel Configuration Performance

The configurations in the search space are divided based on their kernel running time. A configuration with kernel running time within 5% higher than the optimal kernel running time is considered as high-performing. The percentages of configurations with different performance ranges are listed.

GPU Kernel	$< 5\%$	$5\% - 10\%$	$> 10\%$
bilateral	1.68%	2.05%	96.27%
convolution	0.13%	0.44%	99.43%
GEMM	11.11%	88.89%	0
median	0.17%	0.53%	99.30%
raycast	0.04%	0.38%	99.58%
SpMV	16.66%	16.66%	66.68%
stereo	0.10%	0.44%	99.46%

local optimum. The pattern-based search yields better performance compared with the other local search methods because for each iteration it considers three values of each tuning parameter (the origin, one step forward and one step backward). However, more tuning time is spent on reaching convergence.

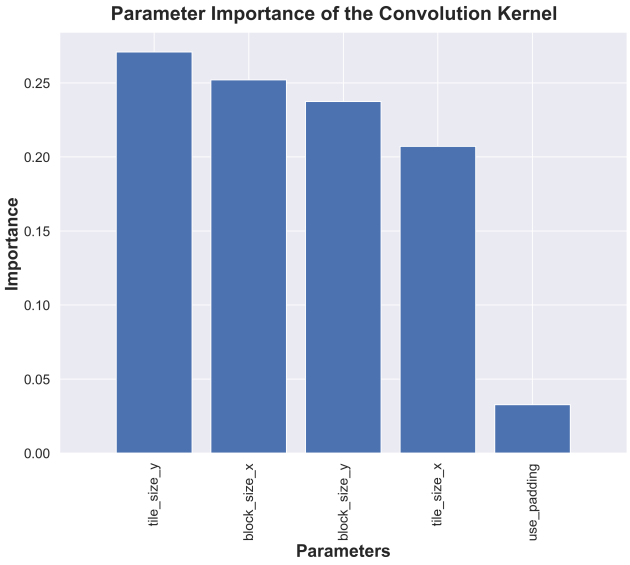

Fig. 11: Relative importance of tuning parameters for Convolution Kernel

Next, we analyze the tuning parameter importance (for achieving better performance) by using the random forests technique. We report typical results for only two applications: convolution and SpMV. The results for convolution are shown in Figure 11. Here, four of the five parameters have a significant impact on the performance - the only exception is the padding parameter, which is introduced to avoid shared memory bank conflicts, but its effects are diminished by using a filter of shape (17×17) on GPUs with 32 memory banks [33] as in Table II. Now, comparing this observation against Figure 6, we see that this is also the application where shrinking sample performs extremely well – matching the execution time of brute force search and a tuning time only slower than two of

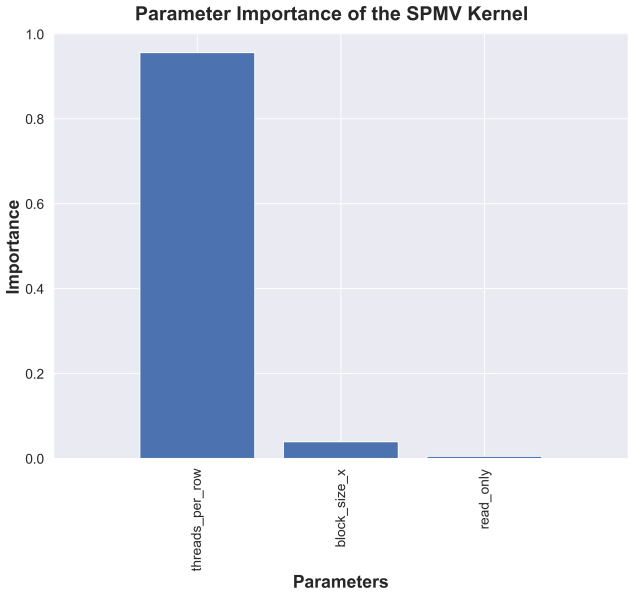

Fig. 12: Relative importance of tuning parameters for SpMV Kernel

the other seven methods. This shows the effectiveness of searching combination of parameter values.

On the other hand, Figure 12 shows that the tuning parameter, number of threads per row, in SpMV kernel has a dominating importance. Comparing with results in Figure 10, we find that many other methods can also deliver a version that performs similar to the best one. However, shrinking sample is still a competitive method both in terms of kernel execution time and tuning time.

VI. Conclusions

We have developed a model-free search algorithm for autotuning, called the shrinking-sample method, for GPU applications. This algorithm focuses on efficient search, while avoiding random guesses, and ensuring that all parts of the search space are covered. Like a global search method, the shrinking-sample method works by decreasing the search scope for each tuning parameter independently and reconstructing a smaller search space by combining possible values for each tuning parameter. However, this method chooses the best section of possible values of each tuning parameter for further search in a greedy way just like a local search method. Therefore, shrinking-sample method aims to take advantages of both local and global search strategies. This algorithm is evaluated by using seven popular benchmark GPU kernels and against four global and three local search methods. The simulation shows that shrinking-sample method has robust and efficient performance among other developed search algorithms considering various search space sizes and different high-performing configurations density. Across seven applications, our method not only chooses better configurations than local methods, but almost always outperforms global methods as well. At the same time, the tuning time is significantly lower than that of global methods (and in some cases, also lower than that of local methods).

Acknowledgements: This work was partially supported by the following NSF grants: 1629392, 2007793, 2034850, 2131509, and 2018627.

References

[1] A.Davidson and J.D.Owen. Toward techniques for auto-tuning gpu algorithms. In *Proceedings of Para 2010: State of the Art in Scientific and ParallelComputing*, 2010.

[2] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe. Opentuner: An extensible framework for program autotuning. In *the 23rd international conference on Parallel architectures and compilation, ACM*, page 303–316, 2014.

[3] P. Balaprakash and J. Dongarra. Autotuning in high-performance computing applications. In *PROCEEDINGS OF THE IEEE*, volume 106, 2018.

[4] P. Balaprakash, S. M. Wild, and P. D. Hovland. Can search algorithms save large-scale automatic performance tuning? *Procedia Comput. Sci.*, 4:2136–2145, 2011.

[5] P. Balaprakash, S. M. Wild, and P. D. Hovland. An experimental study of global and local search algorithms in empirical performance tuning. In *Proc. 10th Int. Conf. Revised Sel. Papers High Perform. Comput. Comput. Sci. (VECPAR). Springer*, number 261-269, 2013.

[6] L. Bao, X. Liu, and W. Chen. Learning-based automatic parameter tuning for big data analytics frameworks. *CoRR, vol. abs/1808.06008*, 2018.

[7] S. A. Billings and H. L. Wei. An adaptive orthogonal search algorithm for model subset selection and non-linear system identification. *International Journal of Control*, 81(5):714–724, May 2008.

[8] J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel. Optimizing matrix multiply using phipac: a portable, high-performance, ansi c coding methodology. In *Proceedings of the 1997 ACM International Conference on Supercomputing*, June 1997.

[9] P. Bruel, M. Amars, and A. Goldman. Autotuning cuda compiler parameters for heterogeneous applications using the opentuner framework. *Concurrency and Computation: Practice and Experience*, page e3973, 2017.

[10] C. Chen, J. Chame, and M. W. Hall. Combining models and guided empirical search to optimize for multiple levels of the memory hierarchy. In *Proceedings of the International Symposium on Code Generation and Optimization*, Mar. 2005.

[11] Y. Ding, J. Ansel, K. Veeramachaneni, X. Shen, U.-M. O'Reilly, and S. Amarasinghe. Autotuning algorithmic choice for input sensitivity. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2015.

[12] T. L. Falch. Auma: Auto tuning by machine learning. https://github.com/acelster/ML-autotuning, 2016.

[13] T. L. Falch and A. C. Elster. Machine learning based auto-tuning for enhanced opencl performance portability. In *IEEE International Parallel and Distributed Processing Symposium Workshops*, 2015.

[14] W. Feng and T. S. Abdelrahman. A sampling based strategy to automatic performance tuning of gpu programs. In *IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2017.

[15] M. Frigo. A fast fourier transform compiler. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, May 1999.

[16] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. In *International Journal of Parallel Programming*, volume 34, pages 261–317, June 2005.

[17] J. Guerreiro, A. Ilic, N. Roma, and P. Tomás. Multi-kernel auto-tuning on gpus: Performance and energy-aware optimization. In *Euromicro Int. Conf. Parallel Distrib. Netw.-Based Process*, pages 438–445, 2015.

[18] J. Holland. Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control and artificial intelligence. In *MIT Press, Cambridge, MA, USA*, 1992.

[19] C. Jung, S. Rus, B. P. Railing, N. Clark, and S. Pande. Brainy: effective selection of data structures. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, volume PLDI '11, pages 86–97, 2011.

[20] J. Kennedy. Particle swarm optimization, in: Encyclopedia of machine learning. *Springer*, pages 760–766, 2011.

[21] S. Kirkpatrick, C. Gelatt, and M. Vecchi. Optimization by simulated annealing. *Springer*, pages 671–680, 1983.

[22] T. Kisuki, P. W. Knijnenburg, and M. F. P. O'Boyle. Combined selection of tile sizes and unroll factors using iterative compilation. In *Proc. Int. Conf.Parallel Archit. Compilation Techn., Washington, DC, USA*, number 237-246, Oct. 2000.

[23] J. Kurzak, Y. M. Tsai, M. Gates, A. Abdelfattah, and J. Dongarra. Massively parallel automated software tuning. In *ICPP 2019 Proceedings of the 48th International Conference on Parallel Processing*, number 92, 2019.

[24] Y. Li, J. Dongarra, and S. Tomov. A note on auto-tuning gemm for gpus. In *International Conference on Computational Science*, page 884–892, 2009.

[25] C. Nugteren and V. Codreanu. Cltune: A generic auto-tuner for opencl kernels. *2015 IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip*, pages 195–202, 2015.

[26] A. Qasem, K. Kennedy, and J. Mellor-Crummey. Automatic tuning of whole applications using direct search and a performance-based transformation system. *J. Supercomput*, 36:183–196, May 2006.

[27] A. Rasch and S. Gorlatch. Atf: A generic directive-based auto-tuning framework. *CCPE*, page 1–16, 2018.

[28] M. Samadi, M. M. A. Hormati, J. Lee, and S. Mahlke. Adaptive input-aware compilation for graphics engines. In *Proceedings of ACM SIGPLAN 2012 Conference on Programming Language Design and Implementation*, 2012.

[29] A. Scriven. Informed search: Coarse hyperparameter tuning in python. https://s3.amazonaws.com/assets.datacamp.com/production/course_15167/slides/chapter4.pdf.

[30] K. Seymour, H. You, and J. Dongarra. A comparison of search heuristics for empirical code optimization. In *Proc. IEEE Int. Conf. Cluster Comput*, number 421-429, Sep./Oct. 2008.

[31] K. Tian, Y. Jiang, E. Zhang, and X. Shen. An input-centric paradigm for program dynamic optimizations. In *the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, June 2010.

[32] A. Tiwari, C. Chen, C. Jacqueline, M. Hall, and J. K. Hollingsworth. A scalable auto-tuning framework for compiler optimization. In *Proceedings of the 2009 IEEE International Symposium on ParallelDistributed Processing. Washington, DC, USA: IEEE Computer Society*, number 1-12, 2009.

[33] B. van Werkhoven. Kernel tuner: A search-optimizing gpu code auto-tuner. *Future Generation Computer Systems*, 90:347–358, 2019.

[34] B. van Werkhoven, J. Maassen, H. Bal, and F. Seinstra. Optimizing convolution operations on gpus using adaptive tiling. *Future Generation Computer Systems*, 30:14–26, 2014.

[35] R. Vuduc, J. W. Demmel, and K. A. Yelick. Oski: A library of automatically tuned sparse matrix kernels. In *Journal of Physics: Conference Series*, volume 16, pages 521–530, June 2005.

[36] R. C. Whaley and J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of Supercomputing '98*, Nov 1998.

[37] J. Xiong, J. Johnson, R. Johnson, and D. Padua. Spl: A language and compiler for dsp algorithms. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2001.

[38] K. Yotov, X. Li, G. Ren, M. Garzaran, D. Padua, K. Pingali, and P. Stodghill. Is search really necessary to generate high-performance blas? In *Proceedings of the IEEE: Special Issue on Program Generation, Optimization, and Platform Adaptation*, volume 93, pages 358–386, Feb. 2005.