2021 IEEE International Conference on Cluster Computing (CLUSTER) | 978-1-7281-9666-4/21/$31.00 ©2021 IEEE | DOI: 10.1109/Cluster48925.2021.00037

csTuner: Scalable Auto-tuning Framework for Complex Stencil Computation on GPUs

Qingxiao Sun[1,2], Yi Liu[2], Hailong Yang[1,2], Zhonghui Jiang[2], Xiaoyan Liu[2], Ming Dun[3], Zhongzhi Luan[2], Depei Qian[2]
State Key Laboratory of Software Development Environment[1], Beijing, China, 100191
School of Computer Science and Engineering[2], Beihang University, Beijing, China, 100191
School of Cyber Science and Technology[3], Beihang University, Beijing, China, 100191
{qingxiaosun,yi.liu,hailong.yang,jiangzhh,liuxiaoyan,dunming0301,zhongzhi.luan,depeiq}@buaa.edu.cn

***Abstract*—The computational patterns of stencil operations are commonly used in HPC applications. Many HPC platforms utilize the computation capability of GPUs to accelerate stencil operations. In recent years, stencils have become more complex in terms of stencil order, memory accesses, and operator patterns. To adapt complex stencils to GPUs, various optimization techniques have been proposed such as blocking and unrolling. However, due to the complexity of GPU architecture, no single parameter setting of the optimization techniques fits all stencils. To address this problem, we propose *csTuner*, a scalable auto-tuning framework that quickly determines the optimal parameter setting for a given combination of optimization techniques. Specifically, *csTuner* leverages a set of statistics and machine learning methods to generate parameter groups and sampled parameter settings from the search space. In addition, *csTuner* adopts the genetic algorithm with approximation to reduce the cost of evolutionary search. The experimental results show that *csTuner* can find better performing settings with higher auto-tuning speed compared to the state-of-the-art works.**

***Index Terms*—Stencil computation, GPU, Auto-tuning, Machine learning, Statistics, Genetic algorithm**

I. Introduction

The stencil computation is one of the most adopted computation patterns in scientific computing applications. Stencil computations appear in many domains such as image processing [26], physical simulation [12] and cellular automata [41]. A stencil computation sweeps a computational grid and processes the fixed neighbors around each point to update its value, where the extent of the neighbors along each dimension is referred to as the *stencil order*. For instance, box-shape stencils are used to perform smoothing and other neighbor-pixel-based computations in image processing [30], [34].

In recent years, stencil computations have become more complex in terms of stencil order, data accesses and operation patterns [40], [49], [52]. For instance, the stencils in many physical simulations are conducted in order-3 or above while processing data from multiple input grids. Besides, a single iteration of stencil computations may perform hundreds of double-precision floating-point operations (FLOPs) [1]. Consistent with existing works [38], we define the above stencils as *complex stencils*. The complex stencils tend to have more abundant parallelism, which makes them excellent candidates for acceleration on GPUs.

However, it is challenging for stencil computations to achieve high performance on GPUs. The programmers must ensure memory coalescing, overlapped computation and data access, reduced synchronization and thread divergence elimination, and trading off between parallelism and resource utilization. Therefore, many optimization techniques based on tiling and streaming [15], [17], [24], [35], [51] have been proposed to adapt to the architecture properties of high computation capability and limited memory bandwidth on GPUs. However, no single optimization technique fits all complex stencils due to the diversity of stencil patterns and increased complexity of GPU architecture [27], [28].

Stencil Domain-Specific Languages (DSLs) explore the automatic code generation with the integration of optimization techniques, and thus allow users to select specific optimizations with parameters fine-tuned [8], [25], [30], [33], [38]. However, it is difficult to evaluate the performance impact of individual optimization techniques when combined. In addition, the optimal parameter settings for each optimization technique vary across the different stencil patterns and GPU architectures. To reduce engineering efforts of parameter tuning, stencil DSLs commonly adopt exhaustive search with expert knowledge to achieve performance auto-tuning.

The auto-tuning mechanisms integrated in the stencil DSLs commonly have similar limitations, *1)* poor scalability to adapt to more optimization options and stencil patterns, and *2)* time-consuming search process with large optimization space. To overcome the above limitations, several stencil auto-tuning frameworks [13], [23] have been proposed to reduce the search overhead and determine the optimal parameter settings for the combined optimization techniques. However, they heavily rely on expert knowledge to narrow the search space. Moreover, the random sampling may ignore the global optimal parameter settings, thus deteriorating the effectiveness of auto-tuning.

To address the above drawbacks, we propose a scalable auto-tuning framework *csTuner*, which efficiently determines the optimal parameter settings of a given optimization combination for stencil computations on GPUs. The *csTuner* first parameterizes the stencil optimization techniques into the parameter search space. After that, the *csTuner* generates the parameter groups and sampled parameter settings through a set of statistic and machine learning methods. Finally, the *csTuner*

Authorized licensed use limited to: BEIHANG UNIVERSITY. Downloaded on December 20,2021 at 02:33:54 UTC from IEEE Xplore. Restrictions apply.

utilizes a customized genetic algorithm with approximation to reduce the cost of evolutionary search. We evaluate *csTuner* on different complex stencils to prove its effectiveness in performance auto-tuning.

Specifically, this paper makes the following contributions:

- We comprehensively analyze the optimization techniques of stencil computations with their parameterized scope on GPUs. Besides, we implement rules for checking optimization constraints so that only valid parameter settings are explored.
- We propose a search space narrowing mechanism, which utilizes a set of statistic and machine learning methods to generate the parameter groups and sampled parameter settings. The parameter groups are used to design regression functions that guide the sampling process.
- We design and implement an evolutionary search mechanism, which reduces the searching cost by considering approximation and performs iterative auto-tuning with a customized genetic algorithm.
- We develop a scalable auto-tuning framework *csTuner* that efficiently determines the optimal parameter settings of optimization combinations for the stencil computations on GPUs. The experimental results show that *csTuner* can identify high-performance parameter settings in a shorter time.

The rest of this paper is organized as follows. Section II and Section III present the background and motivation. Section IV presents the details of *csTuner* methodology. Section V presents the evaluation results of *csTuner*. Section VI discusses the related work, and Section VII concludes this paper.

II. Background

A. GPU Architecture and Execution Model

The GPU consists of dozens to hundreds of Streaming Multiprocessors (SMs) depending on the GPU generation. As shown in Figure 1, each SM contains hundreds of computing cores and other resources such as registers, shared memory and L1 cache. The code executed on the GPU is called *kernel*. When a kernel is launched on the CPU host, thousands of threads are created on GPU and every 32 threads are grouped into a *warp*. Furthermore, multiple warps are grouped into a *thread block* (TB), and the size of a TB is determined by kernel configuration. The TB scheduler dispatches TBs to SMs according to the Round-Robin policy, which maximizes the GPU occupancy under resource and hardware constraints.

Due to the limited computing resources in SMs [16], GPU tasks have to be fine-tuned to achieve a tradeoff between system utilization and performance speedup. For instance, some optimization strategies (e.g., loop unrolling) increase register-level data reuse to improve performance [36]. However, the resulting code is highly constrained by register pressure and may even cause register spilling. Constant and texture memory can be used to reduce memory access latency for read-only data during kernel execution. However, performance gain only occurs when there is a cache hit in the constant or texture

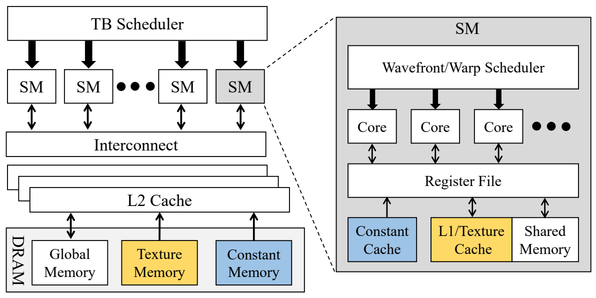

Fig. 1. The basic hardware architecture of GPUs.

memory, otherwise the data still needs to be read from the global memory and exacerbates the bandwidth competition.

B. Optimizations for Stencil Computation

With the continuous development of GPU, it has attracted widespread attention from academia to accelerate stencil computation on GPU [15], [25], [34]–[36], [40], [51]. We briefly discuss the existing GPU optimization techniques for stencil computation.

1) Streaming: Streaming is a commonly used optimization that improves data reuse and reduces computation redundancy along the streaming dimension. For 3-D input grids, an effective implementation of streaming is 2.5-D spatial blocking [25]. Specifically, the computation of 2-D tiles is streamed over one dimension, and the data of each tile is reused for updating the next tiles. However, given large problem size, streaming increases computation granularity thus limiting parallelism. To achieve better performance, concurrent streaming [38] divides the streaming dimension into tiles. In concurrent streaming, the TBs traverse the streaming dimension in parallel at the granularity of tiles. Meanwhile, loop unrolling has also been applied to increase register-level data reuse.

2) Block/Cyclic Merging: Naively, each GPU thread works on a single output point. Merging the computations of several output points reduces the overhead of kernel launching and eliminates duplicated memory accesses. Two strategies have been proposed for merging computations such as block merging and cyclic merging. For block merging, a number of adjacent output points are merged. Whereas for cyclic merging, every two points are merged with a fixed distance. However, both strategies may increase the register pressure and reduce the number of threads that resided on each SM, thus hurting parallelism. Furthermore, block merging in the innermost dimension of the global grid can disrupt memory coalescing [13]. In general, the choice of merging strategy and the number of points to merge can significantly impact the stencil performance.

3) Prefetching: In streaming optimization, after updating the output grid in current iteration, the data located in the shared memory is shifted to continue the computation for next iteration. Due to the concurrent execution of massive threads on GPU, a synchronization barrier has to be performed

Authorized licensed use limited to: BEIHANG UNIVERSITY. Downloaded on December 20,2021 at 02:33:54 UTC from IEEE Xplore. Restrictions apply.

between adjacent iterations to ensure the correctness of the results. The synchronization can cause serialization between kernels and thereby deteriorate performance. Prefetching [38] can hide the delay of synchronization by overlapping the computation and data loading. Specifically, the data used for the next iteration is loaded into registers simultaneously with the computation of the current iteration. However, prefetching may exhaust the registers that are quite limited on GPU.

4) Retiming: Retiming [44] improves data reuse by decomposing a stencil computation into a set of sub-computations along with accumulations. Retiming can balance the resource usage between memory and registers by homogenizing stencil accesses [36]. In general, high-order stencils can benefit from retiming optimizations due to the effective reuse of registers. However, retiming may not improve the performance of stencils with low register pressure.

However, the effectiveness of the above optimization techniques depends on various factors such as GPU architecture, stencil shape/order, and computational pattern. It can easily lead to sub-optimal performance if the parameter settings are not chosen appropriately. This motivates the emergence of auto-tuning mechanisms for stencil computation.

C. Existing Stencil Auto-tuning Mechanisms

Existing stencil Domain-Specific Languages (DSLs) expose performance-related parameters to auto-tuning mechanisms integrated in their frameworks [8], [25], [30], [33], [38]. For instance, *Halide* [33] applies stochastic search to automatically find good pipeline schedules. Besides, Halide limits the number of domain scheduling operations for each function to prevent the explosion of generated code. *Artemis* [38] prunes the search space by hierarchical auto-tuning. Specifically, *Artemis* tunes the computation for high-impact optimizations first and then selects a few high-performance candidates. *GoPipe* [30] finds the best task granularity and scheduling mechanism for each stage of a pipelined box stencil (e.g., image convolution). *AN5D* [25] obtains the top-5 parameter sets based on the performance model and chooses the one that achieves the highest performance in actual execution.

However, the auto-tuning mechanisms integrated into the above DSLs have similar limitations. First of all, the narrowing of the search space (e.g., hierarchical auto-tuning) relies on expert knowledge, which lacks the generality to adapt the auto-tuning mechanisms to various complex stencils and GPU architectures. Secondly, the fixed sampling of the search space may ignore the global optimal parameter settings, thus deteriorating the effectiveness of auto-tuning. Thirdly, since the auto-tuning mechanisms are customized for particular stencil DSLs, they have poor scalability to evaluate more optimization techniques during parameter tuning. Finally, the exhaustive or stochastic methods are insufficient to guide the search along the descent direction, thereby increasing the overhead of auto-tuning.

To overcome the above limitations, several works have considered to speed up the auto-tuning performance of stencil computation [13], [23]. *Garvey* [13] trains a random forest [5] to predict the optimal memory type and group optimization parameters based on experience. After that, *Garvey* exhaustively searches for the parameter settings of each group with random sampling enabled. However, the parameter grouping relies on expert knowledge and lacks generality. Besides, random sampling may ignore the optimal parameter settings and deteriorate the effectiveness of auto-tuning. *FAST* [23] trains a regression model to predict the similarity of the optimal solution space and uses the database to build the mapping between the parameter features and optimization solution. However, *FAST* does not consider the correlation between parameters and introduce large search cost without sampling. *OpenTuner* [3] implements a collection of search techniques (e.g., differential evolution and hill climber) to find the optimal solution. Since *OpenTuner* targets general-purpose computing, it misses optimization opportunities specific to auto-tuning stencil computation on GPUs.

III. Motivation

We make three main observations on the characteristics of stencil computations on GPUs under various parameter settings. The same observations have been proved in existing stencil auto-tuning works [13], [25] or other domains such as tensor program optimization in deep learning [50]. To this end, we randomly sample more than 20,000 parameter settings for each stencil to motivate our work. See Section V-A for details of the experimental configuration.

A. Low Proportion of High-performance Settings

Figure 2 shows the speedup distribution of parameter settings over the optimum, where the speedup values are divided into 5 bins from 0 to 1 with a stride of 0.2. The x-axis indicates the stencils used for evaluation (refer to Table III for details). Each bar indicates the percentage of parameter settings that fall in the given speedup bin. As seen, the majority of the parameter settings perform poorly, with only 5.1% on average achieving a speedup within 20% of the optimal performing setting. In turn, a large fraction of the settings (24.2% on average) achieves more than a $5\times$ slowdown relative to the optimum. This shows that the optimization space is biased towards poorly performing settings. At this time, random sampling of the optimization space is likely to fail to obtain high-performance settings. Therefore, external intervention is required to filter out poorly performing settings as much as possible so as to improve the sampling quality.

B. Correlation Between Optimization Parameters

Figure 3 shows the percentage distribution of parameter pairs with other parameters fixed. The percentage in the legend indicates the ratio when values of one parameter achieving the best performance with another parameter fixed are different from the value of the optimal parameter setting. The missing yellow bin means that no percentage falls in the $[80\%, 100\%)$ interval. A percentage less than 1 means that the optimum may not be obtained when the pair parameters are tuned separately. In addition, a higher percentage indicates a larger

Authorized licensed use limited to: BEIHANG UNIVERSITY. Downloaded on December 20,2021 at 02:33:54 UTC from IEEE Xplore. Restrictions apply.

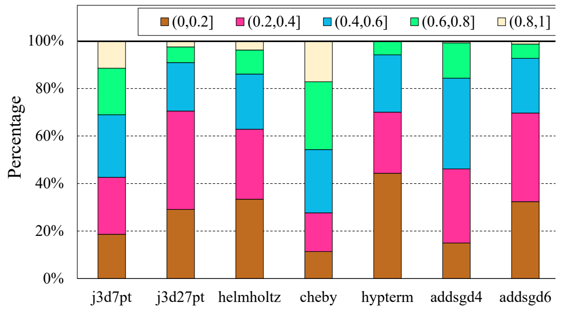

Fig. 2. Speedup distribution of parameter settings over the opt

gap between the results after separate tuning and the optimum. As seen, a large fraction of parameter pairs (28.6% on average) includes parameter values that are not consistent with the optimum. Moreover, an average of 22.3% of the parameter pairs differs from the optimum by more than 40%. The above results prove that there is a difference in the strength of correlation between the optimization parameters. Therefore, the pair-wise correlation shall be taken into account when grouping parameters in order to narrow the gap with the global optimum.

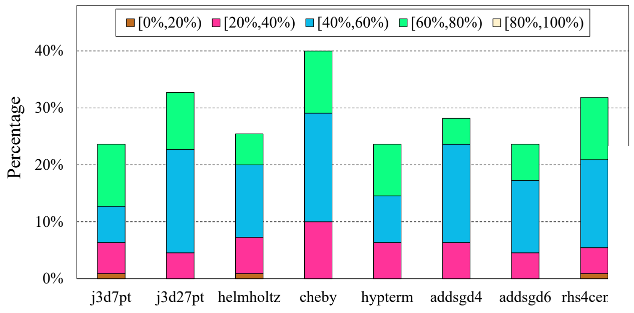

Fig. 3. Percentage distribution of parameter pairs with other parameters fi

C. Approximation of High-performance Settings

Figure 4 shows the speedup of the top-n parameter settii over the optimum. The speedup value of each bar is taken fr the nth best parameter setting, thus representing the larg gap among top-n settings. As seen, top-10, top-50 and t 100 parameter settings all achieve relatively high speed with an average of 96.7%, 92.4% and 90.1% respectively. This indicates that the performance gap among a fixed number of optimal parameter settings is acceptable. Therefore, we can consider finding an approximate optimal parameter setting without determining the global optimum. Doing so achieves a tradeoff between search cost and final performance. Furthermore, the search process can be automatically terminated when the approximate condition is reached.

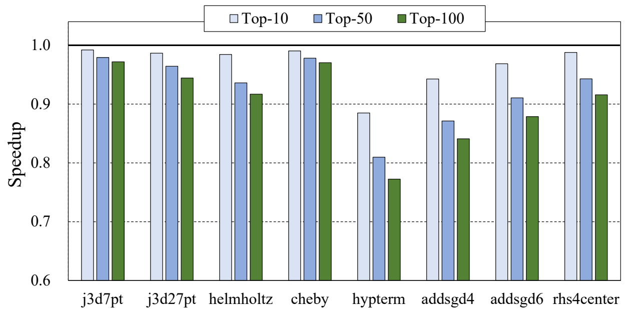

Fig. 4. Speedup of the top-n parameter settings over the optimum.

IV. csTuner Methodology

A. Design Overview

In this section, we propose a scalable stencil auto-tuning framework *csTuner* that determines the optimal parameter setting through a holistic pipeline with statistic and machine learning methods. As shown in Figure 5, the *csTuner* consists of three important components including parameter grouping (Section IV-C), search space sampling (Section IV-D) and evolutionary search (Section IV-E). The parameter grouping component aggregates the strongly correlated parameters through statistical metrics and grouping algorithms. The search space sampling designs a regression performance model, which is trained to ensure the effectiveness of the sampling process. The evolutionary search component implements iterative auto-tuning and efficiently finds the optimal parameter settings through the approximation of the genetic algorithm.

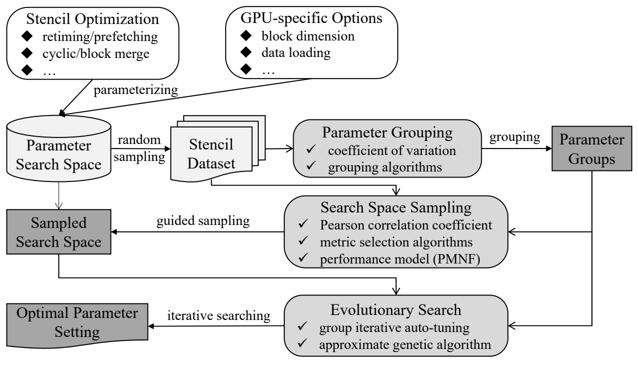

Fig. 5. The design overview of *csTuner*.

Figure 5 illustrates the design overview of *csTuner*. *csTuner* parameterizes the stencil optimization techniques and GPU-specific options into the parameter search space. After that, the *csTuner* randomly samples the search space and collects GPU metrics using *Nsight* [29] to obtain the performance dataset. Note that we only need a small-scale performance dataset for grouping parameters and training performance models, so the cost of collecting the performance dataset is acceptable for the entire auto-tuning pipeline. The parameter groups and sampled search space are used as the input of the customized

Authorized licensed use limited to: BEIHANG UNIVERSITY. Downloaded on December 20,2021 at 02:33:54 UTC from IEEE Xplore. Restrictions apply.

genetic algorithm. The genetic algorithm performs auto-tuning iteratively with approximation regarding the parameter groups. This eliminates the need to manually set the number of iterations based on experience, thus improving the efficiency of auto-tuning.

The *csTuner* pipeline can be extended to incorporate more optimization parameters capturing future stencil optimizations. If the input or stencil pattern changes, the stencil dataset needs to be re-collected with the same process repeated to obtain the optimal parameter settings. In addition to stencil computation, the *csTuner* can also support auto-tuning of more general GPU algorithms due to the versatility of its components.

B. Optimization Space Parameterization

Firstly, we need to parameterize the combination of optimization techniques for the subsequent auto-tuning pipeline. Table I takes a 3-D stencil as an example to present the parameterized optimization space, where M_n is the length of the nth dimension of the input grid. We start from 1 with unit stride to represent the parameters of bool type (e.g., $useShared$) and enumeration type (e.g., SD). In such way, the *csTuner* ensures the legitimacy of the log operations involved in PMNF (Section IV-D). For the numerical parameters, the *csTuner* restricts their values to power of two in consistent with existing works [13], [25], [36]. Note that when grouping parameters, we perform $log2$ operation on the numerical parameters so that the input for calculating the coefficient of variation (CVs) is continuous (Section IV-C). This guarantees fairness when comparing the correlation of different parameter pairs.

TABLE I
THE PARAMETERIZED OPTIMIZATION SPACE OF STENCIL COMPUTATION ON GPUS.

Optimzation	Parameter	Range
TB Dimension	TB_x, TB_y, TB_z	[1, 1024], [1, 1024], [1, 64]
Shared Memory	$useShared$	$\{1, 2\}$
Constant Memory	$useConstant$	$\{1, 2\}$
Streaming	$useStreaming$	$\{1, 2\}$
Streaming Dimension	SD	$\{1, 2, 3\}$
Concurrent Streaming	SB	$[1, M_{SD}]$
Loop Unrolling	UF_x, UF_y, UF_z	$[1, M_1]$, $[1, M_2]$, $[1, M_3]$
Cyclic Merging	CM_x, CM_y, CM_z	$[1, M_1]$, $[1, M_2]$, $[1, M_3]$
Block Merging	BM_x, BM_y, BM_z	$[1, M_1]$, $[1, M_2]$, $[1, M_3]$
Retiming	$useRetiming$	$\{1, 2\}$
Prefetching	$usePrefetching$	$\{1, 2\}$

In addition to the parameter ranges illustrated in Table I, there are some explicit constraints between the optimization parameters. For instance, the TB size (i.e., $TB_x \times TB_y \times TB_z$) must be less than or equal to 1,024. Besides, the parameters of SD and SB are only valid when enabling streaming optimization. When enabling concurrent streaming, the loop unrolling factor in the streaming dimension must be no greater than SB. After enforcing these constraints, the total search space still contains more than 100 million parameter settings. Moreover, there are also implicit constraints due to the limited resources of GPUs. For instance, the settings of the block merging and loop unrolling are restricted by the usage of register and shared memory. *csTuner* checks the above constraints before generating the search codes so that only non-spilled parameter settings are explored during auto-tuning.

C. Parameter Grouping

The principle of parameter grouping is to put strongly correlated parameters in a group to facilitate the design of performance models (Section IV-D) and iterative genetic search (Section IV-E). Firstly, we need to quantify the correlation between any two parameters. Assuming there are N parameters $(P_0, P_1, ..., P_{N-1})$, where $P_i \in R_i$. We change the settings of two parameters with the remaining $N-2$ parameters fixed each time. The settings of the fixed parameters are taken from the optimal parameter settings in the performance dataset. Taking P_0 and P_1 for an example, we evaluate the settings of P_0 from R_0, and finally obtain the setting of P_1 (VS_{01}) that achieves the best performance with P_0 fixed. Note that we skip a particular setting in R_0 if it does not exist in the performance dataset. Next, we quantify the correlation between P_0 and P_1 by calculating the coefficient of variation (CV) of VS_{01} based on Equation 1.

$$c_v = \frac{\sigma}{\mu} = \frac{\sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2}}{\bar{x}} \tag{1}$$

The CV is defined as the ratio of the standard deviation σ to the mean μ. It is widely used to reflect the extent of dispersion of a data set in the field of statistics [2]. Here a higher CV means a lower correlation. Repeating the above process, we can finally obtain A_N^{N-1} CV values representing the correlation among the parameters. After that, we push the parameter pairs into a double-ended queue ($Deque$) [19] based on the ascending order of CVs. We design Algorithm 1 to implement parameter grouping, where the input is the $Deque$ data structure and the output is a list of parameter groups.

As illustrated in Algorithm 1, we first pop the leftmost parameter pair in $Deque$ (Line 7). If none of the two parameters is in the existing parameter groups, then we create a new parameter group to store the parameter pair (Line 8-9). Otherwise, we merge the parameter into the existing parameter group (Line 13-17). We then pop the rightmost parameter pair in $Deque$ (Line 21) and apply the similar process as above (Line 25-30). The former steps are repeated until $Deque$ is empty, and the final parameter groups are obtained by $csTuner$.

D. Search Space Sampling

The principle of search space sampling is to filter out low-performance parameter settings during the sampling process. Note that the performance of stencil computation can be impacted by various factors, so we build accurate performance models based on GPU metrics. The GPU metrics can be collected from the performance dataset using *Nsight*[1]. GPU

[1]The profiling output from *Nsight* includes numerous GPU metrics.

Authorized licensed use limited to: BEIHANG UNIVERSITY. Downloaded on December 20,2021 at 02:33:54 UTC from IEEE Xplore. Restrictions apply.

Algorithm 1 Parameter grouping based on $Deque$ structure.

```
1: Input: double-ended queue $Deque$
2: Output: parameter group list $paraGroup$
3: $queSize = Deque.size$ // original stack size
4: for $i$ in range $[0, queSize)$ do
5:   if $i\%2 == 1$ then
6:     // pop the parameter pair from the right side
7:     $ftPara, bkPara = Deque.popright()$
8:     if neither $ftPara$ and $bkPara$ in $paraGroup$ then
9:       $paraGroup.append([ftPara, bkPara])$
10:    else if both $ftPara$ and $bkPara$ in $paraGroup$ then
11:      $continue$ // skip to next iteration
12:    else
13:      if $ftPara$ in $paraGroup$ then
14:        $paraGroup[ftIn].append(bkPara)$
15:      else
16:        $paraGroup[bkIn].append(ftPara)$
17:      end if
18:    end if
19:  else
20:    // pop the parameter pair from the left side
21:    $firstPara, secondPara = Deque.popleft()$
22:    if both $ftPara$ and $bkPara$ in $paraGroup$ then
23:      $continue$ // skip to next iteration
24:    else
25:      if $ftPara$ not in $paraGroup$ then
26:        $ftPara.append([ftPara])$
27:      end if
28:      if $bkPara$ not in $paraGroup$ then
29:        $bkPara.append([bkPara])$
30:      end if
31:    end if
32:  end if
33: end for
```

metrics can be used to comprehensively analyze the computation and memory efficiency of stencil tasks during execution. For example, the SM utilization and L1/texture/L2 cache hit rate can indicate the efficiency of computation and memory, respectively. However, it is unrealistic to build performance models based on all GPU metrics collected. We combine GPU metrics by comparing the Pearson correlation coefficient (PCC) [4] of pairwise metrics based on Equation 2.

$$PCC(x, y) = \frac{\mathbb{E}[(x - \mu_x)(Y - \mu_y)]}{\sqrt{\sum_{i=1}^{n}(x_i - \mu_x)^2}\sqrt{\sum_{i=1}^{n}(y_i - \mu_y)^2}} \quad (2)$$

The closer to 1 the absolute value of PCC is, the stronger the linear correlation of the metric pair is. Similarly, we push the metric pairs into a $Deque$ according to the ascending order of PCCs. Algorithm 2 illustrates the process of metric combination, where the input includes the $Deque$ structure and the number of metric collections, and the output is a list of metric collections. Specifically, we pop the rightmost metric pair each time. If none of the metrics is in the existing metric collections (Line 7) and the $metrCollection$ size is smaller than $numCollections$ (Line 8), then we create a new metric collection to store the metric pair (Line 9). Otherwise, we merge the metric pair into the existing metric collections (Line 16-20). The metric collections are generated by repeating the above steps until $Deque$ is empty. Finally, we select the metric with the highest PCC correlated to execution time from each metric collection for performance modeling.

Algorithm 2 Metric combination based on $Deque$ structure.

```
1: Input: double-ended queue $Deque$ and number of metric collections $numCollection$
2: Output: metric collection list $metrCollection$
3: $queSize = Deque.size$ // original stack size
4: for $i$ in range $[0, queSize)$ do
5:   // pop the metric pair from the right side
6:   $ftMetr, bkMetr = Deque.popright()$
7:   if neither $ftMetr$ and $bkMetr$ in $metrCollection$ then
8:     if $metrCollection.size < numCollection$ then
9:       $metrCollection.append([ftMetr, bkMetr])$
10:    else
11:      $continue$ // skip to next iteration
12:    end if
13:  else if both $ftMetr$ and $bkMetr$ in $metrCollection$ then
14:    $continue$ // skip to next iteration
15:  else
16:    if $ftMetr$ in $metrCollection$ then
17:      $metrCollection[ftIn].append(bkMetr)$
18:    else
19:      $metrCollection[bkIn].append(ftMetr)$
20:    end if
21:  end if
22: end for
```

For each metric selected, we construct a multi-parameter performance model that predicts the GPU metrics based on input parameter values. We define the non-linear regression function based on performance model normal form (PMNF), which assumes that the performance can usually be expressed as a combination of polynomial and logarithmic terms [9]. PMNF defines a function search space, which is traversed to find the function that most accurately represents the relationship between the parameters and the metrics. However, the PMNF incurs unpredictable search overhead for multi-parameter models, which forces the state-of-the-art (SOTA) performance modeling tool such as Extra-P [6], [39], [42] to only support up to four-parameter PMNF. Therefore, we leverage the parameter groups to simplify the multi-parameter PMNF, where n is the number of groups and m_k is the number of parameters in the kth group.

$$f(P) = \sum_{k=1}^{n} c_k \prod_{l=1}^{m_k} P_l^i \cdot log_2^j(P_l) \quad (3)$$

Inspired by [10], we multiply the parameters within a group (strong correlation) and accumulate the parameters of different groups (weak correlation) as shown in Equation 3. Supposing that $i \in I$ and $j \in J$, the search space of PMNF is reduced to $I \times J$ regardless of the number of parameters. Next, we train a non-linear regression model for each function using the performance dataset, which provides the fitted values of the coefficients (c_k). Since R^2 score is valid only for linear regression [43], we use residual standard error (RSE) [7] to measure the fitness of the functions and select the best one for search space sampling. Empirically, we set a $threshold$ for

Authorized licensed use limited to: BEIHANG UNIVERSITY. Downloaded on December 20,2021 at 02:33:54 UTC from IEEE Xplore. Restrictions apply.

each selected GPU metric to filter out inappropriate parameter settings during the sampling process.

E. Evolutionary Search with Approximation

Although the search space has been greatly narrowed after sampling, it is still time-consuming to use exhaustive search to determine the optimal settings of optimization parameters. Therefore, we propose an evolutionary search using genetic algorithm [14] to find the optimal parameter settings efficiently. Figure 6 presents the multi-process genetic algorithm in *csTuner*. As shown, multiple genes constitute an individual, which is evaluated by the fitness. Many individuals constitute a population, where the operations of each sub-population are handled by a process. The migration among the sub-populations is achieved using MPI communication. For migration, each sub-population exchanges individuals with its two neighborhoods (single-ring topology [48]).

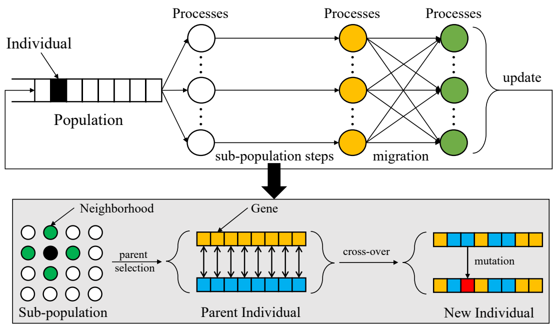

Fig. 6. Multi-process genetic algorithm in *csTuner*.

The new individual in the sub-population is bred through uniform cross-over and mutation. The breeding involves three steps: *1)* the parents are selected from the four neighborhoods (higher fitness means higher selection chance); *2)* each gene of the individual is randomly chosen from the parents; *3)* the genes of the individual mutate with a certain probability (mutation rate). The mutation is used to prevent the individuals from falling into local optimum [46].

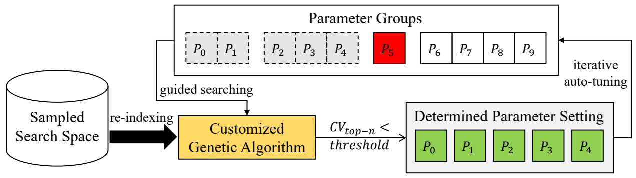
Parameter Groups
P_0 P_1 P_2 P_3 P_4 P_5 P_6 P_7 P_8 P_9
iterative auto-tuning
guided searching
Sampled Search Space
re-indexing
Customized Genetic Algorithm
CV_{top-n} < threshold
Determined Parameter Setting
P_0 P_1 P_2 P_3 P_4

Fig. 7. Iterative auto-tuning of *csTuner* based on parameter groups.

For adopting the genetic algorithm in *csTuner*, we need to guide the search process according to the parameter groups. Genes are stored in binary for mutation, and the valid value range of each gene needs to be given before initialization. However, the parameter values in the sampled search space are no longer continuous. In such a case, there may be a large number of invalid points outside the search space during initialization and mutation. To address this problem, we re-index the valid values of the parameter groups from the sampled search space. As shown in Figure 7, assuming that the available value set of the first parameter group (P_0, P_1) are $\{(0,1),(4,2),(3,4)\}$, we re-index the set to $\{0,2,1\}$ based on the ascending order. Then, the value range of the gene g can be designated as $[0,2]$, and $g \in N$.

The auto-tuning results usually conform to the normal distribution, which means that the top-n parameter settings achieve similar performance [25], [50]. Here we still use CV to represent the approximation of the top-n parameter settings. Empirically, the parameter setting of a group is determined if the CV of top-n fitness (CV_{top-n}) is less than a certain $threshold$. After that, we proceed to the auto-tuning of the following parameter group until all parameter groups are tuned. By incorporating approximation, *csTuner* greatly reduces the overhead of evolutionary search and stops the genetic algorithm without manual intervention.

V. Evaluation

A. Experiment Setup

1) Hardware Platforms and Stencil Programs: The hardware and software specifications are presented in Table II. The PMNF regression models involved in *csTuner* is fitted with the $curve\_fit$ function using scikit-learn v0.23.1 [31]. We explore a set of eight 3-D double-precision stencil programs taken from [36] (Table III). The selected stencils are a mixture of various patterns including stencil order, FLOPs, input grid and I/O arrays. This mixture of stencils provides a comprehensive evaluation of the effectiveness of *csTuner*.

TABLE II
Hardware and Software Specifications.

Hardware	Software
CPU: Intel(R) Xeon(R) CPU E5-2680 v4 @2.40GHz	GPU driver: 455.23.05
GPU: NVIDIA Tesla A100 × 2	CUDA version: 11.1
OS: Linux version 5.8.0-50-generic	GCC version: 9.3

TABLE III
Stencils Used for Evaluation.

Stencil	Input Grid	Order	# FLOPs	# I/O Arrays
j3d7pt	512 × 512 × 512	1	10	2
j3d27pt	512 × 512 × 512	1	32	2
helmholtz	512 × 512 × 512	2	17	2
cheby	512 × 512 × 512	1	38	5
hypterm	320 × 320 × 320	4	358	13
addsgd4	320 × 320 × 320	2	373	10
addsgd6	320 × 320 × 320	3	626	10
rhs4center	320 × 320 × 320	2	666	8

2) Search Methods and Implementation Details: We compare *csTuner* with three popular stencil auto-tuning methods including *Garvey* [13], *OpenTuner* [3] and *Artemis* [36]. For *csTuner*, we randomly sample 128 parameter settings for each stencil as a stencil dataset for grouping parameters and fitting the PMNF models. We set the ranges of i and j involved in the

Authorized licensed use limited to: BEIHANG UNIVERSITY. Downloaded on December 20,2021 at 02:33:54 UTC from IEEE Xplore. Restrictions apply.

PMNF (See Equation 3) to $\{0, 1, 2\}$ and $\{0, 1\}$, respectively. In addition, the sampling ratio of the parameter search space is set to 10%. For the genetic algorithm adopted in *csTuner*, the number of sub-populations is set to 2, where each sub-population contains 16 individuals. The cross-over rate and the mutation rate are set to 0.8 and 0.005, respectively. Note that if the settings of a parameter group are less than the individuals of a population, the auto-tuning of the parameter group degenerates to the exhaustive search.

For *OpenTuner*, we adopt the global genetic algorithm as the basis of its evolutionary technique. The options of the global genetic algorithm are set to be consistent with *csTuner*. Since there is no public *Garvey* implementation available, we re-implement *Garvey* based on [13]. For *Garvey*, we select the optimization of grouping by dimension in [13] and the sampling ratio is also set to 10%. For both *Garvey* and *Artemis*, the number of parameter settings evaluated during one iteration is set to be the same as the population size of the genetic algorithms. This guarantees the fairness of the performance comparison of different auto-tuning methods.

3) Comparison Metrics: The key metrics for determining the effectiveness of auto-tuning methods are the number of iterations and amount of time required to obtain optimal parameter settings. Therefore, we compare *csTuner* against the above methods on two key metrics such as iso-iteration search quality and iso-time search quality [18]. For iso-iteration search quality, all methods are run for a fixed number of iterations. For iso-time quality, all methods are run until a fixed wall-clock time. Specifically, iso-iteration comparison and iso-time comparison are used to verify the performance and speed of auto-tuning methods. To isolate the effects of randomness, we run each method 10 times and present the average results.

B. Results for Auto-tuning Performance

Figure 8 shows the iso-iteration comparison between *csTuner* and the other auto-tuning methods for different stencils. The x-axis and y-axis represent the elapsed iterations and the shortest execution time, respectively. The missing points mean that the parameter settings have been evaluated completely before the current iteration. Overall, *csTuner* has a better starting point and converges faster than other auto-tuning methods. *Garvey* adopts similar system design as *csTuner*, which also performs parameter grouping and search space sampling. Although *Garvey* also converges quickly due to the reduced parameter settings, the random sampling approach limits the stability of its performance. In contrast, *csTuner* uses PMNF to filter out inappropriate parameter settings, thus guaranteeing the high quality of the sampled search space.

OpenTuner converges slowly due to the search of the global parameter space. Although the genetic algorithm helps *OpenTuner* to speed up the convergence, the disadvantage is that it is easy to fall into a local optimum with a small population size. In contrast, the customized genetic algorithm of *csTuner* adopts the iterative auto-tuning based on parameter groups, thus effectively narrowing the search space. Moreover, *csTuner* terminates the search process for a single parameter group before falling into a local optimum using approximation. Note that for certain stencils (e.g., *addsgd6* and *rhs4center*), the *csTuner* converges much faster (after only few iterations) than that of *Artemis* and *OpenTuner* (after 10 iterations). This further proves the effectiveness of the parameter grouping and PMNF function adopted in *csTuner*.

C. Results for Auto-tuning Speed

Figure 9 shows the iso-time comparison between *csTuner* and the other auto-tuning methods for different stencils, where the x-axis represents the elapsed time. Note that the search time per iteration may vary for auto-tuning methods except *OpenTuner*. For example, the number of settings that *Artemis* performs an exhaustive search on a single parameter group may not be divisible by the population size. Besides, since *csTuner* and *Garvey* significantly narrow the search space by sampling, the number of settings during the auto-tuning of a single parameter group is likely to be less than the population size. Therefore, iso-time is a fairer metric than iso-iteration for a limited parameter search space. For most stencils, *csTuner* converges faster than other auto-tuning methods and performs better for parameter settings determined after a fixed search time (100 seconds). The parameter settings determined by *Garvey* achieve the worst performance due to the low quality of the sampled search space.

Since *OpenTuner* does not implement the parameter grouping method, the large search space makes it difficult to converge in a short time. For best performance found under the iso-time evaluation, *Artemis* is closer to *csTuner* on most stencils. This is mainly because the hierarchical auto-tuning based on expert knowledge adopted by *Artemis* is effective to most stencils. In addition, the mutual constraints among optimization parameters of stencil computations significantly reduce the number of valid parameter settings. For a narrow search space, the sampling method adopted in *csTuner* may not show much advantage. Nevertheless, *csTuner* still clearly outperforms *Artemis* for certain stencils (e.g., *cheby* and *addsgd4*). This indicates the parameter grouping algorithms along with statistic methods adopted in *csTuner* can be generalized to various complex stencils. In sum, *csTuner* identifies better parameter settings with higher speed compared to other auto-tuning methods.

D. Applying to other GPU Hardware

To demonstrate the generality of our method, we evaluate *csTuner* on another platform equipped with two NVIDIA Tesla V100 GPUs. Specifically, we re-collect the stencil dataset on the new GPU hardware and use the *csTuner* pipeline to quickly find optimal parameter settings. Figure 10 shows the iso-time performance of various auto-tuning methods normalized to *Garvey* on the V100 GPUs. Again, *csTuner* outperforms other auto-tuning methods for most stencils. For the best performance found in iso-time evaluation, *csTuner* achieves an average speedup of $1.7\times$, $1.2\times$ and $1.3\times$ over *Garvey*, *Artemis* and *OpenTuner* respectively. *Garvey* performs the worst for

Authorized licensed use limited to: BEIHANG UNIVERSITY. Downloaded on December 20,2021 at 02:33:54 UTC from IEEE Xplore. Restrictions apply.

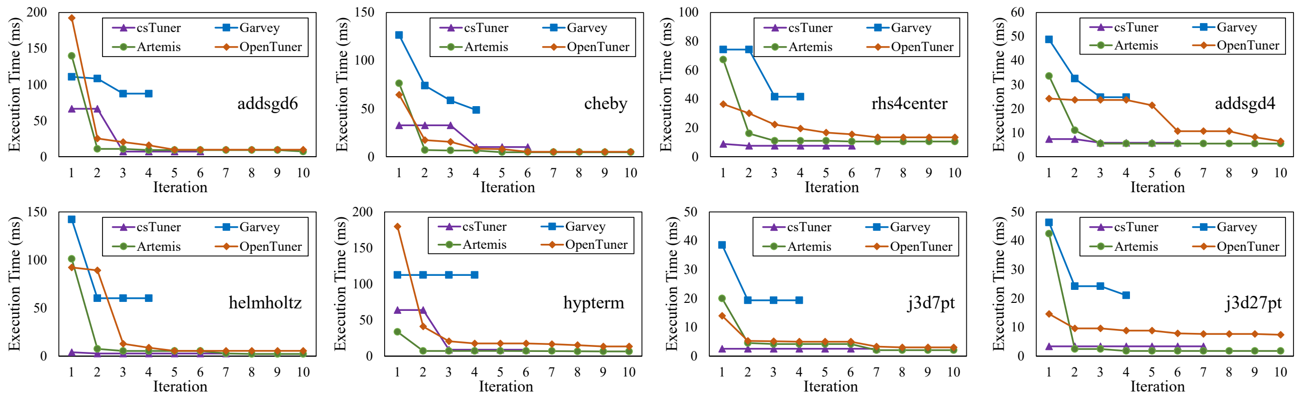

Fig. 8. Iso-iteration comparison of various auto-tuning methods compared to *csTuner*.

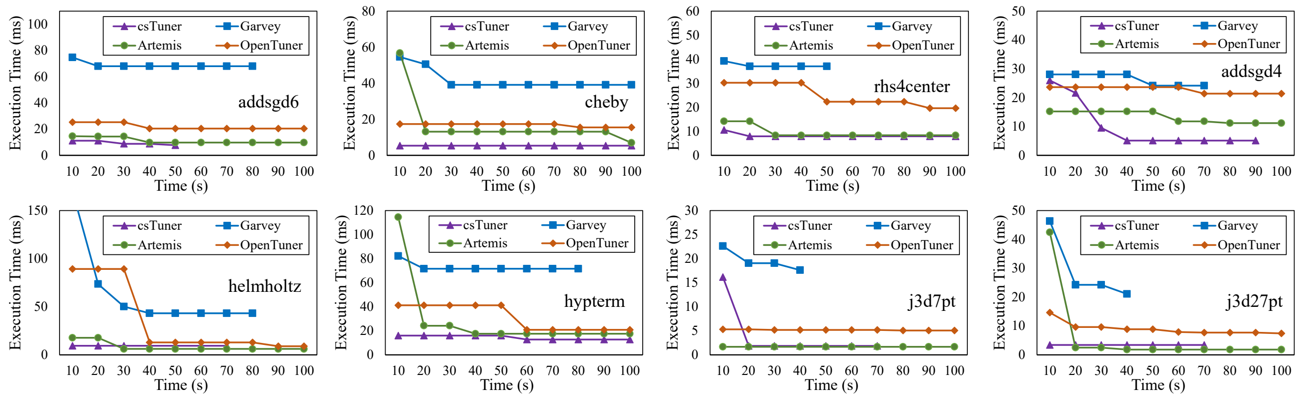

Fig. 9. Iso-time comparison of various auto-tuning methods compared to *csTuner*.

most stencils due to the sampled low-quality parameter settings. On the contrary, it is difficult for *OpenTuner* to find optimal parameter settings in a short time due to the large search space.

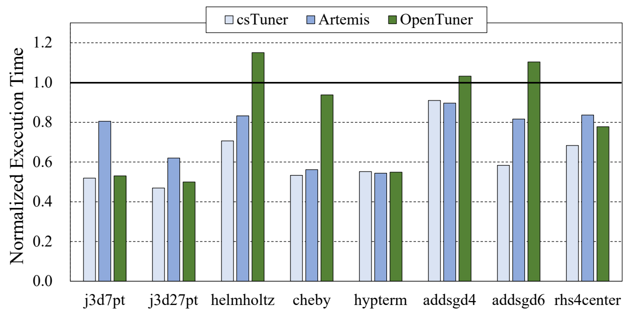

Fig. 10. Iso-time performance of various auto-tuning methods normalized to *Garvey* on a V100 $\times$ 2 GPU platform.

The *csTuner* achieves comparable performance as *Artemis* on three stencils (i.e., *cheby*, *hypterm* and *addsgd4*), and significantly outperforms *Artemis* for others. Note that the statistic-based parameter grouping and PMNF-guided search space sampling adopted in *csTuner* does not require any expert knowledge. Therefore, *csTuner* can be easily applied to various complex stencils and hardware platforms with stable auto-tuning quality.

E. Parameter Sensitivity Analysis

Figure 11 shows the iso-time performance of *csTuner* with different sampling ratios, where the x-axis represents the sampling ratio ranging from 5% to 50% with a stride of 5%. A good sampling ratio can achieve the balance between search range and search speed. A smaller sampling ratio completes the search process more quickly, but the limited search range is unlikely to obtain optimal parameter settings. In contrast, a larger sampling ratio is more promising to obtain optimal parameter settings, yet with a longer search time. As seen, the worst performance is achieved for half of the stencils with 5% sampling ratio. Besides, The sampling ratio with the optimal iso-time performance is usually between 5% and 50%.

For most stencils, the iso-time performance with the sampling ratio using the middle range of settings (e.g., 15%$\sim$40%) remains stable. This proves that the PMNF model adopted in *csTuner* filters out as many poorly performing settings as possible during the sampling process. In addition, it can be observed that high iso-time performance is still achieved when setting the sampling ratio to 50%. This is mainly because

Authorized licensed use limited to: BEIHANG UNIVERSITY. Downloaded on December 20,2021 at 02:33:54 UTC from IEEE Xplore. Restrictions apply.

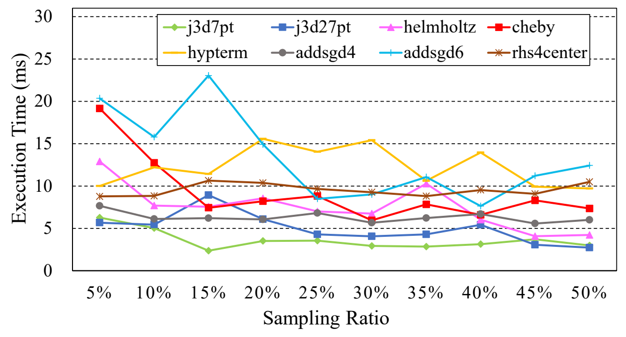

Fig. 11. Iso-time performance of *csTuner* with different sampling ratios.

the valid search space of stencil computations is limited by the constraints among optimization parameters. Although the constrained search space cannot fully exploit the superiority of the guided sampling adopted in *csTuner*, we believe that *csTuner* is more advantageous when applied to a much larger parameter search space for stencil computations.

F. Overhead Analysis

For each stencil, it takes less than five minutes to use *Nsight* to collect the GPU metrics included in the stencil dataset. The stencil dataset determines the cost of metric collection regardless of the size of the parameter search space. Therefore, for complex stencils with more optimizations proposed in the future, the cost of obtaining the stencil dataset can be further amortized. Since the metric collection only needs to be done offline once, we do not consider it in the overhead analysis of online auto-tuning. The overhead of the search process in *csTuner* can be divided into two parts including pre-processing and searching itself. The pre-processing can be further divided into parameter grouping, search space sampling and code generation. The parameter grouping mainly consists of the correlation calculation based on CVs and the grouping algorithm based on *Deque* structure. The search space sampling consists of metric combination, generating PMNF functions, fitting regression models and filtering out non-compliant parameter settings. The code generation writes the sampled parameter settings into CUDA kernels for the subsequent auto-tuning process.

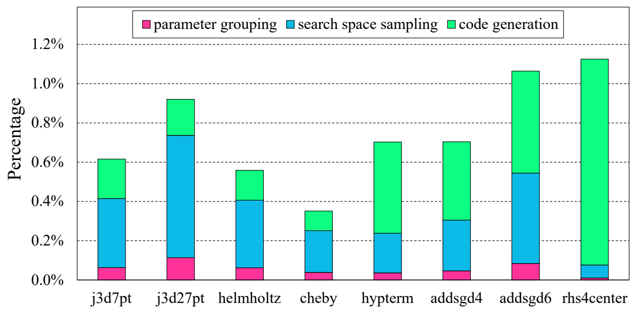

Fig. 12. Performance breakdown of the pre-processing in *csTuner* normalized to the search process.

Since the parameter searching dominates the search process (more than 98%), to better illustrate the overhead of pre-processing, we show the performance breakdown of the pre-processing normalized to the search process in Figure 12. The search process performs iterative auto-tuning at parameter group basis. When the size of a single parameter group is larger than the population size, *csTuner* adopts the genetic algorithm with approximation. Otherwise, *csTuner* degenerates to the exhaustive search. As seen, the pre-processing overhead of *csTuner* is negligible compared to the entire search process, occupying only 0.76% of the search time on average. Specifically, the overhead of code generation increases with the complexity of stencil patterns. However, even for the stencil with the highest FLOPs (i.e., *rhs4center*), the code generation overhead is only 1.04% compared to the entire search time. Similarly, the pre-processing overhead of *csTuner* can be further amortized for the larger search space with more stencil optimizations proposed in the future.

VI. Related Work

Stencil DSLs and Optimizations. Based on the regular patterns of stencil computation, existing research works exploit the integration of optimization schemes into DSLs to achieve automatic code transformation and optimization [8], [15], [17], [24], [25], [30], [33]–[35], [38], [51]. *PATUS* [8] allowed users to define stencils by a C-like syntax and chose the predefined or custom strategies for optimization. *Physis* [24] translated user-written stencil code into scalable implementation for GPU-equipped cluster. *Forma* [34] proposed a DSL for image processing application with stencil operations. Grosser *et al.* [15] presented a novel hybrid tiling method that combined hexagonal tiling and wavefront tiling on GPUs. Hagedorn *et al.* [17] explored how to use *LIFT* primitives to implement stencil codes and optimizations such as tiling. Rawat *et al.* [37] presented a stencil framework based on *STENCILGEN* DSL and optimized the performance by blocking, streaming and resource management using DAG. Matsumura *et al.* [25] proposed a C-based stencil framework named *AN5D*, which implemented high-degree temporal blocking and spatial blocking. *AN5D* also adopted low-level optimizations to reduce the usage of shared memory and registers. *GOPipe* [30] automatically pipelined and dynamically scheduled stencil execution on GPUs. Although the above DSLs significantly reduce engineering efforts, they lack effective support for auto-tuning. *csTuner* can be integrated into these DSLs and quickly obtain the optimal parameter settings for the target optimization schemes.

Performance Auto-tuning on GPUs. Since identifying the optimal kernel variants is extremely challenging for both programmers and code generators, a large amount of research works focus on the auto-tuning of target problems on GPUs [11], [20]–[22], [32], [45], [47], [50]. Kurzak *et al.* [20] proposed heuristic auto-tuning to prune the search space and generate the fastest code variant of matrix multiplication kernels. Li *et al.* [21] resolved the conflict between concurrency and register usage by precomputing the critical

Authorized licensed use limited to: BEIHANG UNIVERSITY. Downloaded on December 20,2021 at 02:33:54 UTC from IEEE Xplore. Restrictions apply.

points and selecting the global optimum. Lim *et al.* [22] proposed a static analyzer tool to estimate the runtime behavior of kernels and tune the applications without running programs. Dongarra *et al.* [11] presented an automated exploration of the implementation space for batched Cholesky factorization to maximize the hardware occupancy. Pfaffe *et al.* [32] integrated hierarchical online auto-tuning with polyhedral parallelization to reduce search complexity and increase convergence speed. *Ansor* [50] sampled optimization combinations from a large hierarchical search space and utilized an evolutionary search with a learned cost model to fine-tune the tensor programs. The above works are orthogonal to this paper that targets the auto-tuning of stencil kernels. In turn, *csTuner* can be extended to other target programs due to its versatility.

VII. Conclusion

In this paper, we propose a scalable auto-tuning framework *csTuner*, which quickly identifies optimal parameter settings in a given optimization space for complex stencils on GPUs. The *csTuner* leverages a set of statistic and machine learning methods to generate parameter groups and guide the search space sampling. After that, *csTuner* utilizes iterative auto-tuning based on parameter groups to further narrow the search space. Finally, we re-design the genetic algorithm to reduce the cost of evolutionary search using approximation. The experimental results show that *csTuner* can identify high-performance parameter settings with higher auto-tuning speed.

For future work, we would like to extend *csTuner* to support auto-tuning of more optimization techniques for complex stencils. In addition, we would like to apply *csTuner* to other domains with even larger search space (e.g., tensor optimizations in deep learning). We would also like to extend *csTuner* to support other hardware such as CPU and accelerator. To achieve that, we only need to adjust the optimization space according to the target hardware and then parameterize the optimization space into tuning options.

Acknowledgements

This work was supported by National Key Research and Development Program of China (No. 2020YFB1506703), National Natural Science Foundation of China (No. 62072018) and State Key Laboratory of Software Development Environment (No. SKLSDE-2021ZX-06). Hailong Yang is the corresponding author.

References

[1] Seismic wave modelling (sw4) - computational infrastructure for geodynamics. https://geodynamics.org/cig/software/sw4 (2013)

[2] Abdi, H.: Coefficient of variation. Encyclopedia of research design **1**, 169–171 (2010)

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

[4] Benesty, J., Chen, J., Huang, Y., Cohen, I.: Pearson correlation coefficient. In: Noise reduction in speech processing, pp. 1–4. Springer (2009)

[5] Breiman, L.: Random forests. Machine learning **45**(1), 5–32 (2001)

[6] Calotoiu, A., Beckinsale, D., Earl, C.W., Hoefler, T., Karlin, I., Schulz, M., Wolf, F.: Fast multi-parameter performance modeling. In: 2016 IEEE International Conference on Cluster Computing (CLUSTER). pp. 172–181. IEEE (2016)

[7] Chinn, S.: A simple method for converting an odds ratio to effect size for use in meta-analysis. Statistics in medicine **19**(22), 3127–3131 (2000)

[8] Christen, M., Schenk, O., Burkhart, H.: Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In: 2011 IEEE International Parallel & Distributed Processing Symposium. pp. 676–687. IEEE (2011)

[9] Copik, M., Calotoiu, A., Grosser, T., Wicki, N., Wolf, F., Hoefler, T.: Extracting clean performance models from tainted programs. In: Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. pp. 403–417 (2021)

[10] Dong, W., Xie, Z., Kestor, G., Li, D.: Smart-pgsim: using neural network to accelerate ac-opf power grid simulation. arXiv preprint arXiv:2008.11827 (2020)

[11] Dongarra, J., Gates, M., Kurzak, J., Luszczek, P., Tsai, Y.M.: Autotuning numerical dense linear algebra for batched computation with gpu hardware accelerators. Proceedings of the IEEE **106**(11), 2040–2055 (2018)

[12] Gamell, M., Teranishi, K., Heroux, M.A., Mayo, J., Kolla, H., Chen, J., Parashar, M.: Local recovery and failure masking for stencil-based applications at extreme scales. In: SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 1–12. IEEE (2015)

[13] Garvey, J.D., Abdelrahman, T.S.: Automatic performance tuning of stencil computations on gpus. In: 2015 44th International Conference on Parallel Processing. pp. 300–309. IEEE (2015)

[14] Goldberg, D.E., Holland, J.H.: Genetic algorithms and machine learning (1988)

[15] Grosser, T., Cohen, A., Holewinski, J., Sadayappan, P., Verdoolaege, S.: Hybrid hexagonal/classical tiling for gpus. In: Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization. pp. 66–75 (2014)

[16] Guide, D.: Cuda c programming guide. NVIDIA, April (2021)

[17] Hagedorn, B., Stoltzfus, L., Steuwer, M., Gorlatch, S., Dubach, C.: High performance stencil code generation with lift. In: Proceedings of the 2018 International Symposium on Code Generation and Optimization. pp. 100–112 (2018)

[18] Hegde, K., Tsai, P.A., Huang, S., Chandra, V., Parashar, A., Fletcher, C.W.: Mind mappings: enabling efficient algorithm-accelerator mapping space search. In: Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 943–958 (2021)

[19] Kashyap, B.R.: The double-ended queue with bulk service and limited waiting space. Operations Research **14**(5), 822–834 (1966)

[20] Kurzak, J., Tomov, S., Dongarra, J.: Autotuning gemm kernels for the fermi gpu. IEEE Transactions on Parallel and Distributed Systems **23**(11), 2045–2057 (2012)

[21] Li, A., Song, S.L., Kumar, A., Zhang, E.Z., Chavarría-Miranda, D., Corporaal, H.: Critical points based register-concurrency autotuning for gpus. In: 2016 Design, Automation & Test in Europe Conference & Exhibition (DATE). pp. 1273–1278. IEEE (2016)

[22] Lim, R., Norris, B., Malony, A.: Autotuning gpu kernels via static and predictive analysis. In: 2017 46th International Conference on Parallel Processing (ICPP). pp. 523–532. IEEE (2017)

[23] Luo, Y., Tan, G., Mo, Z., Sun, N.: Fast: A fast stencil autotuning framework based on an optimal-solution space model. In: Proceedings of the 29th ACM on International Conference on Supercomputing. pp. 187–196 (2015)

[24] Maruyama, N., Nomura, T., Sato, K., Matsuoka, S.: Physis: an implicitly parallel programming model for stencil computations on large-scale gpu-accelerated supercomputers. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 1–12 (2011)

[25] Matsumura, K., Zohouri, H.R., Wahib, M., Endo, T., Matsuoka, S.: An5d: automated stencil framework for high-degree temporal blocking on gpus. In: Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization. pp. 199–211 (2020)

[26] Mullapudi, R.T., Vasista, V., Bondhugula, U.: Polymage: Automatic optimization for image processing pipelines. ACM SIGARCH Computer Architecture News **43**(1), 429–443 (2015)

[27] NVIDIA: V100 gpu architecture. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf (2017)

Authorized licensed use limited to: BEIHANG UNIVERSITY. Downloaded on December 20,2021 at 02:33:54 UTC from IEEE Xplore. Restrictions apply.

[28] NVIDIA: Nvidia a100 tensor core gpu architecture. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf (2020)

[29] NVIDIA: Nvidia nsight compute command line interface (cli) user manual (2021)

[30] Oh, C., Zheng, Z., Shen, X., Zhai, J., Yi, Y.: Gopipe: a granularity-oblivious programming framework for pipelined stencil executions on gpu. In: Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques. pp. 43–54 (2020)

[31] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., et al.: Scikit-learn: Machine learning in python. the Journal of machine Learning research **12**, 2825–2830 (2011)

[32] Pfaffe, P., Grosser, T., Tillmann, M.: Efficient hierarchical online-autotuning: a case study on polyhedral accelerator mapping. In: Proceedings of the ACM International Conference on Supercomputing. pp. 354–366 (2019)

[33] Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., Amarasinghe, S.: Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. Acm Sigplan Notices **48**(6), 519–530 (2013)

[34] Ravishankar, M., Holewinski, J., Grover, V.: Forma: A dsl for image processing applications to target gpus and multi-core cpus. In: Proceedings of the 8th Workshop on General Purpose Processing using GPUs. pp. 109–120 (2015)

[35] Rawat, P.S., Hong, C., Ravishankar, M., Grover, V., Pouchet, L.N., Sadayappan, P.: Effective resource management for enhancing performance of 2d and 3d stencils on gpus. In: Proceedings of the 9th Annual Workshop on General Purpose Processing using Graphics Processing Unit. pp. 92–102 (2016)

[36] Rawat, P.S., Rastello, F., Sukumaran-Rajam, A., Pouchet, L.N., Rountev, A., Sadayappan, P.: Register optimizations for stencils on gpus. In: Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. pp. 168–182 (2018)

[37] Rawat, P.S., Vaidya, M., Sukumaran-Rajam, A., Ravishankar, M., Grover, V., Rountev, A., Pouchet, L.N., Sadayappan, P.: Domain-specific optimization and generation of high-performance gpu code for stencil computations. Proceedings of the IEEE **106**(11), 1902–1920 (2018)

[38] Rawat, P.S., Vaidya, M., Sukumaran-Rajam, A., Rountev, A., Pouchet, L.N., Sadayappan, P.: On optimizing complex stencils on gpus. In: 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS). pp. 641–652. IEEE (2019)

[39] Ritter, M., Calotoiu, A., Rinke, S., Reimann, T., Hoefler, T., Wolf, F.: Learning cost-effective sampling strategies for empirical performance modeling. In: 2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS). pp. 884–895. IEEE (2020)

[40] Sai, R., Mellor-Crummey, J., Meng, X., Araya-Polo, M., Meng, J.: Accelerating high-order stencils on gpus. In: 2020 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS). pp. 86–108. IEEE (2020)

[41] Sano, K., Hatsuda, Y., Yamamoto, S.: Multi-fpga accelerator for scalable stencil computation with constant memory bandwidth. IEEE Transactions on Parallel and Distributed Systems **25**(3), 695–705 (2013)

[42] Shudler, S., Calotoiu, A., Hoefler, T., Strube, A., Wolf, F.: Exascaling your library: Will your implementation meet your expectations? In: Proceedings of the 29th ACM on International Conference on Supercomputing. pp. 165–175 (2015)

[43] Spiess, A.N., Neumeyer, N.: An evaluation of r 2 as an inadequate measure for nonlinear models in pharmacological and biochemical research: a monte carlo approach. BMC pharmacology **10**(1), 1–11 (2010)

[44] Stock, K., Kong, M., Grosser, T., Pouchet, L.N., Rastello, F., Ramanujam, J., Sadayappan, P.: A framework for enhancing data reuse via associative reordering. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 65–76 (2014)

[45] Tillmann, M., Karcher, T., Dachsbacher, C., Tichy, W.F.: Application-independent autotuning for gpus. In: PARCO. pp. 626–635 (2013)

[46] Vasilache, N., Zinenko, O., Theodoridis, T., Goyal, P., DeVito, Z., Moses, W.S., Verdoolaege, S., Adams, A., Cohen, A.: Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. arXiv preprint arXiv:1802.04730 (2018)

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

[48] Xiao, Z., Liu, X., Xu, J., Sun, Q., Gan, L.: Highly scalable parallel genetic algorithm on sunway many-core processors. Future Generation Computer Systems **114**, 679–691 (2021)

[49] Yount, C., Tobin, J., Breuer, A., Duran, A.: Yask—yet another stencil kernel: A framework for hpc stencil code-generation and tuning. In: 2016 Sixth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC). pp. 30–39. IEEE (2016)

[50] Zheng, L., Jia, C., Sun, M., Wu, Z., Yu, C.H., Haj-Ali, A., Wang, Y., Yang, J., Zhuo, D., Sen, K., et al.: Ansor: Generating high-performance tensor programs for deep learning. In: 14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20). pp. 863–879 (2020)

[51] Zheng, Z., Oh, C., Zhai, J., Shen, X., Yi, Y., Chen, W.: Versapipe: a versatile programming framework for pipelined computing on gpu. In: 2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). pp. 587–599. IEEE (2017)

[52] Zohouri, H.R., Podobas, A., Matsuoka, S.: High-performance high-order stencil computation on fpgas using opencl. In: 2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). pp. 123–130. IEEE (2018)

Authorized licensed use limited to: BEIHANG UNIVERSITY. Downloaded on December 20,2021 at 02:33:54 UTC from IEEE Xplore. Restrictions apply.