StencilMART: Predicting Optimization Selection for Stencil Computations across GPUs

Qingxiao Sun[1,2], Yi Liu[2], Hailong Yang[1,2], Zhonghui Jiang[2], 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
{qingxiaosun,yi.liu,hailong.yang,jiangzhh,07680,depeiq}@buaa.edu.cn

2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS) | 978-1-6654-8106-9/22/$31.00 ©2022 IEEE | DOI: 10.1109/IPDPS53621.2022.00090

***Abstract*—Stencil computations are widely used in high performance computing (HPC) applications. Many HPC platforms utilize the high computation capability of GPUs to accelerate stencil computations. In recent years, stencils have become more diverse in terms of stencil order, memory accesses and computation patterns. To adapt diverse stencils to GPUs, a variety of optimization techniques have been proposed such as streaming and retiming. However, due to the diversity of stencil patterns and GPU architectures, no single optimization technique fits all stencils. Besides, it is challenging to choose the most cost-efficient GPU for accelerating target stencils. To address the above problems, we propose *StencilMART*, an automatic optimization selection framework that predicts the best optimization combination and execution time under a certain parameter setting for stencils on GPUs. Specifically, the *StencilMART* represents the stencil patterns as binary tensors and neighboring features through tensor assignment and feature extraction. In addition, the *StencilMART* implements various machine learning methods such as classification and regression that utilize stencil representation and hardware characteristics for execution time prediction. The experiment results show that the *StencilMART* can achieve accurate optimization selection and performance prediction for various stencils across GPUs.**

***Index Terms*—Stencil Computation, GPU, Optimization Strategies, Performance Prediction, Machine Learning**

I. Introduction

Stencil computation is one of the most adopted computation patterns in scientific applications. Stencil computations appear in many domains such as cellular automata [21], physical simulation [7] and image processing [16]. A stencil computation sweeps a computation 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 [17], [19].

In recent years, stencil computations have become more diverse in terms of stencil order, data accesses and computing patterns [10], [20]. The diverse stencils tend to have abundant parallelism, which makes GPU a good candidate for performance acceleration. However, due to the complexity of GPU architecture, the programmers must ensure memory coalescing, reduce thread divergence and trade off between parallelism and resource utilization when optimizing stencils on GPU. Many optimization techniques based on streaming and tiling [9], [14] have been proposed to adapt to the high computation capability and limited memory bandwidth of GPU architecture. However, no single optimization technique fits all stencils due to the diversity of stencil patterns.

Stencil domain-specific languages (DSLs) explore the automatic code generation with the integration of various optimization techniques [12], [15], [17], [18]. Although the DSLs are effective in improving stencil performance, it is difficult to evaluate the performance impact of individual optimization techniques within a particular optimization combination (OC). In addition, stencil auto-tuning frameworks [8], [25] have been proposed to determine the optimal parameter settings for specific OCs. However, whether an OC can generate high performant stencil code depends on the target stencil and GPU architecture. Conducting a time-consuming parameter search for sub-optimal OCs will significantly deteriorate the effectiveness of auto-tuning mechanisms.

Performance prediction is often used to study the impact of optimizations on stencil computation [6], [13]. Since no actual execution is required, performance prediction can efficiently reduce the search cost involved in stencil auto-tuning. Furthermore, performance prediction across architectures can help users decide the most cost-efficient GPU for accelerating target stencils. The OC selection and performance prediction can be naively considered machine learning problems such as classification and regression. However, it is challenging to extract the effective features representing stencil patterns and GPU properties to solve the above problems. Even with the features available, utilizing these features to accurately predict stencil performance under OCs remains unresolved.

To address the above problems, we propose an automatic optimization selection framework *StencilMART*, which can predict the best OC for an input stencil running on a particular GPU. Moreover, the *StencilMART* supports cross-architecture performance prediction that obtains the execution time for stencils without accessing the target GPU. The *StencilMART* first generates random stencil programs that satisfy specific stencil patterns with neighbor accesses. After that, the *StencilMART* transforms the stencil access patterns into binary tensors and neighboring features. Finally, the *StencilMART* implements the prediction model with various machine learning methods, where the input includes stencil representation and hardware characteristics. To the best of our knowledge, this is the first work that targets the optimization selection and cross-architecture performance prediction for stencil computation.

Specifically, this paper makes the following contributions:

1530-2075/22/$31.00 ©2022 IEEE
DOI 10.1109/IPDPS53621.2022.00090

- We comprehensively analyze the impact of optimization selection on the stencil performance due to the diverse access patterns and hardware characteristics. We also discuss how cross-architecture performance prediction can contribute to cost-efficient GPU selection.
- We propose a stencil transformation mechanism that represents stencils as binary tensors and neighboring features. In addition, we offer a random stencil generator that outputs a variety of stencils that satisfy specific stencil patterns with neighbor accesses.
- We design and implement machine learning-based classification and regression mechanisms for optimization selection and performance prediction. The customized mechanisms utilize stencil representation and GPU hardware characteristics for accurate prediction.
- We develop an automatic optimization framework *StencilMART* that accurately predicts the best OC or the performance under a specific parameter setting for stencils across different GPUs. The experiment results show that the *StencilMART* can achieve accurate optimization selection and performance prediction for various stencils.

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 *StencilMART* design. Section V presents the evaluation results of *StencilMART*. Section VI discusses the related work, and Section VII concludes this paper.

II. Background

A. GPU Architecture and Developing Trend

The NVIDIA GPU consists of dozens to hundreds of Streaming Multiprocessors (SMs) depending on the GPU generation. 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*. Multiple warps are further 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.

In the meanwhile, we notice the changes in computing resources among different NVIDIA GPU generations [26]. We observe that the architecture development trend of GPUs is as follows: *1) The number of SMs keeps growing*. The latest Volta, Turing and Ampere architectures are equipped with 80 SMs (Volta V100), 72 SMs (Turing TU102) and 108 SMs (Ampere A100), respectively. *2) The intra-SM resources remain almost constant.* The intra-SM resources include register file, shared memory, L1 cache, and GPU cores. For memory-intensive applications such as stencils, the GPU equipped with more computing resources does not offer significant performance benefits [25]. In this case, it is not worth purchasing or renting the most powerful GPU available considering the cost efficiency (i.e., performance per dollar).

B. Optimizations for Stencil Computation

Widespread attention has been attracted to accelerate stencil computation on GPU due to its high computation capability [9], [15], [19], [20]. We briefly discuss the optimizations of stencil computation on GPUs (Table I).

TABLE I
THE OPTIMIZATIONS OF STENCIL COMPUTATION ON GPUS.

No.	Optimzation	Abbreviation	Constraint
1	Streaming	ST	–
2	Block Merging	BM	Not valid when CM enabled.
3	Cyclic Merging	CM	Not valid when BM enabled.
4	Retiming	RT	Only valid when ST enabled.
5	Prefetching	PR	Only valid when ST enabled.
6	Temporal Blockling	TB	–

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 [15]. 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 [20] divides the streaming dimension into tiles, where the TBs traverse the streaming dimension in parallel at the granularity of tiles. Meanwhile, loop unrolling has 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 [8]. 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 the current iteration, the data located in the shared memory is shifted to continue the computation for the next iteration. Due to the concurrent execution of massive threads on GPU, a synchronization barrier has to be performed between adjacent iterations to ensure the correctness of the results. The synchronization can cause serialization between kernels and thereby deteriorate performance. Prefetching [20] 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 [22] 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 [20]. 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.

5) Temporal Blocking: Even though stencil computation has data dependency across time steps, the dependency range of one point is limited by the stencil pattern and the number of time steps elapsed since the point's last update [15]. Temporal blocking exploits the hidden temporal locality by fusing time steps and avoiding global memory accesses. The dependency along the time dimension is resolved by redundantly loading from adjacent blocks. However, temporal blocking may incur performance degradation for register-constrained stencils.

The above optimizations can be combined under certain constraints (Table I) to improve performance further. However, the optimization combination should be carefully selected to suit the target stencil and hardware architecture.

C. Limitations of Stencil Auto-tuning Mechanisms

Due to the diversity of stencil patterns, any optimization has to be fine-tuned to maximize its performance. Stencil Domain-Specific Languages (DSLs) expose performance-related parameters to auto-tuning mechanisms integrated into their frameworks [17], [18], [20]. For instance, *Halide* [18] applies stochastic search to find good pipeline schedules automatically. *Artemis* [20] tunes the computation for high-impact optimizations first and then selects a few high-performance candidates. *GoPipe* [17] finds the best task granularity for each stage of a pipelined box stencil (e.g., image convolution). Since the auto-tuning mechanisms are customized for particular stencil DSLs, they have poor scalability to evaluate more optimizations during parameter tuning.

To overcome the limitation, several works have considered speeding up the auto-tuning performance of stencil computation [8], [25]. *Garvey* [8] groups optimization parameters based on experience. After that, *Garvey* exhaustively searches for the parameter settings of each group with random sampling enabled. *csTuner* [25] leverages statistics and machine learning methods to generate parameter groups and sampled settings. Then, *csTuner* re-designs the genetic algorithm with approximation to reduce the search time. As illustrated above, the stencil auto-tuning mechanisms usually perform parameter search for pre-specified optimizations or their combinations. However, this does not indicate whether the optimization combination is suitable for target stencils.

In addition to parameter auto-tuning, performance prediction is utilized to study the impact of optimizations on stencil computation [6], [13]. Martínez *et al.* [13] feeds the kernel configurations and hardware counters to the support vector machine (SVM) to predict the GFLOPS and execution time of stencils. Cosenza *et al.* [6] utilized ordinal regression to predict the performance ranking of stencil code variants and the quality of the obtained ranking is evaluated by Kendall coefficients. However, the above works are only implemented on multi-core CPUs and do not take cross-architecture performance predictions into account. Due to the high price of a GPU, the users would ideally want to know its performance on stencil computations before spending money to get access to the GPU [27].

III. Motivation

We make four main observations by comparing the performance of optimization combinations (OCs), where any combination of optimizations under the constraints (Table I) is taken into consideration. The representative stencils we select cover a variety of shapes (star, box and cross), orders (1-4) and dimensions (2-D and 3-D). The input grids of 2-D and 3-D stencils are 8192^2 and 512^3, respectively. We conduct experiments on different NVIDIA GPU generations, including 2080 Ti, P100, V100 and A100 (Table III). For each OC, the parameter setting with the shortest execution time is selected to ensure the fairness of performance comparison.

A. Performance Gap among OCs

Since the insights on different GPU architectures are similar, we only explain the performance results on V100. Figure 1 shows the performance gap between the worst OC and the best OC for each stencil. Note that there are some cases where OC crashes under certain stencils, which are not reflected in the figure. For example, temporal blocking fails to be applied for 3-D order-4 stencils without streaming enabled. This is because streaming can effectively avoid intra-SM resource spilling that may be caused by temporal blocking. As seen, the performance gap among OCs is significant, where the best OC achieves an average speedup of 9.95× over the worst OC. In addition, for stencils of the same shape, a higher dimension or order usually means a larger performance gap. Therefore, the OC should be carefully selected to ensure successful execution while improving performance, especially for complex stencils. However, manual determining the high-performance OCs requires considerable engineering efforts. This motivates the mechanism for automatically selecting the high-performance OCs for stencil computations.

B. Distribution of Best OCs for Stencils

Figure 2 shows the distribution of best OCs for stencils on GPUs. The missing bars indicate that the OC does not achieve the best performance for any stencil, such as temporal blocking without streaming (i.e., TB, TB_CM and TB_BM) under all architectures. As seen, the OCs with streaming perform better for most stencils. This is because streaming improves data reuse and reduces computation redundancy. Besides, more optimizations are valid with streaming enabled such as prefetching and retiming, which brings more opportunities for performance improvement. It can also be observed that the distribution of the best OCs is relatively even. This indicates

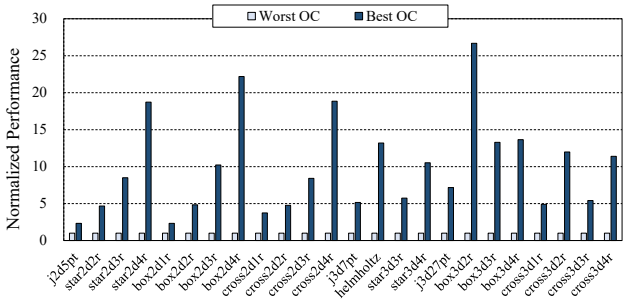


Fig. 1. The performance of the best OC of each stencil normalized to its worst OC on V100.

that the best OC changes across stencils, there is no single OC fits for all. The reason is that the computation pattern of stencils has a significant impact on the effectiveness of optimizations. The above observation motivates us to predict the best OC for stencils through machine learning methods.

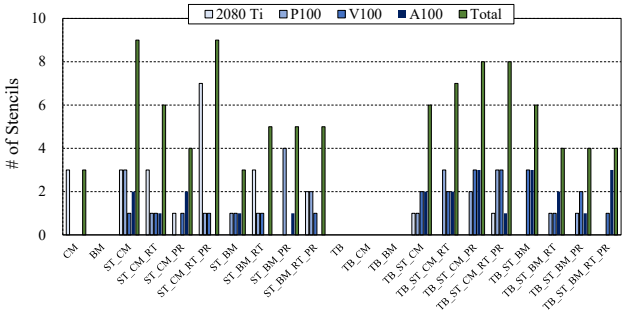


Fig. 2. Comparison of the number of stencils that each OC achieves the best performance on GPUs.

C. Pairwise Correlation of OCs

We define that high correlation corresponds to the small difference in performance achieved by pairwise OCs under the same stencil. Further, this indicates that the effect of pairwise OCs on stencil computation is similar. We use the Pearson correlation coefficient (PCC) [4] to quantify the correlation between pairwise OCs. The closer to 1 the absolute value of PCC is, the stronger the correlation of the OC pair is. Figure 3 shows the value distribution of top-100 PCCs achieved by pairwise OCs on GPUs. As seen, the value distribution of top-100 PCCs is close, and the intersection of pairwise OCs under all architectures accounts for 28% of the total. This indicates that the influence of certain OCs on stencil computation is general among architectures. Therefore, if the selection of the best OC is analogous to a classification problem, the OC pairs in the intersection can be grouped to reduce the classes to be predicted. After that, we can select the OC that obtains the best performance under more cases (Figure 2) from each OC group as the prediction target.

D. Performance Comparison across Architectures

Figure 4 shows the performance comparison of various GPU architectures normalized to 2080 Ti. The performance of each stencil is taken from the best OC obtained on the target architecture. As seen, the performance of stencils is not proportional to the number of computing cores. For example, the performance of $cross2d1r$ on the desktop-class GPU (2080 Ti) is even better than that on V100. In addition, the most "powerful" GPU (A100) is not always the best for stencil computations. For example, $box3d3r$ and $box3d4r$ achieve the best performance on V100 instead of A100. More importantly, cost-efficiency leads to selecting a different GPU for stencil computation. Therefore, it is necessary to know the performance of stencils on the destination GPU before spending money to get access. This motivates us to scale the execution time of stencil computations measured on one GPU (origin GPU) to another (destination GPU).

IV. StencilMART Methodology

A. Design Overview

In this section, we propose an automatic stencil optimization selection framework *StencilMART* that can predict the best optimization combination (OC) for stencil computation on GPUs. Moreover, the *StencilMART* supports cross-architecture performance prediction, which can obtain the execution time for stencil computation without accessing the target GPU. As shown in Figure 5, the *StencilMART* consists of four important components including random stencil generator (Section IV-B), stencil representation (Section IV-C), classification mechanisms for OC selection (Section IV-D) and regression mechanisms for performance prediction (Section IV-E). The random stencil generator outputs a variety of stencil programs for training data collection. The stencil representation transforms the access patterns of stencils into assigned tensors and neighboring features. Machine learning-based classification and regression mechanisms effectively determine the best OC and predict the cross-architecture performance.

Figure 5 illustrates the holistic pipeline of *StencilMART*. The access pattern of each generated stencil is transformed into the sparse tensor and neighboring features through tensor assignment and feature extraction. In addition, the performance of each stencil input is profiled with different OCs on the target GPU, and the OC with the best performance is labeled. Specifically, the *StencilMART* randomly searches the parameter settings under each OC and selects the shortest execution time for performance comparison. Each parameter setting and its corresponding execution time are also stored in the stencil dataset for cross-architecture performance prediction. The profiled dataset is used to train the classification or regression model for OC selection or performance prediction on the target GPU, respectively.

The *StencilMART* implements various mechanisms based on both traditional machine learning (e.g., *GBDT*) and deep learning (e.g., *ConvMLP*), where the users can select the best-performing mechanism according to their own needs. Note

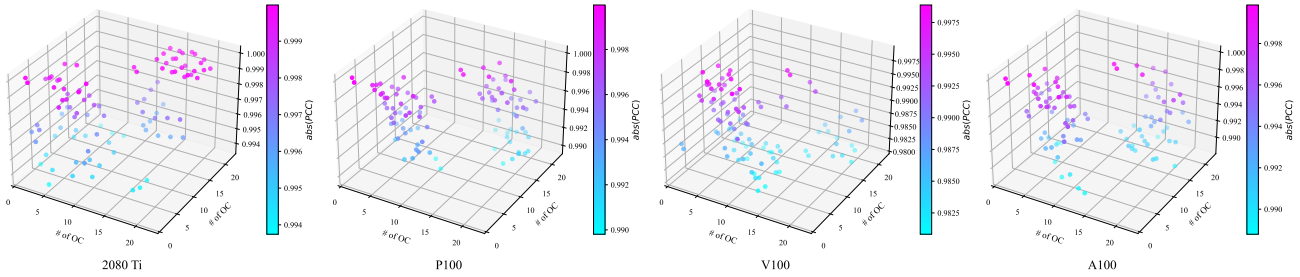


Fig. 3. The value distribution of top-100 PCCs achieved by pairwise OCs on GPUs.

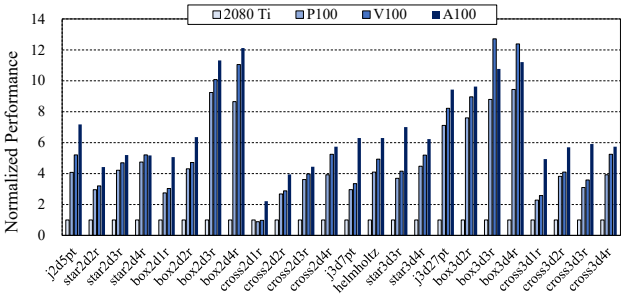


Fig. 4. The best performance of stencils under each GPU architecture normalized to 2080 Ti.

that prediction of the best OC and the performance across architectures exhibits as two different tasks, which do not affect the accuracy of each other.

B. Random Stencil Generation

Inspired by [6], we represent the access pattern of a stencil with any dimension or shape as a binary sparse tensor. Figure 6 shows an example of transforming a 2-D stencil with a maximum order of 4 into a sparse tensor with a size of 9×9, where the neighbor points accessed and the central point are assigned a value of 1. The higher dimensional stencils can be analogized in the same way. From the above, we consider sampling access points in the tensor space to generate random stencils for model training. In this regard, the most straightforward solution is to sample within the index range of a fixed-sized tensor randomly. However, this solution does not conform to the computation characteristic of stencils that processes the neighbors of each point to update its value.

Algorithm 1 illustrates the process of random stencil generator that meets the computation pattern with neighbor accesses, where the input is the stencil order and the output is a list of neighbor points accessed by a stencil. Specifically, we iteratively sample access points from low-order neighbors to high-order neighbors. During each iteration, we randomly sample the higher-order neighbors of the selected points in the previous iteration (Lines 8-14). After that, we store the non-redundant neighbor points in the list and advance to the next iteration (Lines 16-17). The random stencils generated in this way cover the popular stencil shapes (Figure 6) and conform to the neighbor access patterns of stencil computation.

Algorithm 1 Stencil generator that meets neighbor patterns.

1: **Input:** stencil order (N)
2: **Output:** The list of neighbor points accessed ($npList$)
3: **for** $order$ in range $[1, N]$ **do**
4: **if** $order == 1$ **then**
5: // randomly sample neighbors of the central point
6: $selected_{order} = central.neighbors.random()$
7: **else**
8: // randomly sample neighbors of low-order selected points
9: $selected_{order} = selected_{order-1}.neighbors.random()$
10: // delete the sampled low-order neighbor points
11: $selected_{order}.delete(neighbor_{order-1})$
12: **if** $order > 2$ **then**
13: $selected_{order}.delete(neighbor_{order-2})$
14: **end if**
15: **end if**
16: // store non-redundant neighbor points to the list
17: $npList.append(set(selected_{order}))$
18: **end for**

C. Stencil Representation

As shown in Figure 6, we convert the offset of the accessed neighbor points from the central point into the location of the non-zero elements of a tensor. The representation of a sparse tensor captures the distribution of neighbors accessed and the Euclidean distance among each other. This type of information largely dominates the latency of memory operations, which in turn significantly impacts the performance of stencil computation under certain optimizations. After that, we can feed the assigned tensor of a fixed size to the convolutional neural network (CNN) to predict the best OC of a stencil on the target GPU. The CNN can effectively process the local variation of a tensor with non-linear patterns to achieve high accuracy of classification tasks.

However, this straightforward method is not always the most effective, especially for regression tasks. Regression algorithms are usually combined with feature engineering to achieve better fitting results [28]. As shown in Table II, we extract the candidate feature set according to the computation patterns of stencils. Different from [23], the candidate features extracted by *StencilMART* focus on the distance between neighbor points and the central point instead of the sparsity distribution in the entire tensor space. For example, the feature set includes the number and ratio of non-zero points in neighbors of each order. In principle, both the feature set

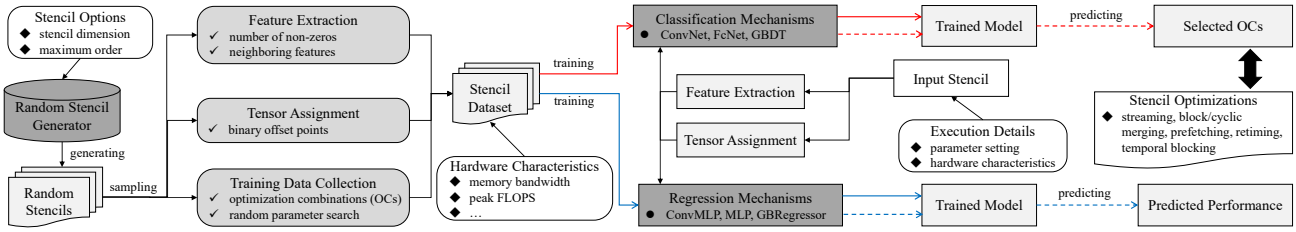


Fig. 5. The design overview of *StencilMART*.

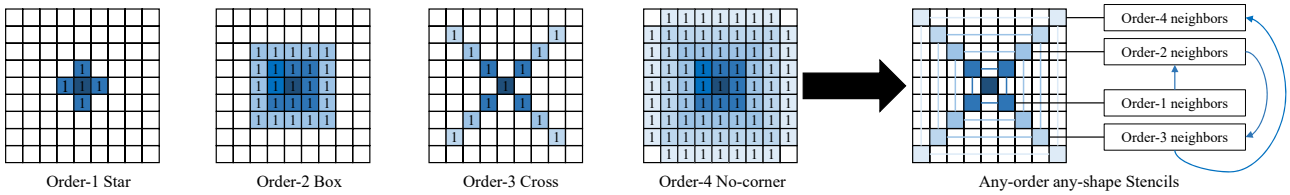


Fig. 6. An example of transforming the access patterns of 2-D stencils into sparse tensors.

and the assigned tensor represent the access pattern of a stencil somehow. For users, which representation is more suitable depends on the performance comparison of different mechanisms in specific scenarios.

TABLE II
THE CANDIDATE FEATURE SET OF A STENCIL.

No.	Feature	Meaning
1	$order$	The maximum extent of non-zeros.
2	nnz	The number of non-zeros in the tensor.
3	$sparsity$	The density of non-zeros in the tensor.
4	$nnz_{order-n}$	The number of non-zeros of order-n neighbors.
5	$nnzRatio_{order-n}$	The ratio of non-zeros of order-n neighbors.

D. Classification Mechanisms for OC Selection

As described in section III-C, we can merge OCs through the Pearson correlation coefficient (PCC) to reduce the classes to be predicted. By doing so, the *StencilMART* avoids jumping among OCs with similar performance, which slows down convergence and interferes with prediction results. In addition, each class must contain sufficient data objects so that its underlying features can be efficiently captured during training. The *StencilMART* implements several classification mechanisms for OC selection of stencil computation. Among machine learning algorithms, we adopt gradient boosted decision tree (*GBDT*), where the input is the stencil feature set (Table II). The *GBDT* is widely used to solve data science problems due to its simplicity and effectiveness [5].

We have also implemented two deep neural networks (DNNs) including *ConvNet* and *FcNet*, where the input is the assigned tensor and feature set for stencils. Take 2-D stencils with the maximum order of 4 as an example, Figure 7 shows the *ConvNet* design using tensor representation. The *ConvNet* digs the hidden features of the tensor input through multiple convolutional layers and fully connected layers, and then outputs the probability of the stencil belonging to each OC. For a particular stencil, the OC with the highest probability is predicted to be the best. The *ConvNet* can handle higher-order stencils simply by varying the input size without modifying the network structure. In addition, we adapt *ConvNet* to 3-D stencils by increasing the dimensionality of convolutional operations. The *FcNet* does not contain any convolutional layers but contains more fully connected layers. The prediction accuracy of *FcNet* is sensitive to the number of layers. For example, *FcNet* with too few layers may fail to learn efficient features, while too many layers lead to overfitting.

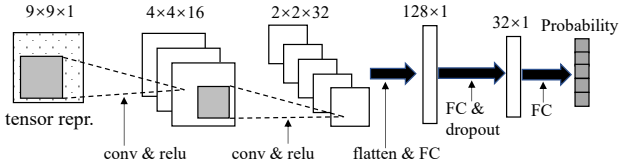


Fig. 7. The design of *ConvNet* using tensor representation.

E. Cross-architecture Performance Prediction

The *StencilMART* uses pre-trained machine learning models to make performance predictions. We treat this prediction task as a regression problem: given a series of input features and a target GPU, predict the execution time of stencil computation on that GPU. The input features include three parts: candidate feature set (or assigned tensor) of a stencil, parameter setting under a specific OC and GPU hardware characteristics. The parameter space of OCs includes parameters of numeric type (e.g., merging factor), Boolean type (e.g., shared memory usage) and enumeration type (e.g., streaming dimension) [25]. For the numerical parameters, we restrict their values to power of two inconsistent with existing works [15], [20]. We parameterize the range of the Boolean type as $\{0, 1\}$. We start from 1 with unit stride to represent the parameters of

enumeration type. Note that when converted to input features, the *StencilMART* performs $log2$ operation on the numerical parameters to ensure the stability of network training.

Inspired by [27], we attach GPU features closely related to memory efficiency and computation performance to the input feature vector. We attach the GPU's *1)* memory capacity and bandwidth; *2)* number of streaming multiprocessors (SMs); *3)* peak FLOPS specified by the manufacturer. After collecting data on the GPUs, we add the execution time of each instance and its corresponding input features on a particular GPU to the training set. The *StencilMART* implements several regression mechanisms including *GBRegressor*, *MLP* and *ConvMLP* for performance prediction. For *MLP* and *ConvMLP*, we normalize the inputs to the range of $[0, 1]$ by dividing by the maxinum value of each input feature. The *GBRegressor* utilizes gradient boosting for regression tasks, which produces a prediction model in the form of an ensemble of decision trees [5].

The multilayer perceptron (*MLP*) comprises an input layer, multiple hidden layers and an output layer that produces the predicted execution time for stencil computation. The number of hidden layers and the number of units per hidden layer can be adjusted to balance prediction performance and inference overhead. Figure 8 shows the *ConvMLP* design using tensor representation. Unlike *MLP*, the *ConvMLP* uses assigned tensors instead of candidate feature sets as the representation of stencils. Specifically, the *ConvMLP* combines CNN and MLP, where the inputs of CNN and MLP are the assigned tensor and the feature vector containing the parameter setting and hardware characteristics, respectively. After that, the outputs of CNN and MLP are merged as joint features and flow into the fully connected layer. Similarly, the *ConvMLP* can be easily adapted to 3-D stencils by increasing the dimensionality of convolutional filters.

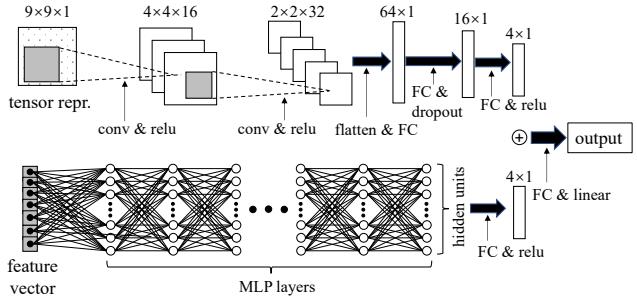


Fig. 8. The design of *ConvMLP* using tensor representation.

V. Evaluation

A. Experiment Setup

1) Hardware and Software Platforms: As shown in Table III, we evaluate the effectiveness of *StencilMART* on different GPU generations. The rental cost of GPUs is taken from Google Cloud[1]. The machines equipped with GPUs are listed in Table IV. The experiments are conducted on Ubuntu 16.04 with CUDA v10.0 and cuDNN v7. The neural networks involved in *StencilMART* (e.g., *ConvNet* and *MLP*) are built using TensorFlow release v1.15 [1], whereas *GBDT* and *GBRegressor* are built using XGBoost release v1.4.2 [5].

[1]Google Cloud pricing in us-central1, as of October 2021.

TABLE III
THE GPUs USED FOR EVALUATION.

GPU	Generation	Mem.	Mem. BW	SMs	TFLOPS	Rental
P100	Pascal	16 GB	720 GB/s	56	5.3	$1.46/hr
V100	Volta	32 GB	900 GB/s	80	7.8	$2.48/hr
2080Ti	Turing	11 GB	616 GB/s	68	0.41	–
A100	Ampere	40 GB	1,555 GB/s	108	9.7	$2.93/hr

TABLE IV
THE MACHINES USED FOR EVALUATION.

CPU	Frequency	Cores	Main Mem.	GPU
Xeon Silver 4110	2.1 GHz	16	192 GB	2080 Ti
Xeon E5-2680 v4	2.4 GHz	28	252 GB	P100, V100, A100

2) Stencil Programs and Datasets: We randomly generate 500 2-D and 500 3-D double-precision stencil programs using *StencilMART*, where the maximum stencil order is set to 4. In consistent with existing works [15], [20], we set the input grids of 2-D and 3-D stencils to 8192^2 and 512^3. We leave the extension of *StencilMART* to consider the grid size as one of its model input for future work. We merge the OCs through PCCs and reduce the number of predicted OCs to 5. For each stencil program, we randomly select parameter settings from OCs and make measurements on four different GPUs. After that, we obtain 64,927 2-D and 76,240 3-D stencil instances on each GPU to form the stencil dataset. Note that the shortest execution time of each OC under different parameter settings is used for OC selection. The stencil dataset is further divided during cross validation.

3) Cross Validation: We use the 5-fold cross validation method [23] to evaluate the accuracy of the models. Specifically, we randomly divide the stencil dataset into five folds. In each round, a single fold is selected as the test set, and the other four folds are used as the training set [24]. For *ConvNet* and *FcNet*, we select the Adam stochastic optimizer with 0.0001 learning rate and a batch size of 50. For *ConvMLP* and *MLP*, we set the learning rate to 0.0005 and the batch size to 256. The convolution filter size is set to 3×3 for both *ConvNet* and *ConvMLP*. We have fine-tuned the number of layers and the layer size of all neural networks. We also carefully tune the hyperparameters of *GBDT* and *GBRegressor*. For OC selection, we compare the predicted best OC with state-of-the-art stencil frameworks including *Artemis* [20] and *AN5D* [15], where the number of randomly selected parameter settings remains the same to ensure a fair comparison. For performance prediction, we use the mean absolute percentage error (MAPE) as the comparison metric.

B. Results for OC Selection

1) Prediction Accuracy: Figure 9 shows the prediction accuracy of *ConvNet*, *FcNet* and *GBDT* on GPUs. As seen, the

ConvNet achieves the highest prediction accuracy in general. The average prediction accuracy of *ConvNet* for 2-D and 3-D stencils is 84.4% and 83.0%, respectively. This proves the practicality of representing the stencil patterns as tensor distribution. Moreover, the local variations of stencil patterns can be effectively captured by convolution operations for classification tasks. The *GBDT* performs slightly worse than *ConvNet*, achieving an average accuracy of 81.7% and 80.8% for 2-D and 3-D stencils. This indicates that the features extracted by *StencilMART* fully reflect the computation patterns with neighbor accesses in stencils. The results show that the *StencilMART* quickly learns the stencil features and maintains convergence on all GPU generations. Due to the poor performance of *FcNet*, we will only present the evaluation results of *ConvNet* and *GBDT* in Section V-B2.

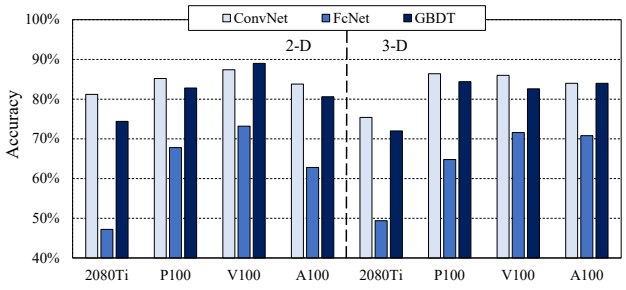


Fig. 9. Prediction accuracy of different classification mechanisms of *StencilMART* on GPUs.

2) Performance Speedup: The performance speedup of stencil computations using the OC predicted by *StencilMART* is presented in Figure 10 and Figure 11. The performance of *Artemis* and *AN5D* are chosen as the baseline. We can observe that both *ConvNet* and *GBDT* achieve higher performance than *Artemis* and *AN5D* on all GPU generations. The *ConvNet* achieves a slightly higher speedup than *GBDT* due to its higher prediction accuracy. Specifically, the *ConvNet* achieves an average speedup of 1.30× and 1.32× over *Artemis* for 2-D and 3-D stencils. The *ConvNet* also achieves 1.33× and 1.09× over *AN5D*, respectively. The results show that no single OC fits all stencil programs. This further demonstrates that selecting the right OC is critical to achieving the high performance of stencil computations. The stable performance speedup of *StencilMART* proves its high scalability for various stencil shapes and GPU architectures.

C. Results for Performance Prediction

1) Percentage Error: Figure 12 shows the test error (MAPE) of *ConvMLP*, *MLP* and *GBRegressor* on GPUs. As seen, all three mechanisms accurately predict the execution time of stencil computations. The prediction results indicate that the *StencilMART* effectively extracts the features of stencil instances and GPU architectures that contribute to execution time. We can observe that *MLP* clearly outperforms *ConvMLP* and *GBRegressor* in most cases. The average test error of *MLP* for 2-D and 3-D stencils is 6.2% and 5.3%, respectively. In

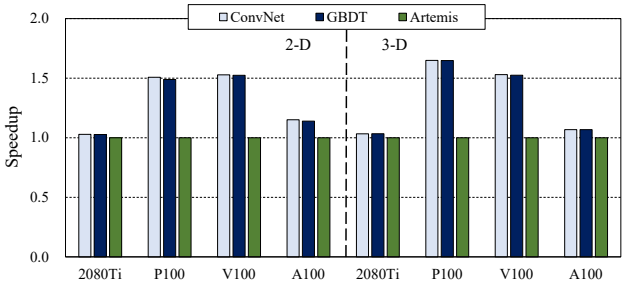


Fig. 10. Speedup of *ConvNet* and *GBDT* over *Artemis* on GPUs.

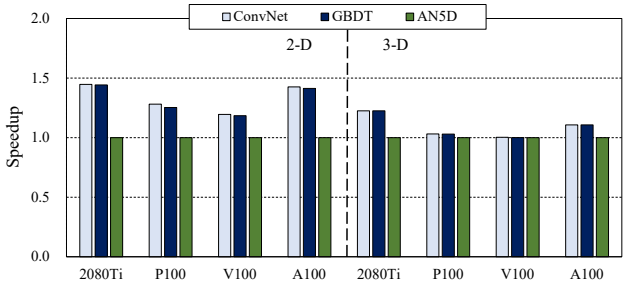


Fig. 11. Speedup of *ConvNet* and *GBDT* over *AN5D* on GPUs.

comparison, *ConvMLP* achieves an average error of 13.4% and 11.6%, whereas *GBRegressor* achieves 9.5% and 6.3%. This indicates the fully connected model architecture of *MLP* can better encode the stencil and GPU features to achieve better prediction results. To conclude, the *StencilMART* is general since it supports different types of stencil instances across GPU generations. We will only present the evaluation results of *MLP* in the following due to its superior performance.

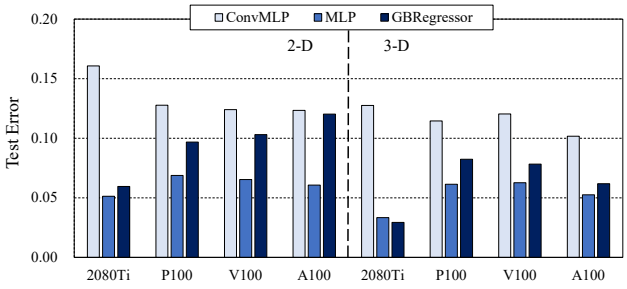
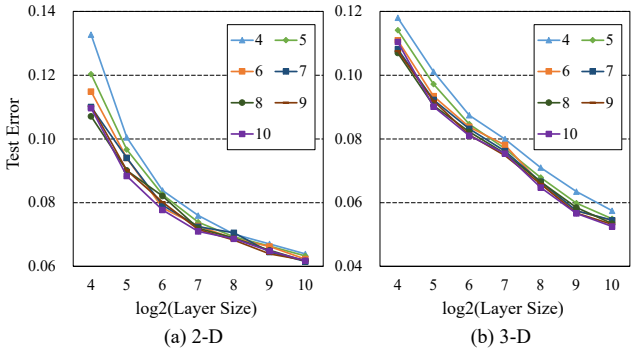


Fig. 12. Test error of different regression mechanisms of *StencilMART* on GPUs.

2) MLP Designs: To better understand how the network design affects *MLP*'s prediction accuracy, we conduct a sensitivity study where we vary the number of hidden layers (4 to 10) along with their size. Figure 13 shows the average test error for *MLP* across GPUs after being trained for 100 epochs. The x-axis represents the layer size ranging from 2^4 to 2^{10} with a stride of $\times 2$. As seen, the *MLP* for 2-D and 3-D stencils appear to follow a similar test error trend. Specifically, increasing the number of layers and their sizes leads to lower test errors. In addition, increasing the number of layers beyond

seven leads to diminishing returns on stencil computations. Therefore, we can conclude that using seven layers for *MLP* is a reasonable choice. The *StencilMART* provides an easy-to-use interface for modifying network parameters to evaluate the impact of network designs on prediction accuracy.



(a) 2-D (b) 3-D

Fig. 13. Test error of *MLP* as we vary the number of hidden layers and layer size. The x-axis is in a logarithmic scale.

D. Case Study: To Rent or Not To Rent a Cloud GPU

One scenario the researchers may face is deciding whether to rent GPUs in the cloud for stencil computations or to stick with a local GPU. With *StencilMART*, they can use their local GPU to make predictions about the performance of each cloud GPU so as to make this decision in an informed way. Next, we analyze the prediction accuracy of *StencilMART* according to the pure performance and cost efficiency.

1) Pure Performance: Figure 15 shows the ground truth on stencil instances considering pure performance. The prediction accuracy for each GPU is also presented. As seen, more "powerful" GPUs do not always achieve better performance for stencil computations. Specifically, the 2080Ti, P100, V100, and A100 account for 20.2%, 17.8%, 40.2% and 21.8% of 2-D stencil instances with the best performance, whereas 20.1%, 16.6%, 26.4% and 36.9% of 3-D stencil instances. This indicates the necessity of cross-architecture performance prediction for the decision to rent cloud GPUs. Another observation can be drawn that despite any prediction errors, the *StencilMART* still correctly predicts the best GPU for most stencil instances. The average accuracy of *StencilMART* for 2-D and 3-D stencil instances is 96.7% and 97.3%, respectively. The prediction results therefore allow users to make correct decisions based on pure performance.

2) Cost Efficiency: Since Google Cloud does not have the 2080Ti for rental, we only compare the cost efficiency of the other three GPUs. Figure 15 shows the ground truth on stencil instances considering cost efficiency. The prediction accuracy for each GPU is also presented. We can discover that the P100 is the most cost-efficient to rent for most stencil instances. Specifically, the P100, V100 and A100 account for 61.0%, 22.7% and 16.3% for 2-D stencil instances with the best cost efficiency, whereas 56.7%, 20.6% and 22.7% for 3-D stencil instances. Therefore, if the users are not critically

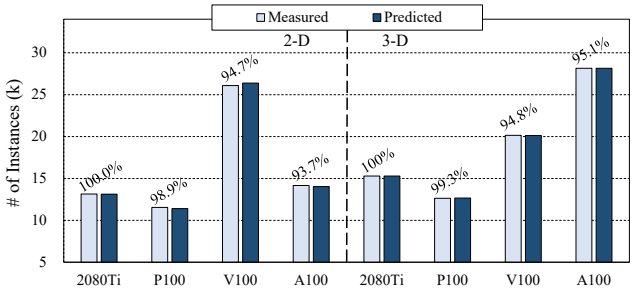


Fig. 14. The ground truth on stencil instances considering pure performance. The prediction accuracy is shown above each bar.

constrained by time and want to optimize for the cost, they would more likely rent the P100 instead of V100 or A100 with higher performance. Nevertheless, the *StencilMART* achieves an average accuracy of 97.3% and 96.1% for 2-D and 3-D stencil instances. The results indicate that the *StencilMART* can correctly predict the most cost-efficient GPU, which in turn allows users to make the right decision for rental.

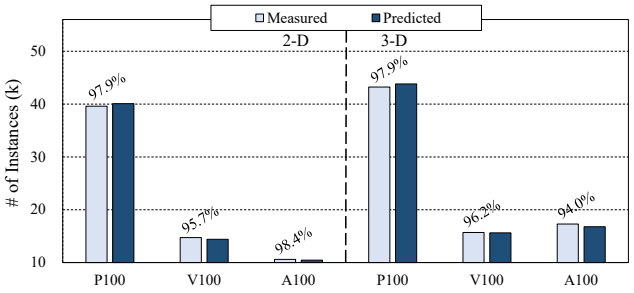


Fig. 15. The ground truth on stencil instances considering cost efficiency. The prediction accuracy is shown above each bar.

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 [9], [10], [12], [14], [15], [17]–[20]. *Physis* [14] translated user-written stencil code into scalable implementation for GPU-equipped cluster. *Forma* [19] proposed a DSL for image processing application with stencil operations. Grosser *et al.* [9] presented a novel hybrid tiling method that combined hexagonal tiling and wavefront tiling on GPUs. Hagedorn *et al.* [10] explored how to use *LIFT* primitives to implement stencil codes and optimizations such as tiling. Matsumura *et al.* [15] 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* [17] automatically pipelined and dynamically scheduled stencil execution on GPUs. However, none of the above works support the optimization selection for stencil computation. The *StencilMART*

can be integrated into these DSLs and quickly determine the best optimization combination for target stencils.

Performance Prediction on GPUs. Since it's non-trivial to characterize the execution behavior of GPU kernels for efficient task scheduling, a large amount of research works focus on performance prediction on GPUs [2], [3], [11], [27], [29]. Ardalani *et al.* [3] built an ensemble model using forward selection to estimate the GPU execution time with only single-threaded CPU implementation. Konstantinidis *et al.* [11] proposed a quantitative performance model based on the roofline approach, which utilized microbenchmarking to investigate GPU performance on operation intensity values. *PPT-GPU* [2] added models for different memory hierarchies to Performance Prediction Toolkit (PPT) and predicted the GPU performance based on PTX ISA and GPU configurations. *Daydream* [29] constructed a kernel-level dependency graph and introduced a set of graph-transformation rules to simulate the overall runtime for DNNs. Yu *et al.* [27] scaled the execution time of each operation in a training iteration from one GPU to another using either wave scaling or multilayer perceptrons. To the best of our knowledge, this is the first work that targets the cross-architecture performance prediction for stencil computation.

VII. Conclusion and Future Work

In this paper, we propose an automatic optimization selection framework *StencilMART*, which effectively predicts the best optimization combination for a stencil running on a particular GPU. Furthermore, the *StencilMART* supports cross-architecture performance prediction for stencils. Specifically, the *StencilMART* represents the stencil patterns as binary tensors and neighboring features through tensor assignment and feature extraction. After that, the *StencilMART* implements the prediction model using various machine learning mechanisms such as classification and regression. The experiment results show that the *StencilMART* can achieve high prediction accuracy for optimization selection. In addition, the *StencilMART* can accurately predict the execution time of stencil computations across GPUs, thus allowing users to make informed decisions for GPU selection.

For future work, we would like to extend *StencilMART* to support stencil kernels with boundary conditions. To achieve that, we need to quantify the impact of boundary conditions on performance and further parameterize them as model input. Then, other components of *StencilMART* can be reused for optimization selection or performance prediction.

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] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al.: Tensorflow: A system for large-scale machine learning. In: 12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16). pp. 265–283 (2016)

[2] Arafa, Y., Badawy, A.H.A., Chennupati, G., Santhi, N., Eidenbenz, S.: Ppt-gpu: Scalable gpu performance modeling. IEEE Computer Architecture Letters **18**(1), 55–58 (2019)

[3] Ardalani, N., Lestourgeon, C., Sankaralingam, K., Zhu, X.: Cross-architecture performance prediction (xapp) using cpu code to predict gpu performance. In: Proceedings of the 48th International Symposium on Microarchitecture. pp. 725–737 (2015)

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

[5] Chen, T., Guestrin, C.: Xgboost: A scalable tree boosting system. In: Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining. pp. 785–794 (2016)

[6] Cosenza, B., Durillo, J.J., Ermon, S., Juurlink, B.: Autotuning stencil computations with structural ordinal regression learning. In: 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS). pp. 287–296. IEEE (2017)

[7] 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)

[8] 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)

[9] 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)

[10] 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)

[11] Konstantinidis, E., Cotronis, Y.: A quantitative roofline model for gpu kernel performance estimation using micro-benchmarks and hardware metric profiling. Journal of Parallel and Distributed Computing **107**, 37–56 (2017)

[12] Li, M., Liu, Y., Hu, Y., Sun, Q., Chen, B., You, X., Liu, X., Luan, Z., Qian, D.: Automatic code generation and optimization of large-scale stencil computation on many-core processors. In: Proceedings of the 50th International Conference on Parallel Processing. pp. 1–12 (2021)

[13] Martínez, V., Dupros, F., Castro, M., Navaux, P.: Performance improvement of stencil computations for multi-core architectures based on machine learning. Procedia Computer Science **108**, 305–314 (2017)

[14] 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)

[15] 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)

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

[17] 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)

[18] 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)

[19] 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)

[20] 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)
[21] 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)
[22] 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)
[23] Sun, Q., Liu, Y., Dun, M., Yang, H., Luan, Z., Gan, L., Yang, G., Qian, D.: Sptfs: sparse tensor format selection for mttkrp via deep learning. In: SC20: International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 1–14. IEEE (2020)
[24] Sun, Q., Liu, Y., Yang, H., Dun, M., Luan, Z., Gan, L., Yang, G., Qian, D.: Input-aware sparse tensor storage format selection for optimizing mttkrp. IEEE Transactions on Computers (2021)
[25] Sun, Q., Liu, Y., Yang, H., Jiang, Z., Liu, X., Dun, M., Luan, Z., Qian, D.: cstuner: Scalable auto-tuning framework for complex stencil computation on gpus. In: 2021 IEEE International Conference on Cluster Computing (CLUSTER). pp. 1–12. IEEE (2021)
[26] Sun, Q., Liu, Y., Yang, H., Luan, Z., Qian, D.: Smqos: Improving utilization and energy efficiency with qos awareness on gpus. In: 2019 IEEE International Conference on Cluster Computing (CLUSTER). pp. 1–5. IEEE (2019)
[27] Yu, G.X., Gao, Y., Golikov, P., Pekhimenko, G.: A runtime-based computational performance predictor for deep neural network training. arXiv preprint arXiv:2102.00527 (2021)
[28] Zheng, A., Casari, A.: Feature engineering for machine learning: principles and techniques for data scientists. " O'Reilly Media, Inc." (2018)
[29] Zhu, H., Phanishayee, A., Pekhimenko, G.: Daydream: Accurately estimating the efficacy of optimizations for {DNN} training. In: 2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20). pp. 337–352 (2020)