2021 IEEE/ACM Programming Environments for Heterogeneous Computing (PEHC) | 978-1-7281-8676-4/21/$31.00 ©2021 IEEE | DOI: 10.1109/PEHC54839.2021.00006

GenMAT: A General-Purpose Machine Learning-Driven Auto-Tuner for Heterogeneous Platforms

Naifeng Zhang* Ajitesh Srivastava* Rajgopal Kannan† and Viktor K. Prasanna*
*University of Southern California
{naifengz, ajiteshs, prasanna}@usc.edu
†US Army Research Lab
rajgopal.kannan.civ@mail.mil

***Abstract*—As computing platforms evolve with heterogeneous resources, developing optimized code that fully exploits the computing power becomes increasingly challenging. Domain experts need extensive knowledge of computer architecture, compiler optimizations, and parallel computing to understand which implementation will work best for their problem domain and data. Even with considerable time learning, writing, and debugging high-performance code, such optimizations may not generalize to different inputs, applications, or computing platforms. To assist the end-users in optimally deploying workloads on the heterogeneous environment with high productivity, a fundamental problem is to automatically find the best "variant" of an application—the implementation with the optimal configurations on the most suitable hardware resource resulting in the minimum runtime. We propose GenMAT, a portable tool for identifying the best variant of any application specified as a meta-program with exposed tunable parameters on any hardware. GenMAT automatically profiles the application by varying the exposed tunable parameters to generate a small set of profiling data. Then, GenMAT trains a compact machine learning model that is used to quickly predict the runtimes of a large number of possible parameter settings to identify the best variant. We show that the variant selected by GenMAT has a runtime deviation within 3.5% of the true best variant in determining the best linear algebra library for matrix operations. For identifying the best Halide schedule, GenMAT correctly ranks the runtimes of thousands of candidates with an average Spearman's rank correlation coefficient of 0.95.**

***Index Terms*—automatic performance tuning, heterogeneous platforms, performance modeling, machine learning**

I. Introduction

The heterogeneity and scale of computing platforms have grown significantly, yielding an unprecedented amount of computing power. Nevertheless, to write high-performance code that fully harnesses the computing power generated by heterogeneous platforms, domain experts need extensive high performance computing (HPC) knowledge and spend a considerable amount of time learning, writing, debugging, and tuning the high-performance code. Still, challenges remain in generalizing such optimizations to different inputs, applications, and computing platforms, rendering trying numerous configurations inevitable. Not only time-consuming, such trial-and-error-based optimization is bound to limit optimality of the code due to limited exploration of all implementation and optimization choices—which library should be used, how many threads should be assigned, how to unroll and tile loops, whether to use a CPU or GPU implementation, etc.

Each of the choices for implementing an application on a hardware resource may lead to a different object code resulting in a different runtime. We refer to a particular choice of code and hardware to implement and run an application as a *variant*. We define the variant selection problem as finding the variant that results in the minimum runtime. With ever-growing libraries in programming languages, fast-emerging domain-specific languages (DSLs), and evolving architecture, an ideal solution to the variant selection problem should be *portable*, i.e., (i) it should be able to decide among many arbitrary implementations of the same application on different hardware and (ii) it should not be specific to a language or architecture, so that it can adapt to new technologies, applications, and algorithms in the future. An additional benefit of such a solution is the increased *productivity*—the end-user does not need to spend time on learning new optimizations nor on manually tuning the code. Instead, the ideal implementations will be automatically identified. To simplify the representation of various implementations, we assume that the application is presented in the form of a meta-program/script that takes a set of parameters as arguments. In this context, a meta-program $M(P, \mathbf{X})$ runs the given program P with parameters $\mathbf{X} = \{x_1, x_2, \ldots, x_n\}$. Each parameter x_i can be considered a tuning knob, whose value affects the application's runtime, and each set of parameter values along with a choice of hardware generates a variant. We wish to identify a set of parameter values that results in the lowest runtime among all hardware resources.

We propose GenMAT, a general-purpose auto-tuner that identifies the variant that produces the lowest runtime among all the available hardware/components on heterogeneous platforms. GenMAT automatically (i) profiles the target application, (ii) trains a performance prediction model, and (iii) uses the model to generate the predicted best execution plan that suggests which hardware on the target platform should be used and the optimal implementation of the target application on

978-1-7281-8676-4/21/$31.00 ©2021 IEEE
DOI 10.1109/PEHC54839.2021.00006

that hardware.

GenMAT is not limited to tuning a specific application or for a specific architecture. While better optimizations may be obtained by limiting the scope and incorporating domain knowledge, we trade such optimizations for generalizability and productivity—the user only needs to specify the tunable parameters and add few lines of code to change the target program into a meta-program, as GenMAT simply relies on the ability to run the meta-program with various sets of parameters to build a performance model. Furthermore, the user can store a trained performance model to the built-in knowledge base and reuse it to select variants of the same application without profiling again, e.g., running the same application for different input data.

GenMAT's default performance prediction model is a compact neural network. The modular nature of GenMAT allows for easily replacing the neural network with a different machine learning (ML) model, which enables the community to improve GenMAT as better performance prediction algorithms are discovered. Benefiting from the black-box nature of the ML approach, GenMAT can be used for applications running on individual hardware, large systems, and even "partitions" (e.g., a CPU and GPU combination) that are specialized for a workload.

Our key contributions are:

- We propose GenMAT, a portable ML-based auto-tuner that is agnostic to any programming language or hardware. The user only needs to specify the tunable parameters and add as low as three lines of code to the target program to start GenMAT, which automatically identifies the best-performing variant—the best implementation on the most suitable hardware resource—on the target platform.
- We implement GenMAT in a modular fashion, which allows developers to easily modify it, for example, to improve the performance prediction model or accelerate profiling.
- We demonstrate that the variant selected by GenMAT has a runtime deviation within 3.5% of the true optimal variant for deciding the linear algebra library for a given matrix operation. GenMAT correctly ranks the runtimes of thousands of Halide schedules (variants) with an average Spearman's rank correlation coefficient of 0.95.
- We demonstrate that GenMAT can process more than 5000 candidates to identify the best variant in $1ms$, thereby allowing GenMAT to make decisions at both compile-time and runtime.

II. Related Work

Prior work on performance optimization/tuning is mostly empirical [1], [2], designing algorithms [3], [4] or developing ML-based systems [5]–[10] to find an optimal implementation of programs that produces the lowest runtime on specific architecture or using specific programming languages. Fastest Fourier Transform in the West (FFTW) [1] and Automatically Tuned Linear Algebra Software (ATLAS) [2] use empirical search for optimization on fast Fourier transform (FFT) and basic matrix operations, respectively. Steiner et al. [3] introduce a value function-based algorithm for performance optimization and another algorithm to train the value function estimate. However, they do not specify the training time of the value function estimate, and therefore there is no evidence showing that it can generalize to a new program in a short amount of time as GenMAT does. Decima [7], a general-purpose scheduling framework for computationally intensive tasks with pipelines, utilizes reinforcement learning (RL) techniques to learn from the actual computation environment, and the RL agent can adapt to different workloads. Decima uses performance prediction neural networks that consist of 12,736 parameters, which is approximately $170\times$ larger than GenMAT's default predictor. Hayashi et al. [9] use supervised ML techniques to construct performance heuristics that select a preferable hardware device from CPUs and GPUs. However, this approach is limited to Java 8 parallel stream APIs.

DSLs such as Halide [11] make it easier to express certain optimizations through a "schedule" that determines the execution plan with a set of parameters. However, finding these parameters is still an open problem as there is no fixed set of parameters, let alone a fixed set of parameter values, that optimizes all programs. Halide autoschedulers [5], [6] work specifically for Halide kernels by automatically generating a schedule shape and choosing the scheduling parameters. The ML predictor used by GenMAT outperforms the autoscheduler of Mullapudi et al. [5] with a 1.7x speedup on the Blur kernel [12]. Darkroom [4] is a compiler and DSL for high-performance image processing. The core algorithm of Darkroom solves an integer linear program to generate ASIC design, FPGA code, or CPU code, given the user's high-level code. OpenTuner [10] is a framework for building domain-specific program auto-tuners, using ensembles of disparate search techniques to find an optimal solution. SPIRAL [13] is an autotuning system that uses a unifying mathematical formalism named operator language to capture computational kernels and then solves an optimization problem to find the best candidate program for the given platform and computational kernel.

Our Focus: Unlike existing work, we create a lightweight, *portable* auto-tuner that is not restricted to any programming language or architecture. *As long as the user can run the to-be-tuned program from the command line*, GenMAT is applicable. Instead of learning a new auto-tuning system or language, the user only needs to specify the tunable parameters and add as low as *three* lines of code to the target program to start GenMAT. Moreover, extremely lightweight design and profiling accelerating techniques allow GenMAT to be used at both compile-time and runtime.

While we present our design for general programs, we envision that GenMAT can be leveraged to tune large workloads, frameworks, and systems. A workload can be assumed to consist of a collection of tasks, where each task is an operation that has a significant contribution to the runtime. Since a small

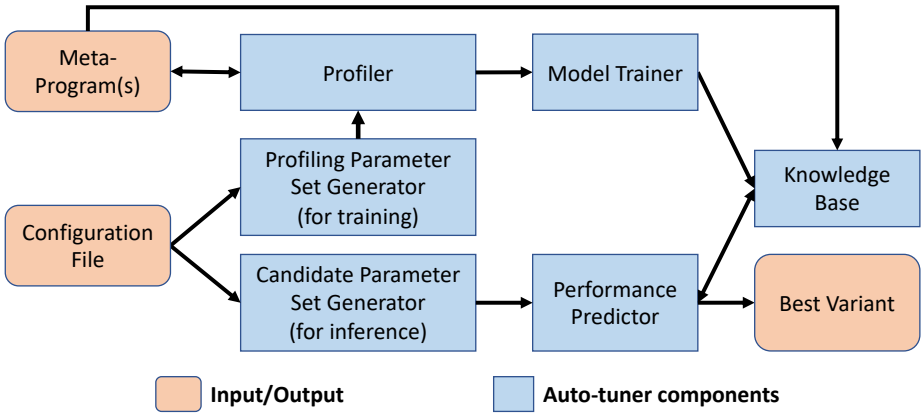

Fig. 1: Overview of GenMAT modules and their interactions.

number of pre-defined tasks (such as matrix operations) can cover a large number of workloads [14], GenMAT should be able to utilize known libraries implementing those tasks for variant selection instead of relying on the user to provide various implementations to choose from. Given our motivation, we demonstrate the effectiveness of GenMAT on programs that are dominant tasks in many workloads, such as matrix multiplication, convolution, pooling, and FFT.

III. Design

As shown in Fig. 1, GenMAT has a modular design where each module can be separately customized in the future for improvement. The inputs to GenMAT are one or more meta-programs (see Section III-A) and a configuration text file (see Section III-B). One meta-program corresponds to one decision (e.g., which hardware to use) described in Section III-B1.

GenMAT repeats the following procedures for each meta-program specified in the configuration file.

1) GenMAT checks if the meta-program has been profiled before by consulting the knowledge base.
   - If no, the Profiling Parameter Set Generator module will be used to generate a collection of parameter sets (see Section III-B2a) with specific parameter values for training the performance prediction model. Then, the Profiler module runs the meta-program with each set of parameter values to generate training data. This data is passed to the Model Trainer module, which trains a performance model, loads it into the Performance Predictor module, and stores it into the knowledge base.
   - If yes, the previously-trained performance prediction model will be retrieved from the knowledge base and loaded into the Performance Predictor module.
2) The Candidate Parameter Set Generator module generates a collection of partial parameter sets with specific values (see Section III-E).
3) GenMAT provides an interface for the user to query for the best variant for any input size (see Section III-G).
4) Each candidate (i.e., the queried input size combined with each pre-generated partial set of values) is evaluated by the Performance Predictor module to obtain a predicted runtime.
5) GenMAT ranks all the candidates using the predicted runtime and selects the set of parameter values that is predicted to result in the lowest runtime for the meta-program.

After repeating the process for all meta-programs, for the queried input size, GenMAT compares the predicted minimum runtime of each meta-program and selects the meta-program that produces the lowest runtime—the best variant. The final output is a decision followed by a set of values for the tunable knobs, which represents the best implementation on the most suitable hardware resource.

A. Program to Meta-Program

To change a program into a meta-program, the user needs to modify the original code to

- specify the tunable parameters
- read the command line arguments and assign them to tunable parameters accordingly
- (*Optional*) enforce constraints on parameters

For some meta-programs, there are certain constraints on the parameters for the meta-program to be successfully executed. In Halide, for example, the inner vectorization width cannot exceed the outer tile size. GenMAT is designed in a way that the user has the choice of enforcing the constraints or not. If the user chooses not to enforce the constraints, whenever GenMAT runs into an error during profiling, it will skip to the next set of parameter values. Therefore, the user can decide whether to spend time learning interactions among parameters before starting GenMAT or to tolerate the cost of trial-and-errors (which depends on when the error will be thrown) during profiling.

The Appendix shows the code of the original Blur kernel written in Halide and the code of the same Blur kernel in the form of a meta-program. The user only needs to change

input metadata and numeric parameters (described in Section III-B2a) in the original program to `tuning[i]` and add three lines of code (line 4 to line 6 in Listing 7) if not enforcing the constraints. The user can extend those three lines of code to enforce the constraints among parameters.

```
1 decision=cpu gpu1 gpu2
2 input-metadata=2048 2048 2048
3 numeric=1024 32
4 functional=1*2*3
5 path=mm.cpp
6 compile-cmd=g++ -g -fopenmp mm.cpp -o mm_cpu
7 run-cmd=./mm_cpu
8 reuse-model=n
9 input-metadata=8192 8192 8192
10 numeric=1024 1024 1024 1024
11 functional=1*2*3
12 path=mm.cu
13 compile-cmd=nvcc mm.cu -o mm_gpu
14 run-cmd=CUDA_VISIBLE_DEVICES=0 ./mm_gpu
15 reuse-model=n
16 input-metadata=8192 8192 8192
17 numeric=1024 1024 1024 1024
18 functional=1*2*3
19 path=mm.cu
20 compile-cmd=nvcc mm.cu -o mm_gpu
21 run-cmd=CUDA_VISIBLE_DEVICES=1 ./mm_gpu
22 reuse-model=n
```

Listing 1: Example of the configuration file for matrix multiplication.

B. Configuration File

The configuration file for GenMAT consists of one line of decisions (see Section III-B1) and one or more execution blocks (see Section III-B2), where each block corresponds to one decision in order. Listing 1 is an example of a configuration file where the target application is matrix multiplication.

1) Decisions: Each decision is a nominal choice that corresponds to the execution block in order. That is, GenMAT only needs to know that the ith decision-string corresponds to the ith execution block instead of the meaning of each string. Examples of decisions are choices among different hardware resources or whether to use a compilation flag or not.

```
1 decision=cpu gpu1 gpu2
```

Listing 2: Example of decisions in the configuration file.

2) Execution Blocks: The execution block specifies how to run the meta-program with tunable parameters. In Listing 1, the first execution block (line 2 to line 8) corresponds to cpu, the second (line 9 to line 15) corresponds to gpu1, and the third (line 16 to line 22) corresponds to gpu2. Each execution block consists of two parts: parameter sets and execution commands. In the following, we use the first execution block as an example for illustration.

a) Parameter Sets: The user needs to specify the following parameters:

- **input-metadata** parameters determine metadata on inputs that the program takes. We assume that the given program has an input generator function that generates an input based on the provided metadata. Using matrix multiplication as an example, we assume that there is a function that takes in matrix dimensions (input metadata) and outputs a matrix (input data) that will be used for matrix multiplication. We refer to a set of input metadata values as an input size.
- **numeric** parameters are the tuning knobs whose value affects the application's runtime. For example, the tile size to be used for a matrix operation. For one input size, GenMAT will predict the set of numeric values that results in the lowest runtime.
- **functional** parameter (*Optional*) denotes an additional input that can assist with performance modeling, such as the estimated complexity of the program (see Section III-D). Unlike other parameters, functional parameter is not passed as an argument to the meta-program for profiling but passed to the Performance Predictor module.

```
2 input-metadata=2048 2048 2048
3 numeric=1024 32
4 functional=1*2*3
```

Listing 3: Example of parameter sets for matrix multiplication on CPU.

Each value specified for input metadata or numeric parameter is the inclusive upper bound for that parameter. The default inclusive lower bound for each parameter is 2. For example, the ranges of the numeric parameters to be tuned in Listing 3 are $[2, 1024]$ and $[2, 32]$, respectively. GenMAT calculates a suitable base to perform logarithmic sampling for the numeric parameters so that the total number of sets generated approximately matches the desired number specified by the user. By default, the number of profiling parameter sets is 250, and the number of candidate parameter sets is 5000.

For instance, under the specification of Listing 3, the following line can be executed as a variant:

```
./mm_cpu 1024 1024 128 16 4
```

which means executing the meta-program *mm_cpu* for two matrices of size 1024×1024 and 1024×128 with tile size 16 using 4 threads. The functional parameter in this example suggests that the product of the first, second, and third argument should be treated as an additional input for performance modeling.

b) Execution Commands: The execution commands consist of:

- **path**: relative path of the meta-program with respect to the configuration file.
- **compile-cmd**: command to compile the original program.
- **run-cmd**: command to run the original program.
- **reuse-model**: if GenMAT detects that there exists a pre-trained model for the meta-program,
  - **y** means to use the pre-trained model and therefore skip profiling and training.

– **n** means to profile and train a new model.

Since the user only adds file I/O code to the original program, the commands to compile and run the original program will be able to compile and run the meta-program.

```
5 path=mm.cpp
6 compile-cmd=g++ -g -fopenmp mm.cpp -o mm_cpu
7 run-cmd=./mm_cpu
8 reuse-model=n
```

Listing 4: Example of execution commands for matrix multiplication on CPU.

GenMAT is started using a bash command:

```
bash genmat.sh
```

C. Profiling for Training

Given the ranges of tunable parameters (i.e., input metadata and numeric parameters), GenMAT profiles the meta-program by running it with various randomly sampled sets of values. For certain types of programs, GenMAT can automatically identify tunable parameters and constraints on them to generate random parameter values. For example, in Halide programs, the "scheduling parameters" are considered as numeric parameters, and GenMAT can identify them along with the constraints by scanning for Halide scheduling keywords such as *.tile* and *.vectorize*.

GenMAT also accelerates profiling by

- running the meta-program on small inputs so that the runtime to obtain one training data instance during profiling is low. We have shown that our default performance model can be trained on small inputs and yet generalize well to large inputs (see Section IV-B3b).
- using a compact neural network for performance modeling, due to which the training time is in the order of seconds.

D. Performance Modeling

For each application on arbitrary hardware with arbitrary implementation, GenMAT trains a performance model using the profiled data. The model will be used for performance prediction to select the best variant. By default, GenMAT utilizes the lightweight performance prediction model proposed by Srivastava et al., Augmented Neural Network (NN+C) [12]. The key innovation of NN+C is "augmenting" a mathematical function, which captures the complexity of the application on the target hardware, as an input feature to increase prediction accuracy. In GenMAT, the augmentation is enabled by the functional parameter. Note that specifying the functional parameter is optional, and if it is not provided, GenMAT will run NN+C without the augmentation.

The lightweight design of NN+C—two hidden layers which result in less than 75 parameters—allows extremely fast training and inference that enables GenMAT to make decisions during compile-time as well as runtime. We make NN+C default in GenMAT as it has been shown to outperform traditional feed-forward neural networks for performance prediction on 48 implementation-hardware combinations [12]. GenMAT allows for replacing NN+C with any other neural network as a TensorFlow model [15].

E. Variant Selection

GenMAT generates thousands of candidate parameter values by randomly varying numeric parameters for any given input size. Note that the Profiling Parameter Set Generator generates data points by varying all tunable parameters (i.e., input metadata and numeric parameters), whereas the Candidate Parameter Set Generator generates partial candidates first by varying numeric parameters. When the user queries for the best variant for an input size (see Section III-G), complete candidates will be formed by combining candidate numeric values with the given input size. Then, GenMAT uses the Performance Predictor module to evaluate each of the candidates, ranks them according to their predicted runtime, and selects the variant that is predicted to produce the least runtime. Therefore, GenMAT can identify the fastest variant without actually running the meta-program with each of the candidates.

F. Model Reusability

GenMAT comes with a built-in lightweight knowledge base. When GenMAT encounters a meta-program, profiles it, and trains a performance model, it saves the model in the knowledge base with an ID associated with the hash of the meta-program file. When GenMAT is used again for the same meta-program, it can reuse the previously trained model directly, without profiling and training. For instance, if the user wants to obtain the best variant for a different input size, the previously trained model is reusable. If GenMAT is integrated with a compiler, storing the model in the knowledge base also allows the compiler to consult it if needed at other stages of compilation and execution. GenMAT can output multiple top candidate variants, and the compiler can decide which one to use during execution for global optimization. GenMAT allows for integrating an external knowledge base to replace the built-in one. We have shown that the knowledge base proposed by Zhou et al. [14] can be integrated into GenMAT.

G. User Interface

In the end, GenMAT provides a user interface for querying for the best variant for *any* input size within or out of the range specified in the configuration file. In Section IV-B3b, we have shown that our performance models can generalize well to unforeseen input sizes. Thus, the user can query for out-of-range input sizes but may obtain sub-optimal results. As shown in Listing 5, for each query (line 1 and line 4), GenMAT outputs the decision and a set of numeric parameter values. By default, if there are ten or fewer possible input sizes within the range, GenMAT will automatically query for all possible input sizes and output the results. Then, GenMAT will start the interface for the user to query for other input sizes.

```
>1024 1024 128
decision: cpu
numeric: 256 16
>4096 4096 4096
decision: gpu1
numeric: 8 256 256 128
```

Listing 5: Example of the user interface.

GenMAT's source code, including a thorough code walk-through, is available on our public repository[1].

IV. Evaluation

A. Experimental Setup

We conducted our experiments on five platforms: Intel(R) Xeon(R) CPU E5-2650 v2 @ 2.60GHz (Xeon), with 32 cores, 64 threads; Intel(R) Core i7-8750H CPU @ 2.20GHz (I7), with 12 cores, 24 threads; Intel(R) Core i5-7360U CPU @ 2.30GHz (I5), with 2 cores, 4 threads; NVIDIA Tesla K40c (Tesla) and NVIDIA Quadro K420 (Quadro). To perform matrix operations on CPU, we use Eigen/Dense and Eigen/Sparse from the Eigen library [16], and uBLAS/matrix and uBLAS/matrix_sparse from the Boost library [17] in C++. On GPU, we use global memory and shared memory in CUDA to optimize matrix operations. To perform Gaussian Blur (Blur) and fast Fourier transform (FFT), we use Halide.

Our experiments are designed to evaluate the following of GenMAT: (i) capability to decide the best library to implement an application, (ii) capability to identify the best tuning for numeric parameters, and (iii) overall processing time.

B. Results

1) Choosing the Best Library: We evaluate GenMAT on choosing the best library (variant) for matrix-matrix multiplication, max-pooling, and matrix convolution on each of the aforementioned platforms. Except for matrix convolution on Tesla and Quadro, other program-hardware combinations have a single best variant choice for all input sizes. For example, using the Boost library leads to a better performance than using the Eigen library for max-pooling for all input sizes on all CPUs we conducted experiments on. Therefore, we only discuss the results of matrix convolution on Tesla and Quadro, where choosing between shared and global memory implementation of CUDA is not clear.

To evaluate GenMAT's performance in choosing the best library, we measure the average (over all input sizes) percentage runtime deviation, i.e., how close is the runtime of the selected variant compared to the true best variant:

$$\text{avg}\left(\frac{T_{\text{selected}} - T_{\text{best}}}{T_{\text{best}}}\right) \times 100\%$$

where for each input size, T_{selected} is the runtime of the selected variant and T_{best} is the runtime of the true best variant. To obtain T_{best} for each input size, we exhaustively run and measure the runtime of all the variants generated by the Candidate Parameter Set Generator.

As shown in Table I, the GenMAT-selected variant has an average percentage runtime deviation of 3.9% and 3.5% from the true optimal choices on Tesla and Quadro, respectively. This indicates that the selected variant (if not the best variant) has a runtime significantly closer to the optimal one compared to the baselines that always pick global memory or always pick shared memory.

TABLE I: Comparison of average percentage runtime deviations for matrix convolution on Tesla and Quadro.

	Tesla	Quadro
GenMAT	3.9%	3.5%
$\text{CUDA}_{\text{Global Memory}}$	9.4%	9.5%
$\text{CUDA}_{\text{Shared Memory}}$	7.8%	7.4%

2) Determining Halide Schedules: We evaluate GenMAT on determining Halide schedules for Halide Blur and FFT kernels. A Halide schedule is a set of values for vectorization width, tile size, etc., which directly affects the runtime of the Halide kernel. While in Section IV-B1, GenMAT chooses between two or three libraries, GenMAT needs to select the best schedule from thousands of candidate Halide schedules.

We use Spearman's rank correlation coefficient (ρ) to evaluate GenMAT's capability to rank candidates. Let the predicted runtimes be $\{\hat{t}_1, \hat{t}_1, \ldots, \hat{t}_N\}$ and the ground truth be $\{t_1, t_2, \ldots, t_N\}$. Let $d_i = |\operatorname{rank}(t_i) - \operatorname{rank}(\hat{t}_i)|$, where $\operatorname{rank}(t_i)$ is the rank of t_i among $\{t_1, t_2, \ldots, t_N\}$ and $\operatorname{rank}(\hat{t}_i)$ is the rank of $\hat{t}_i$ among $\{\hat{t}_1, \hat{t}_1, \ldots, \hat{t}_N\}$, ranking from the lowest value to the highest value. Spearman's rank correlation coefficient is defined as

$$\rho = 1 - \frac{6\sum_{i=1}^{N} d_i^2}{N\left(N^2 - 1\right)}$$

ρ ranges from -1 to 1. ρ of 1 indicates a perfect positive correlation of prediction's ranks and truth's ranks, whereas ρ of -1 indicates a perfect negative correlation. ρ of 0 indicates there is no correlation between the ranks.

As shown in Table II, GenMAT obtains an average ρ of 0.94 for Halide Blur and an average ρ of 0.96 for Halide FFT. The high value of ρ indicates that the GenMAT predictor correctly ranks most of the variants with respect to their runtime. Therefore, the Halide schedule chosen by GenMAT—one that is ranked first—is likely to be the true best schedule or at least results in a runtime that is close to the minimum runtime.

TABLE II: Spearman's rank correlation coefficients for Halide Blur and Halide FFT on CPUs and GPUs.

Halide Blur					Halide FFT		
CPU			GPU		CPU		
Xeon	I7	I5	Tesla	Quadro	Xeon	I7	I5
0.87	0.91	0.96	0.99	0.95	0.99	0.95	0.95

[1]https://github.com/Naifeng/Auto-Tuner

TABLE III: Effects of accelerated profiling for Halide Blur on Xeon and I5.

	Xeon		I5	
Acceleration	No	Yes	No	Yes
Profiling inputs	$(n \times n), n \in A$	$(m \times m), m \in B$	$(n \times n), n \in A$	$(m \times m), m \in B$
Variant Selection inputs	$(n \times n), n \in A$	$(l \times k), l, k \in [2^{14}, 2^{15}]$	$(n \times n), n \in A$	$(l \times k), l, k \in [2^{14}, 2^{15}]$
Profiling Time	3318s	257s	406s	40s
Spearman's Coefficient	0.91	0.86	0.97	0.78
Runtime deviation	18%	**5%**	6%	**7%**

$A = \{2^{10}, 2^{11}, 2^{12}, 2^{13}, 2^{14}, 2^{15}\}, B = A \setminus \{2^{14}, 2^{15}\}$.

3) Overall Processing Time:

a) Selecting Variant: For matrix operations and Halide kernels, GenMAT can inference on 5000 candidate variants to select the best variant in 0.75 milliseconds, resulting in 0.15 microseconds per candidate. The minute processing time allows GenMAT to explore a large parameter space in time-constrained settings, thereby making GenMAT useful at both compile-time and runtime.

b) Accelerated Profiling: To demonstrate the accelerated profiling, we present the results of tuning the Halide Blur kernel, which operates on an input image. We evaluate if the performance model training on instances with input sizes from 2^{10} to 2^{13} can accurately predict on instances with input sizes from 2^{14} to 2^{15}. Since smaller inputs are executed quickly, it significantly shortens profiling time. Furthermore, the profiling step only uses square images as inputs, while the tuning is performed on rectangular images. We generated 250 instances for profiling and 750 candidates for variant selection. More candidates could be generated, but we restricted to a small number of candidates as we had to execute each of them to obtain the ground truth for evaluation.

As shown in Table III, the accelerated profiling significantly reduces profiling time (12.9× for Xeon and 10.2× for I5) while maintaining a satisfactory ρ (0.86 for Xeon and 0.78 for I5). It is noteworthy that the runtime deviation increases by only 1% for I5 and even decreases by 13% for Xeon, indicating that profiling on small regular inputs can generalize well to large relatively-irregular inputs.

Eventually, the user needs to make the decision of whether to trade performance prediction accuracy for a shorter profiling time. Accelerated profiling is necessary when profiling a program with large inputs is infeasible—for example, when the user can only allocate limited node-hours on a shared computational resource.

V. Limitations and Future Work

Minimal Re-Profiling: When using GenMAT, we assume that the target program's runtime environment during execution remains the same as the runtime environment during profiling. For example, if profiling is performed on exclusive use of the platform, then we assume that there will be no other co-scheduled programs during execution. Thus, if the runtime environment changes, we encourage the user to run GenMAT to profile again for accurate tuning. Benefiting from the lightweightness of the default performance prediction model and techniques to accelerate profiling, GenMAT can re-profile and re-train the predictor in a short amount of time. We envision that in the future, GenMAT can exploit the changes of the runtime environment so that little or no re-profiling needs to be performed.

Explainable Machine Learning: Our default predictor—NN+C—is a black-box approach that takes in data points and outputs predictions. Benefiting from this property, GenMAT can make predictions by only manipulating the inputs to the target program to profile, without knowing the specific architecture or the implementation itself. However, the black-box treatment of the programs makes it difficult to interpret the prediction results. For example, it is hard to analyze why the average percentage runtime deviation is high on one architecture and low on another. We aim to make the default predictor explainable.

Scaling: While we present our design for general programs, we envision that GenMAT can be leveraged to tune large workloads, frameworks, and systems. Because a small number of pre-defined tasks (e.g., matrix operations) cover a wide range of large workloads, GenMAT should be able to automatically extract pre-defined tasks and utilize known libraries implementing those tasks for variant selection instead of relying on the user to provide various implementations to choose from. Besides tuning multiple pre-defined tasks of a large workload, we will also apply GenMAT on systems where there are hundreds or thousands of configuration knobs that control the system, such as database management systems (DBMS) [18] and large-scale atomic/molecular massively parallel simulator (LAMMPS) [19].

VI. Conclusion

We have introduced GenMAT for automatic performance tuning of programs which are presented as meta-programs with exposed tunable parameters. The user only needs to specify the tunable parameters and add as low as three lines of code to start GenMAT, which accelerates profiling, accurately predicts performance, and selects the variant that produces a runtime close to the runtime of the true optimal choice. The fast processing time allows GenMAT to work at both compile-time and runtime. We implemented GenMAT in a modular fashion and open-sourced it to encourage further development.

References

[1] M. Frigo and S. G. Johnson, "The design and implementation of fftw3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005.

[2] R. C. Whaley and J. J. Dongarra, "Automatically tuned linear algebra software," in *SC'98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing*. IEEE, 1998, pp. 38–38.

[3] B. Steiner, C. Cummins, H. He, and H. Leather, "Value function based performance optimization of deep learning workloads," *arXiv preprint arXiv:2011.14486*, 2020.

[4] J. Hegarty, J. Brunhaver, Z. DeVito, J. Ragan-Kelley, N. Cohen, S. Bell, A. Vasilyev, M. Horowitz, and P. Hanrahan, "Darkroom: compiling high-level image processing code into hardware pipelines," *ACM Trans. Graph.*, vol. 33, no. 4, pp. 144–1, 2014.

[5] R. T. Mullapudi, A. Adams, D. Sharlet, J. Ragan-Kelley, and K. Fatahalian, "Automatically scheduling halide image processing pipelines," *ACM Transactions on Graphics (TOG)*, vol. 35, no. 4, pp. 1–11, 2016.

[6] A. Adams, K. Ma, L. Anderson, R. Baghdadi, T.-M. Li, M. Gharbi, B. Steiner, S. Johnson, K. Fatahalian, F. Durand *et al.*, "Learning to optimize halide with tree search and random programs," *ACM Transactions on Graphics (TOG)*, vol. 38, no. 4, pp. 1–12, 2019.

[7] H. Mao, M. Schwarzkopf, S. B. Venkatakrishnan, Z. Meng, and M. Alizadeh, "Learning scheduling algorithms for data processing clusters," in *Proceedings of the ACM Special Interest Group on Data Communication*. ACM New York, NY, USA, 2019, pp. 270–288.

[8] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze *et al.*, "{TVM}: An automated end-to-end optimizing compiler for deep learning," in *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, 2018, pp. 578–594.

[9] A. Hayashi, K. Ishizaki, G. Koblents, and V. Sarkar, "Machine-learning-based performance heuristics for runtime cpu/gpu selection," in *Proceedings of the principles and practices of programming on the Java platform*, 2015, pp. 27–36.

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

[11] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," *ACM SIGPLAN Notices*, vol. 48, no. 6, pp. 519–530, 2013.

[12] A. Srivastava, N. Zhang, R. Kannan, and V. K. Prasanna, "Towards high performance, portability, and productivity: Lightweight augmented neural networks for performance prediction," in *2020 IEEE 27th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. IEEE, 2020, pp. 21–30.

[13] F. Franchetti, T. M. Low, D. T. Popovici, R. M. Veras, D. G. Spampinato, J. R. Johnson, M. Püschel, J. C. Hoe, and J. M. Moura, "Spiral: Extreme performance portability," *Proceedings of the IEEE*, vol. 106, no. 11, pp. 1935–1968, 2018.

[14] H. Zhou, A. Srivastava, R. Kannan, and V. Prasanna, "Design and implementation of knowledge base for runtime management of software defined hardware," in *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2019, pp. 1–7.

[15] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[16] G. Guennebaud, B. Jacob *et al.*, "Eigen v3," http://eigen.tuxfamily.org, 2010.

[17] B. Schäling, *The boost C++ libraries*. Boris Schäling, 2011.

[18] D. Van Aken, A. Pavlo, G. J. Gordon, and B. Zhang, "Automatic database management system tuning through large-scale machine learning," in *Proceedings of the 2017 ACM International Conference on Management of Data*, 2017, pp. 1009–1024.

[19] J. Kadupitiya, G. C. Fox, and V. Jadhao, "Machine learning for parameter auto-tuning in molecular dynamics simulations: Efficient dynamics of ions near polarizable nanoparticles," *The International Journal of High Performance Computing Applications*, vol. 34, no. 3, pp. 357–374, 2020.

[20] J. Ragan-Kelley *et al.* Halide, a language for fast, portable computation on images and tensors. [Online]. Available: https://halide-lang.org/

APPENDIX

```
#include "Halide.h"
using namespace Halide;
int main(int argc, char **argv) {
      Func blur_x, blur_y;
      Var x, y, xi, yi;
      Func input;
      input(x,y) = x + y;
      blur_x(x, y) = (input(x-1, y) + input(x, y) + input(x+1, y))/3;
      blur_y(x, y) = (blur_x(x, y-1) + blur_x(x, y) + blur_x(x, y+1))/3;
      blur_y.tile(x, y, xi, yi, 256, 32).vectorize(xi, 8).parallel(y);
      blur_x.compute_at(blur_y, x).vectorize(x, 8);
      blur_y.realize(1024, 1024);
      printf("Success!\n");
      return 0;
}
```

Listing 6: Blur kernel written in Halide [20].

```
#include "Halide.h"
using namespace Halide;
int main(int argc, char **argv) {
      std::vector<int> tuning(argc);
      for (int i = 0; i < argc; ++i)
            tuning[i] = atoi(argv[i]);
      Func blur_x, blur_y;
      Var x, y, xi, yi;
      Func input;
      input(x,y) = x + y;
      blur_x(x, y) = (input(x-1, y) + input(x, y) + input(x+1, y))/3;
      blur_y(x, y) = (blur_x(x, y-1) + blur_x(x, y) + blur_x(x, y+1))/3;
      blur_y.tile(x, y, xi, yi, tuning[2], tuning[3]).vectorize(xi, tuning[4]).parallel(y);
      blur_x.compute_at(blur_y, x).vectorize(x, tuning[5]);
      blur_y.realize(tuning[1], tuning[1]);
      printf("Success!\n");
      return 0;
}
```

Listing 7: Blur kernel in the form of a meta-program.