ShaderPerFormer: Platform-independent Context-aware Shader Performance Predictor

ZITAN LIU, University of Science and Technology of China, China
YIKAI HUANG, University of Science and Technology of China, China
LIGANG LIU, University of Science and Technology of China, China

The ability to model and predict the execution time of GPU computations is crucial for real-time graphics application development and optimization. While there are many existing methodologies for graphics programmers to provide such estimates, those methods are often vendor-dependent, require the platforms to be tested, or fail to capture the contextual influences among shader instructions. To address this challenge, we propose ShaderPerFormer, a platform-independent, context-aware deep-learning approach to model GPU performance and provide end-to-end performance predictions on a per-shader basis. To provide more accurate predictions, our method contains a separate stage to gather platform-independent shader program trace information. We also provide a dataset consisting of a total of 54,667 fragment shader performance samples on 5 different platforms. Compared to the PILR and SH baseline methods, our approach reduces the average MAPE across five platforms by 8.26% and 25.25%, respectively.

CCS Concepts: • **Computing methodologies** → **Graphics systems and interfaces**.

Additional Key Words and Phrases: shader performance prediction, performance modeling, GPU

ACM Reference Format:
Zitan Liu, Yikai Huang, and Ligang Liu. 2024. ShaderPerFormer: Platform-independent Context-aware Shader Performance Predictor. *Proc. ACM Comput. Graph. Interact. Tech.* 7, 1, Article ?? (May 2024), 17 pages. https://doi.org/10.1145/3651295

1 INTRODUCTION

Shaders are crucial for generating sophisticated visual effects in real-time rendering applications. However, their performance varies significantly across different GPU platforms. Thus, understanding the nuances of shader performance on various platforms is essential for optimizing rendering efficiency.

Numerous methodologies have been developed for estimating GPU performance and offering optimization advice. Tools such as the Radeon Graphics Profiler [AMD 2023] and NSight Graphics [NVIDIA 2023] utilize GPU internal performance counters to provide shader optimization suggestions, while architecture simulators like Emerald [Gubran and Aamodt 2019] predict shader performance through architectural analyses. However, the effectiveness of these tools is limited by their vendor-specific or platform-dependent nature, necessitating individual executions on each GPU or dedicated simulation run. This specificity imposes a substantial burden on developers in the highly fragmented PC market, requiring separate modeling or profiling for each platform.

Prior research on shader optimization and simplification typically employs a straightforward method for estimating performance. He et al. [2015] estimate shader performance using a heuristic that accounts for scalar and texture operations within a shader—an approach that has been

Authors' addresses: Zitan Liu, jauntyliu@mail.ustc.edu.cn, University of Science and Technology of China, Hefei, China; Yikai Huang, earendil@mail.ustc.edu.cn, University of Science and Technology of China, Hefei, China; Ligang Liu, lgliu@ustc.edu.cn, University of Science and Technology of China, Hefei, China.

extensively adopted in later studies [Huo et al. 2022; Yuan et al. 2018]. However, the execution time of shader instructions is known to be influenced by a multitude of factors, such as compiler optimizations and the architectural characteristics of GPUs, including the cache system and the number of arithmetic units. Therefore, models that overlook these contextual factors are prone to producing inaccurate performance predictions.

Based on the observations above, we propose ShaderPerFormer, a platform-independent context-aware shader performance predictor that learns from shader performance samples, builds a proxy model for a particular GPU platform, and makes predictions using the proxy model built.

To ensure platform neutrality, we choose to only depend on the information that the application presented to the graphics API. SPIR-V [Khronos Group 2023b] is an intermediate representation (IR) format for shader programs in Vulkan API, and we choose to build our predictor input upon SPIR-V for three reasons: First, prior works on program understanding [Niu and Li 2023; Peng et al. 2021] have argued that the IR is a good candidate for exposing program characteristics on learning; Second, to make the entire predictor vendor-independent, the input format must decouple from actual GPU instructions emitted, which are frequently poorly documented in practice; Third, the SPIR-V ecosystem, built by the Khronos Group and various vendors, provides numerous utilities for program transformation, optimization, and fuzzing, which are beneficial to our task.

In general, the execution time of a particular instruction is not only related to the operation it performs but also its contextual information. For GPU native instructions, the execution time is influenced by the operations it performs, the data type of the operands, the availability of operands, and the occupancy of backend arithmetic and logic units. Further, for SPIR-V IR, it will undergo a series of transformations associated with local and global context information to lower into GPU native instructions. Those context-dependent behaviors at the SPIR-V level are manifested in that the execution time of an instruction is influenced by the sequence of instructions in which it is located. Therefore, we treat our problem as a sequential learning problem and use Transformer [Vaswani et al. 2017] as the network architecture for our predictor.

Despite being written in a Domain-Specific Language (DSL), shaders executed on modern hardware can include dynamic loops and branches, thereby rendering the language Turing-complete in a theoretical context. The halting problem, which is undecidable, can be constructed within a Turing-complete model of computation [Turing 1937]. Consequently, to enable accurate predictions of shader behavior, it is necessary to provide additional information to address this inherent complexity. Furthermore, the invocation counts of shader programs in the later stages of the pipeline are determined by the triangle input and prior pipeline stages, with varying counts significantly affecting total execution time. Based on these observations, we choose to do shader instruction tracing at the IR level to provide additional hints for our predictor to learn, which we subsequently demonstrate to be useful. It should be noted that the tracing stage can be done on any other platform that is capable of running the shaders to be tested, thereby preserving platform independence.

Our contributions can be summarized as follows:

- ShaderPerFormer, a Transformer based shader performance regression network that captures instruction contextual information
- A platform-independent basic block based instruction tracing framework at the IR level for giving instruction execution counts
- An accurate and diversified fragment shader performance dataset, with a total of 54,667 fragment shader performance samples on 5 different platforms

2 RELATED WORK

Our work is related to works on shader optimization and simplification, performance modeling, and programming language understanding.

2.1 Shader Optimization and Simplification

Shader are tiny programs executed on graphics hardware, so conventional compiler optimization techniques are widely employed in transforming shaders into GPU-native instructions. For Vulkan API, the optimizing, lowering, and emitting works are done in the user-mode driver provided by GPU vendors. Examples of those optimization procedures include dead code elimination, constant propagation, and inlining. However, Crawford and O'Boyle [2018] found that vendor drivers do not capture all offline optimization opportunities for shader programs. Crawford and O'Boyle [2019] also observed that most shaders could be optimized further given runtime inputs.

There are also works on lossy shader simplification [He et al. 2015; Huo et al. 2022; Liang et al. 2023; Sitthi-Amorn et al. 2011; Wang et al. 2014], which investigates how shaders can be simplified for better performance with respect to given error tolerance for shaded objects on specific camera settings. These works are mostly done with Genetic Programming (GP) and greedy search. For each optimization iteration, different lossy simplifications can be made, including substitution of expressions with average value or zero [Sitthi-Amorn et al. 2011], moving between different pipeline stages [He et al. 2015; Wang et al. 2014]. Huo et al. [2022] further proposed ShaderTransformer, a deep learning based quality predictor for simplified shader variants, to speed up the mutate-simplify iteration.

It is worth noting that current works on lossy shader simplification either use a simple heuristic to represent the runtime cost of shaders [He et al. 2015; Huo et al. 2022; Yuan et al. 2018] or require separate profile runs for performance measurement [Wang et al. 2014]. We believe the cost model we propose in this paper could be applied in this field, giving the ability to simplify shaders with respect to specific GPU platforms without dedicated executions.

2.2 Performance Modeling

Performance modeling is crucial for many design decisions; consequently, extensive research has been dedicated to the estimation of program performance on various computing platforms, including CPU, GPGPU, and other Domain-Specific Architectures (DSAs). Predominantly, the field employs two categories of models: analytical-based models and learning-based models. Works on graphics performance modeling are also discussed in this section.

Analytical-based models can be roughly divided into *architecture simulators* and *simple analytical models*. *Architecture simulators* are hand-crafted simulators that are usually cycle-accurate for the architecture components that are related to performance. For example, NaviSim [Bao et al. 2023] is a simulator for AMD RDNA architecture, giving predictions on compute kernel execution time by writing simulation procedures for wavefront dispatching, RDNA instruction issuing, decoding, and execution. GPGPU-Sim [Bakhoda et al. 2009; Khairy et al. 2020] is a simulator for CUDA / OpenCL workloads that runs on NVIDIA GPUs. *Simple analytical models*, on the other hand, tend to use simple abstractions that capture the common aspects of the architecture family they model. Popular choices of these models include the roofline model [Konstantinidis and Cotronis 2017; Williams et al. 2009] and the pipeline model [Abel and Reineke 2022; Abel et al. 2023; Lemeire et al. 2023]. To summarize, analytical-based models in general require an in-depth understanding of the architecture they model as well as extensive micro-benchmarks related to the characteristics they capture.

Learning-based models, by leveraging machine learning, reduce the need for detailed architecture inspection and are easier to adapt to new architectures with the appropriate measurements. Several works estimate basic block throughput on CPU using either hierarchical LSTM [Mendis et al. 2019] or Graph Neural Network (GNN) [Sykora et al. 2022] capturing from in-the-wild application profiles [Chen et al. 2019; Mendis et al. 2019]. Zhai et al. [2023] modeled the tensor program tuning problem as a Natural Language Processing (NLP) regression task.

A portion of performance models focus on modeling graphics system performance. O'neal et al. [2017] have built a predictive model using random forests for GPU graphics workloads targeting Intel Skylake integrated graphics, using the internal cycle-accurate simulator as the target for alignment. Gubran and Aamodt [2019] proposes Emerald, an SoC simulator built upon GPGPU-Sim [Bakhoda et al. 2009] for graphics and GPGPU system modeling. Sembrant et al. [2017] builds GLTraceSim, a framework for generating and replaying CPU and GPU memory traces under graphics workload.

Our work proposes a learning-based model, as GPU tends to have big architecture differences across vendors and generations. To our knowledge, there are currently no publicly available cross-vendor GPU performance models that focus on shader performance modeling and operate in a black-box fashion, and this work aims to bridge this gap.

2.3 Programming Language Understanding

Programming languages (PL) can be seen as specific forms of languages, and many methods originated from the Natural Language Processing (NLP) community have been adopted into the understanding of PL. Utilizing Transformer [Vaswani et al. 2017], the BERT [Devlin et al. 2019] model and its paradigm have inspired various code understanding models like CodeBERT [Feng et al. 2020] and GraphCodeBERT [Guo et al. 2021]. Our work similarly employs Transformer to capture the contextual relevance within shader instructions.

As is surveyed by Niu et al. [2022], typical tasks for PL understanding models include clone detection, code classification, defect detection, and code retrieval. There is also a task specifically designed to capture the performance-related semantics, for example, the OpenCL Heterogeneous Mapping task brought by Cummins et al. [2017], which is studied by later works [Cummins et al. 2021; Peng et al. 2021]. By collecting our shader performance dataset, we hope to inspire more work on incorporating performance semantics into code intelligence models.

3 PIPELINE OVERVIEW

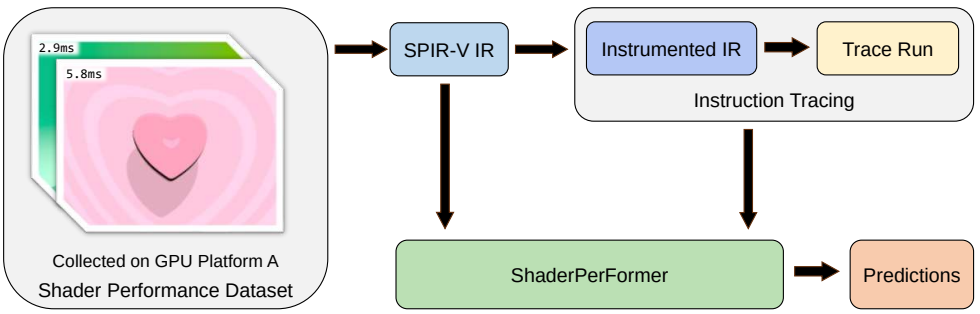


Fig. 1. Overview of our proposed pipeline. The GPU platform A is the GPU to be modeled, and the dataset collected on GPU Platform A is used to train our model.

Figure 1 contains the pipeline of our work. We first collect a shader performance dataset (Section 4), which contains shader source codes, shader uniform input, and the execution time on the target GPU platform. Our goal is to build a proxy model ("ShaderPerFormer") for this target GPU platform. Our proxy model requires two types of input, original instructions in IR format and associated instruction traces. The first is gathered by using stock compiler glslang [Khronos Group 2023a], and the second is gathered by a separate instruction tracing stage (Section 5). The instruction tracing stage consists of instrumenting the original IR and running the modified IR, but the latter execution doesn't need to be in the target platform. We then feed these inputs into our model, ShaderPerFormer, which outputs predictions of the time required to draw a frame. Measured values in our dataset are used to optimize our proxy model during training.

4 SHADER PERFORMANCE DATASET

We leverage shaders from Shadertoy [Quilez and Jeremias 2013] to train and validate our performance predictor. The shaders on Shadertoy always assume a quad to be put on the vertex stage, and all the actual logics are done on the fragment stage. This left us with a good chance to thoroughly explore the performance of fragment shaders.

Despite the lack of geometry input, Shadertoy users have created numerous shaders that render non-trivial results by using techniques like Ray Marching [Hart 1996]. Some of the Shadertoy shaders are also explored in Yang et al. [2022]. Therefore, we think Shadertoy shaders are representative in terms of fragment shader performance modeling.

Collecting. We have collected 27,911 shaders from Shadertoy using the API interface the website provides.

Compiling. Shaders downloaded from Shadertoy have an entry-point function called `mainImage`. We wrap the mainImage into a main function used as the new entry point and provide uniform variables like `iTime` and `iResolution` in a uniform block. We then compile the shaders into SPIR-V using glslang.

Profiling. We wrote a Python extension `vkExecute` to measure the performance, and the core profiling logic is described in Algorithm 1. To eliminate CPU submission overhead, we use GPU-based timestamp counter to track the time used. The `unitTs` represents the time elapsed per counter increase, which is platform-dependent. To amortize GPU drawing overhead, we additionally emit draw commands by `num_cycles` times in between the timestamp counter reads, because many shaders we collect are very fast (>1000fps) on modern platforms. We also do `num_trials` times of GPU command submissions and store all the results for later analysis. In all of our profiling runs, we lock the shader and the memory clock frequency on the platforms to be tested using the interfaces vendors provided.

Error recovery. Shadertoy does not verify whether a user-uploaded shader is legitimate, so compilation errors will occur for some shaders. Also, since shaders have grown into a Turing-complete DSL, its execution times are not guaranteed to be bounded, so modern GPU drivers are likely to raise a per-engine or full GPU reset when rendering commands in a queue took too long to finish. When such a reset event happens, the Vulkan context of the profiler will get invalidated. We conduct shader profiling into separate processes to avoid the management program being killed.

5 SHADER INSTRUCTION TRACING

Shader instruction traces are used to provide execution flow information to the predictor for a given shader uniform input.

Algorithm 1 Pseudocode for the profiling routine

```
function ProfileShaderOnce(num_cycles)
    Do image memory barrier, layout transition and reset timestamp query pool
    Wait for previous commands to finish
    cmdBuf ← allocateCmdBufFromPool()
    Emit write timestamp ts_1 command into cmdBuf
    for i ← 1 to num_cycles do
        Emit bind graphics pipeline and descriptors command into cmdBuf
        Emit draw command into cmdBuf
    end for
    Emit write timestamp ts_2 command into cmdBuf
    Submit to command queue
    Wait for previous commands to finish
    return (ts_2 − ts_1) × unitTs
end function
function ProfileShader(num_cycles, num_trials)
    results ← []
    for i ← 1 to num_trials do
        results[i] ← ProfileShaderOnce(num_cycles)
    end for
    return results
end function
```

To generate an accurate trace, one possible method would be writing an interpreter for all shader GPU native instructions. However, as our goal is to build a cross-vendor shader performance predictor, such a method would be challenging to implement due to the diversity of GPU vendors and models. Therefore, we choose to trace at the SPIR-V IR level.

The decision to trace in SPIR-V is two-fold: On one hand, the SPIR-V-based tracing gives a unified solution for all target GPU platforms, and since the results are generally consistent across different devices, we can safely generate traces for other platforms that we're predicting without accessing the platform to be predicted. However, tracing at the SPIR-V level also imposes a challenge that the eventual execution flow may deviate from the IR execution flow because of driver compiler optimizations. We further solve this problem by using a deep neural network to estimate those context-related transformations.

To trace at the SPIR-V level, one could either implement a SPIR-V interpreter (e.g. Talvos [Tal 2019]) or instrument additional tracing commands into the original shader program and run the modified shader. We chose the latter because this allows us to reuse the existing software stack Khronos Group has created on Vulkan and SPIR-V.

Fragment shader programs are executed once per fragment, so a per-invocation fine-grained execution flow would keep track of the exact sequence of instruction execution for all invocations. This would be costly to gather and store. For simplicity, we only record the execution count summed from all invocations for each of the instructions in the SPIR-V shader.

Our goal now becomes getting the execution count for each reachable instruction in the program. However, not all instructions can be the source and target of a branching instruction in the program, and those instructions that are guaranteed to be sequentially executed form a basic block. Therefore, instructions inside a basic block share the same execution count, and we choose to instrument only at the beginning of each basic block that is reachable from the shader entry-point function.

```
%291 = OpLabel
%432 = OpAccessChain %_ptr_StorageBuffer_ulong %basic_block_trace_buffer %uint_0 %
    uint_8
%433 = OpAtomicIAdd %ulong %432 %uint_1 %uint_0 %ulong_1
%292 = OpLoad %float %209
%293 = OpFOrdGreaterThanEqual %bool %292 %float_1
       OpSelectionMerge %319 None
       OpBranchConditional %293 %294 %319
```

Fig. 2. Example of modified IR. `OpLabel` is the marker for the beginning of a basic block in SPIR-V. `%432` and `%433` are new instructions inserted at the beginning of this basic block to increment the counter. `%basic_block_trace_buffer` is declared to be a Vulkan storage buffer in a predefined descriptor set slot at the beginning of this SPIR-V module. The integer constant `%uint_8` is given as the offset to the counter array for this basic block presented. `%ulong` stands for 64-bit unsigned integer.

We implemented a separate instrumentation pass in SPIRV-Tools [Khronos Group 2023c], a widely used SPIR-V transformation and optimization framework. We first label and give a unique index for each basic block that is reachable from the entry point function. We then emit an atomic instruction incrementing the counter in the counter array offset by index by 1. Figure 2 presents an example of a modified basic block. We use `uint64` to avoid potential counter overflow.

After instrumentation, we allocate a separate storage buffer, place this buffer into the descriptor set, and run our instrumented shader to get the execution count for each basic block of the original shader.

6 PERFORMANCE MODELING

6.1 Baseline Methods

He et al. [2015] uses the following simple heuristics to estimate the performance of the shader in one pipeline stage,

$$t = N_{\text{scalarOps}} + 100 \times N_{\text{textureOps}} \tag{1}$$

where $N_{\text{scalarOps}}$ is the number of scalar operations present in the shader and $N_{\text{textureOps}}$ is the number of texture operations present in the shader. The factor 100 is a cost factor set for texture operations for convenience.

Extending this idea further, we present two methods as baselines,

$$t = c_{\text{inst}} \cdot N_{\text{inst}} \tag{2}$$

$$t = \sum_{i=1}^{M} c_{\text{inst}_i} \cdot N_{\text{inst}_i} \tag{3}$$

Simple Heuristics (SH). We assign an amortized cost c_{inst} for each instruction in the IR. Therefore, the total time t of a shader that has executed N_{inst} instructions is calculated as in Equation 2.

Per Instruction Linear Regression (PILR). We further assume a different cost c_{inst_i} for each type of opcode in the IR. We then do the summation using the number of instructions of type opcode$_i$ and its cost c_{inst_i} as in Equation 3.

c_{inst} and c_{inst_i} are parameters to be optimized for each platform to be modeled, and both methods can be easily optimized with the least squares method when combined with MSE loss. However, since our shader performance samples span a large range of time, simply using the MSE will heavily

bias toward slow shader samples. We use weighted MSE loss (Equation 4) to combine both fast and slow shader samples.

$$L_{\text{WMSE}} = \sum_i \frac{1}{t_{\text{real}_i}} (t_{\text{real}_i} - t_{\text{pred}_i})^2 \tag{4}$$

6.2 ShaderPerFormer

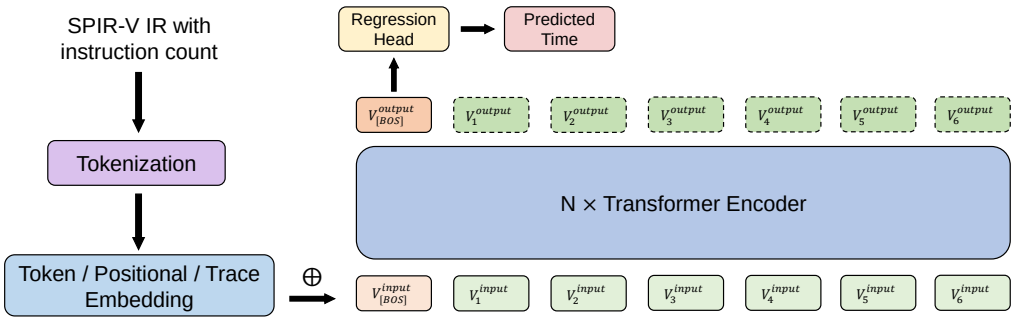


Fig. 3. Overview of our ShaderPerFormer model.

Note that both baseline methods assume the time consumption of each SPIR-V instruction is not relevant to other instructions. To this end, we propose a ShaderPerFormer, a Transformer-based sequential learning model that utilizes this type of contextual information of IR instructions.

Figure 3 gives an overview of our ShaderPerFormer model. We first send SPIR-V IR with associated instruction traces into a dedicated tokenizer. After tokenization, we generate and sum the embedding of tokens, the positions of the tokens, and their associated execution counts ("traces"), and feed those into the input of N layers of the Transformer Encoder. We then extract the first token from the output of the last Transformer Encoder, feed it into a regression head, and obtain the predicted time.

6.2.1 Tokenization. Traditional NLP models rely on tokenizers (e.g. BPE [Sennrich et al. 2016], SentencePiece [Kudo and Richardson 2018]) to convert from text to a sequence of integers named as tokens. Since the SPIR-V format is well-defined in its specification, we believe a dedicated tokenizer will be more efficient.

Our tokenizer, written in C++ as a Python extension, parses the SPIR-V, and does the tokenization function-by-function. We start tokenization from the entry point function, and then all functions that are reachable from the entry point function in a depth-first order. We tokenize instruction-by-instruction inside a function. Also, a separate `[BOS]` token is added at the beginning of the tokenized program.

In SPIR-V instructions, opcode and ID operands are of 32-bit integers, and we map them to tokens 1:1. For some literal operands that span an arbitrary number of bytes, we map each byte to token 1:1.

6.2.2 Embedding Generation. As in Figure 4, we generate the embedding vector for token $\mathbf{V}_{t_i}^{\text{tok}}$ and the embedding vector for absolute positions $\mathbf{V}_i^{\text{pos}}$ by looking up a 1:1 mapped learnable embedding table. However, as instruction trace counts can range from 0 to $2^{63} - 1$, managing a lookup table like tokens and absolute positions is not feasible in practice.

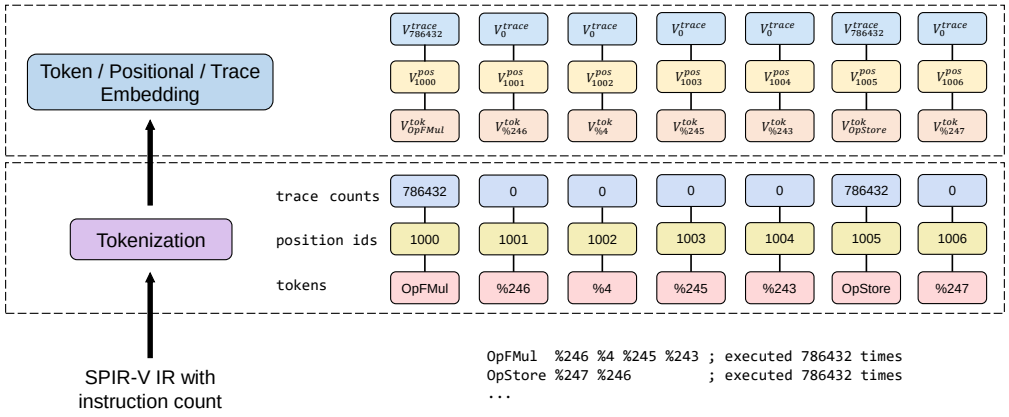


Fig. 4. The tokenization and embedding generation process of ShaderPerFormer. For readability, the tokenized opcodes and operands are shown in their text form. This figure shows a piece of IR starting from position 1000, and instructions are executed 786432, which is 1 per fragment in 1024×768 resolution. The tokenization results for operands of OpStore are omitted.

Similar to numerical encodings in Born and Manica [2023] but more simple, we directly convert the count to be encoded into binary form, and pad additional dimensions with 0. Given $\overline{x_i}$ representing the i^{th} digit of x's binary representation, the trace embedding $\mathbf{V}_x^{\text{trace}}$ can be written as

$$\mathbf{V}_x^{\text{trace}} = [\overline{x_0}, \overline{x_1}, \ldots, \overline{x_{63}}, 0, \ldots, 0]. \tag{5}$$

The final embedding is calculated by

$$\mathbf{V}_i^{\text{input}} = \mathbf{V}_{t_i}^{\text{tok}} + \mathbf{V}_i^{\text{pos}} + \mathbf{V}_{m_i}^{\text{trace}}, \tag{6}$$

where t_i and m_i are the token and the trace count of position i, respectively.

6.2.3 Encoder and Output Head. We use 9 layers of Transformer Encoder, with hidden dimension 768 and 12 attention heads. We then take the hidden vector of the first token (which is fixed to be `[BOS]`) and send it into a regression head for output predictions.

The regression head is calculated as

$$\text{RegressionHead}(x) = \text{Linear}(\text{Dropout}(\text{Tanh}(\text{Linear}(\text{Dropout}(x))))). \tag{7}$$

6.2.4 Normalization and Loss Function. Since our performance samples vary from 10^{-6} to 10^{-1} seconds, optimizing w.r.t. MSE using mini-batch directly without normalization can lead to loss instabilities. Therefore, we choose to do log normalization for our data input, which is to do $\log(t)$ for every time t in the dataset.

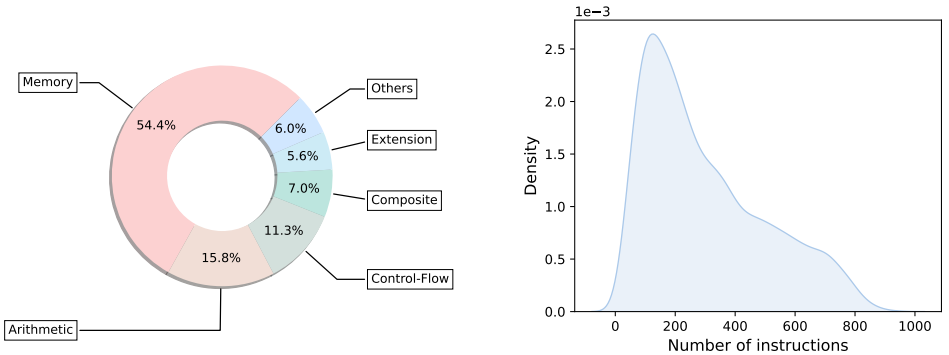
We use MSE for our loss function. Considering the normalization we use, the loss function can also be called MSLE (Mean Squared Logarithmic Error).

7 EVALUATION

7.1 Dataset

We fixed the shader uniform inputs by setting `iTime` and `iFrame` to 1 and collected our dataset on 5 different GPU platforms using the method described in Section 4, which covers both desktop and laptop GPUs, as well as discrete and integrated GPUs. The detailed environment information can be seen in Table 4 in the Appendix.

We additionally filtered out shader samples that use the feature not yet supported or those that are likely to be buggy. As GLSL tends to have subtle implementation differences across platform drivers, and different platforms exhibit different driver timeout definitions, the number of shaders that are left valid as samples differ slightly across the platform we tested. The results are in Table 1. Figure 5 gives an insight into shaders that passed the filters at the IR level.



(a) Breakdown of Shader Instruction Types (b) Shader Instruction Count Distribution

Fig. 5. Statistics for shader samples that passed all filters in RTX3060. The SPIR-V instruction types are extracted from the SPIR-V Specification.

Table 1. Various filters and the number of shaders left after each filter. *Not Black or White* and *Within Time Limit* are used to filter out potentially buggy shaders. *Within Tokenized Length Limit* is used to filter out shaders that are too long to be fed into our model.

Filters	RTX3060	RX7900GRE	UHD630	RTX4060	GTX1660Ti
Shadertoy Shaders			27911		
Single-pass Shaders			20669		
Run Successful	13871	14084	13856	14094	13913
Trace Successful	13870	14080	13850	14090	13913
Not Black or White	13376	13925	13364	13484	13379
Within Tokenized Length Limit	10794	11360	10784	10935	10806
Within Time Limit	10794	11360	10781	10935	10797

We observed non-linear performance scaling patterns in our dataset. Figure 6(a) gives the distribution of performance samples with respect to time mean. On the Intel UHD Graphics 630, we not only observed slower performance but also a distribution shape that is significantly different from those of the other GPUs. This suggests that using linear scaling of shader execution times from other platforms to estimate times on the UHD 630 platform is likely to be challenging.

We set `num_cycles` and `num_trials` to be 30 and 10, respectively. We also checked the CV (Coefficient of Variation) of our 10 measurements for each shader to make sure the performance data we gather are of adequate precision. The CV is defined as the ratio of the standard deviation by mean (σ/μ), which captures the relative precision of our data gathered. Figure 6(b) gives the

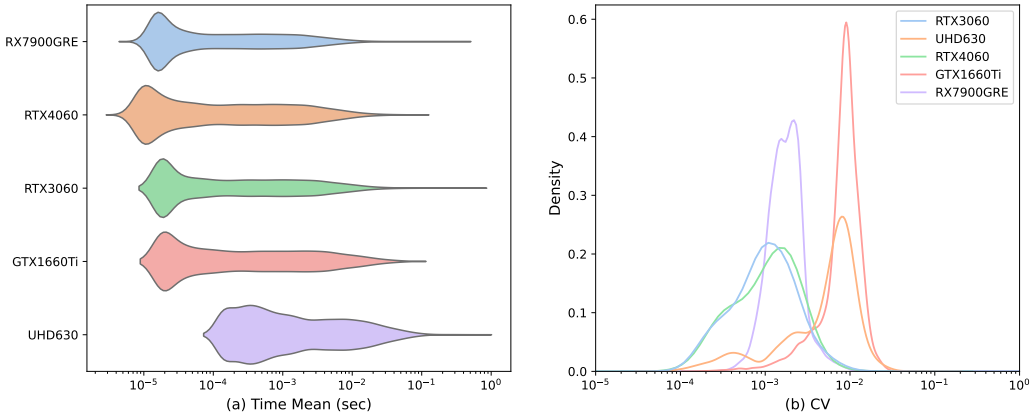


Fig. 6. (a) Sample distribution w.r.t. time for all measured GPU platforms. (b) Distribution of CV for all measured GPU platforms.

CV for all measured platforms, from which we can conclude that our CV is less than 3% for most platforms we measure. We think this is adequate for both real-world usage and later analysis.

7.2 ShaderPerFormer

7.2.1 Training and Evaluation Splitting. We split our performance samples into a training set (80%), a test set (5%) and a validation set (15%). The splitting is consistent across all architectures we measure.

We use one NVIDIA GeForce RTX 4090 to train our model. We use Adam [Kingma and Ba 2017] as our optimizer, a learning rate of 3×10^{-5}, and the default Adam configurations are used ($\beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 1 \times 10^{-8}$). We further use linear learning rate warm-up with a warm-up ratio of 0.1. We train each model with 50 epochs with an effective batch size of 40 and select the model that has the best MAPE (Mean Absolute Percentage Error) on the test set at the end of each epoch.

7.2.2 Baseline Comparisons. We evaluated two baseline methods and ShaderPerFormer in the above configuration. We report the MAPE between the predicted time and the measured time in validation sets. The results are in Table 2.

ShaderPerFormer is the most accurate method among all methods tested, which outperforms the PILR and the SH baseline method in terms of average MAPE on 5 platforms by 8.26% and 25.25% respectively.

We additionally report the Spearman correlation coefficient between the predicted results and the measured results for each method. Spearman correlation is especially useful for shader optimization tasks, in which optimizers typically only need to determine which shader is the fastest but not the absolute performance. Our method has the best rank correlation compared with baseline methods in all platforms tested.

Figure 7 shows the heatmap plot relating the execution time measured and predicted for all methods under all 5 platforms tested. We divided the measured and predicted time range into 50×50 bins on a log scale, and the color in each bin represents the count of time samples. A perfect estimator would lie on the line $y = x$. It is observed that our proposed method in general exhibits a higher density near the identity line for most architectures compared to other baseline methods.

Table 2. Metrics evaluated at validation sets of each GPU platform. SH represents Simple Heuristics, PILR represents Per Instruction Linear Regression, and ShaderPerFormer represents our proposed method. The best metric in the group is bolded.

	SH		PILR		ShaderPerFormer (**Ours**)	
Platform	MAPE	Spearman	MAPE	Spearman	MAPE	Spearman
RTX3060	75.26%	0.9584	46.90%	0.9200	**40.44%**	**0.9642**
UHD630	30.73%	0.9594	27.32%	0.9676	**26.60%**	**0.9722**
RTX4060	81.17%	0.9588	55.60%	0.9290	**44.62%**	**0.9632**
GTX1660Ti	82.56%	0.9607	54.99%	0.9353	**41.59%**	**0.9681**
RX7900GRE	36.32%	0.9510	36.28%	0.9433	**26.54%**	**0.9578**
Average	61.21%	—	44.22%	—	**35.96%**	—

7.2.3 Trace Ablation Study. To examine the usefulness of the instruction trace counts we propose, we evaluated the performance of our method along with baseline methods without the trace.

To ensure a fair comparison, we leave all the network architectures as-is and set the instruction counts of all positions to 1. This will make trace embedding of all positions equal to $\mathbf{V}_1^{\text{trace}} = [1, 0, \ldots, 0]$.

For baseline methods, we assume that each instruction in the shader is executed once and only once. The rest of the methods remain unchanged.

As is shown in Table 3, instruction traces are shown to be useful for both ShaderPerFormer and Per Instruction Linear Regression. Simple Heuristics also gains its benefits for some of the platforms tested.

Table 3. Ablation study on trace carried out on ShaderPerFormer and baseline methods. SH represents Simple Heuristics, PILR represents Per Instruction Linear Regression, and ShaderPerFormer represents our proposed method. The best metrics are bolded.

Method	**Platform**	**MAPE**		**Spearman**	
		Without Trace	With Trace	Without Trace	With Trace
SH	RTX3060	76.02%	**75.26%**	0.7131	**0.9584**
	UHD630	66.97%	**30.73%**	0.7152	**0.9594**
	RTX4060	**79.54%**	81.17%	0.7118	**0.9588**
	GTX1660Ti	**80.20%**	82.56%	0.7142	**0.9607**
	RX7900GRE	68.65%	**36.32%**	0.6857	**0.9510**
PILR	RTX3060	86.43%	**46.90%**	0.8551	**0.9200**
	UHD630	72.87%	**27.32%**	0.8503	**0.9676**
	RTX4060	90.39%	**55.60%**	0.8563	**0.9290**
	GTX1660Ti	90.26%	**54.99%**	0.8571	**0.9353**
	RX7900GRE	81.19%	**36.28%**	0.8388	**0.9433**
ShaderPerFormer (**Ours**)	RTX3060	81.02%	**40.44%**	0.8608	**0.9642**
	UHD630	70.42%	**26.60%**	0.8749	**0.9722**
	RTX4060	98.10%	**44.62%**	0.7518	**0.9632**
	GTX1660Ti	109.60%	**41.59%**	0.8784	**0.9681**
	RX7900GRE	51.16%	**26.54%**	0.8448	**0.9578**

8 DISCUSSION AND LIMITATION

There are a few limitations to the dataset and associated evaluation process in our work. Although we believe in the generality of our proposed framework, we currently only validated our methods on fragment shaders that don't involve texture sampling or multi-pass shading. Furthermore, the shaders collected from Shadertoy for validating our proposed method may exhibit performance characteristics that differ from those typically encountered in other fields that use shaders, such as games.

We are also aware of the fact that the accuracy of our predictor still has room for improvement, given the fluctuations in our performance data, which are measured below 3%. As discussed in Section 1, there are a multitude of factors that influence the performance of shader programs. However, the current dataset may not be sufficient to fully reveal these performance characteristics to our proposed model. One possible way to improve the model's performance might involve data augmentation. For example, mutating or performing offline optimization on shaders and running them could expose more information to the neural network. Furthermore, a fine-grained per-invocation SPIR-V trace might be useful for better exploiting characteristics like instruction latency hiding.

Another perspective that may be amenable to enhancement is the interpretability of performance data. Although our model is capable of furnishing relatively accurate predictions on the target platform, the task of pinpointing the reasons behind the performance bottlenecks of the current shader (e.g., memory bound, ALU bound, etc.) still necessitates manual intervention by programmers.

Looking ahead, we plan to expand ShaderPerFormer to include predictions of the total time for rendering a frame, adding the modeling of the fixed components of the graphics pipeline into our analysis. Furthermore, we suggest that our framework could be adapted to fit into larger foundation models, potentially improving their ability to predict performance-related outcomes and optimizing user-supplied shaders in a fully autonomous way.

9 CONCLUSION

We present ShaderPerFormer, a platform-independent context-aware data-driven framework for shader performance estimation. The accuracy of our model surpasses methods that only model instruction costs but not their contexts. It achieves its level of accuracy by using a Transformer network to model the contextual behavior between IR instructions and their execution traces.

The code, trained models, and associated dataset snapshots of this paper can be downloaded from https://github.com/libreliu/ShaderPerFormer. By open-sourcing our work, we hope to inspire more advancements in graphics performance tooling that every developer can benefit from.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable comments and suggestions. We thank authors on Shadertoy website for creating beautiful shaders. We also thank Jiyan He (University of Science and Technology of China) for providing helpful insights into this problem. This work is supported by the National Key R&D Program of China (2022YFB3303400) and the National Natural Science Foundation of China (62025207).

A DETAILED INFORMATION ON PLATFORMS TESTED

We collected our performance samples on the 5 platforms described in Table 4.

Table 4. The full name and the type of the GPU platforms to be tested.

Abbreviation	Full Name	Vendor	Type
RTX3060	NVIDIA GeForce RTX 3060	NVIDIA	Desktop Discrete
UHD630	Intel UHD Graphics 630 (CML GT2)	Intel	Desktop Integrated
RTX4060	NVIDIA GeForce RTX 4060 Laptop GPU	NVIDIA	Laptop Discrete
GTX1660Ti	NVIDIA GeForce GTX 1660 Ti	NVIDIA	Laptop Discrete
RX7900GRE	AMD Radeon RX 7900 GRE	AMD	Desktop Discrete

REFERENCES

2019. Talvos: A dynamic-analysis framework and debugger for Vulkan/SPIR-V programs. https://github.com/talvos/talvos. Accessed: 2024-01-01.

Andreas Abel and Jan Reineke. 2022. UiCA: Accurate Throughput Prediction of Basic Blocks on Recent Intel Microarchitectures. In *Proceedings of the 36th ACM International Conference on Supercomputing* (Virtual Event) *(ICS '22)*. Association for Computing Machinery, New York, NY, USA, Article 33, 14 pages. https://doi.org/10.1145/3524059.3532396

Andreas Abel, Shrey Sharma, and Jan Reineke. 2023. Facile: Fast, Accurate, and Interpretable Basic-Block Throughput Prediction. In *2023 IEEE International Symposium on Workload Characterization (IISWC)*. 87–99. https://doi.org/10.1109/IISWC59245.2023.00023

AMD. 2023. Radeon Graphics Profiler. https://gpuopen.com/rgp/. Accessed: 2024-01-01.

Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt. 2009. Analyzing CUDA workloads using a detailed GPU simulator. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software*. 163–174. https://doi.org/10.1109/ISPASS.2009.4919648

Yuhui Bao, Yifan Sun, Zlatan Feric, Michael Tian Shen, Micah Weston, José L. Abellán, Trinayan Baruah, John Kim, Ajay Joshi, and David Kaeli. 2023. NaviSim: A Highly Accurate GPU Simulator for AMD RDNA GPUs. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques* (Chicago, Illinois) *(PACT '22)*. Association for Computing Machinery, New York, NY, USA, 333–345. https://doi.org/10.1145/3559009.3569666

Jannis Born and Matteo Manica. 2023. Regression Transformer enables concurrent sequence regression and generation for molecular language modelling. *Nature Machine Intelligence* 5, 4 (01 Apr 2023), 432–444. https://doi.org/10.1038/s42256-023-00639-z

Yishen Chen, Ajay Brahmakshatriya, Charith Mendis, Alex Renda, Eric Atkinson, Ondřej Sýkora, Saman Amarasinghe, and Michael Carbin. 2019. BHive: A Benchmark Suite and Measurement Framework for Validating x86-64 Basic Block Performance Models. In *2019 IEEE International Symposium on Workload Characterization (IISWC)*. 167–177. https://doi.org/10.1109/IISWC47752.2019.9042166

Lewis Crawford and Michael O'Boyle. 2018. A Cross-platform Evaluation of Graphics Shader Compiler Optimization. In *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 219–228. https://doi.org/10.1109/ISPASS.2018.00035

Lewis Crawford and Michael O'Boyle. 2019. Specialization Opportunities in Graphical Workloads. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 272–283. https://doi.org/10.1109/PACT.2019.00029

Chris Cummins, Zacharias V. Fisches, Tal Ben-Nun, Torsten Hoefler, Michael F P O'Boyle, and Hugh Leather. 2021. ProGraML: A Graph-based Program Representation for Data Flow Analysis and Compiler Optimizations. In *Proceedings of the 38th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 139)*, Marina Meila and Tong Zhang (Eds.). PMLR, 2244–2253. https://proceedings.mlr.press/v139/cummins21a.html

Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. 2017. End-to-End Deep Learning of Optimization Heuristics. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 219–232. https://doi.org/10.1109/PACT.2017.24

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, Jill Burstein, Christy Doran, and Thamar Solorio (Eds.). Association for Computational Linguistics, Minneapolis, Minnesota, 4171–4186. https://doi.org/10.18653/v1/N19-1423

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, Trevor Cohn, Yulan He, and Yang Liu (Eds.). Association for

Computational Linguistics, Online, 1536–1547. https://doi.org/10.18653/v1/2020.findings-emnlp.139

Ayub A. Gubran and Tor M. Aamodt. 2019. Emerald: Graphics Modeling for SoC Systems. In *Proceedings of the 46th International Symposium on Computer Architecture* (Phoenix, Arizona) *(ISCA '19)*. Association for Computing Machinery, New York, NY, USA, 169–182. https://doi.org/10.1145/3307650.3322221

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. GraphCodeBERT: Pre-training Code Representations with Data Flow. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net. https://openreview.net/forum?id=jLoC4ez43PZ

John C. Hart. 1996. Sphere tracing: a geometric method for the antialiased ray tracing of implicit surfaces. *The Visual Computer* 12, 10 (01 Dec 1996), 527–545. https://doi.org/10.1007/s003710050084

Yong He, Tim Foley, Natalya Tatarchuk, and Kayvon Fatahalian. 2015. A System for Rapid, Automatic Shader Level-of-Detail. *ACM Trans. Graph.* 34, 6, Article 187 (nov 2015), 12 pages. https://doi.org/10.1145/2816795.2818104

Yuchi Huo, Shi Li, Yazhen Yuan, Xu Chen, Rui Wang, Wenting Zheng, Hai Lin, and Hujun Bao. 2022. ShaderTransformer: Predicting Shader Quality via One-Shot Embedding for Fast Simplification. In *ACM SIGGRAPH 2022 Conference Proceedings* (Vancouver, BC, Canada) *(SIGGRAPH '22)*. Association for Computing Machinery, New York, NY, USA, Article 44, 9 pages. https://doi.org/10.1145/3528233.3530722

Mahmoud Khairy, Zhesheng Shen, Tor M. Aamodt, and Timothy G. Rogers. 2020. Accel-Sim: An Extensible Simulation Framework for Validated GPU Modeling. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 473–486. https://doi.org/10.1109/ISCA45697.2020.00047

Khronos Group. 2023a. glslang. https://github.com/KhronosGroup/glslang. Accessed: 2024-01-01.

Khronos Group. 2023b. SPIR-V Specification. https://registry.khronos.org/SPIR-V/specs/unified1/SPIRV.html. Accessed: 2024-01-01.

Khronos Group. 2023c. SPIRV-Tools. https://github.com/KhronosGroup/SPIRV-Tools. Accessed: 2024-01-01.

Diederik P. Kingma and Jimmy Ba. 2017. Adam: A Method for Stochastic Optimization. arXiv:1412.6980 [cs.LG]

Elias Konstantinidis and Yiannis Cotronis. 2017. A quantitative roofline model for GPU kernel performance estimation using micro-benchmarks and hardware metric profiling. *J. Parallel and Distrib. Comput.* 107 (2017), 37–56. https://doi.org/10.1016/j.jpdc.2017.04.002

Taku Kudo and John Richardson. 2018. SentencePiece: A simple and language independent subword tokenizer and detokenizer for Neural Text Processing. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, Eduardo Blanco and Wei Lu (Eds.). Association for Computational Linguistics, Brussels, Belgium, 66–71. https://doi.org/10.18653/v1/D18-2012

Jan Lemeire, Jan G. Cornelis, and Elias Konstantinidis. 2023. Analysis of the analytical performance models for GPUs and extracting the underlying Pipeline model. *J. Parallel and Distrib. Comput.* 173 (2023), 32–47. https://doi.org/10.1016/j.jpdc.2022.11.002

Yuzhi Liang, Qi Song, Rui Wang, Yuchi Huo, and Hujun Bao. 2023. Automatic Mesh and Shader Level of Detail. *IEEE Transactions on Visualization and Computer Graphics* 29, 10 (2023), 4284–4295. https://doi.org/10.1109/TVCG.2022.3188775

Charith Mendis, Alex Renda, Dr.Saman Amarasinghe, and Michael Carbin. 2019. Ithemal: Accurate, Portable and Fast Basic Block Throughput Estimation using Deep Neural Networks. In *Proceedings of the 36th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 97)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.). PMLR, 4505–4515. https://proceedings.mlr.press/v97/mendis19a.html

Changan Niu and Chuanyi Li. 2023. FAIR: Flow Type-Aware Pre-Training of Compiler Intermediate Representations. arXiv:2309.04828 [cs.SE]

Changan Niu, Chuanyi Li, Bin Luo, and Vincent Ng. 2022. Deep Learning Meets Software Engineering: A Survey on Pre-Trained Models of Source Code. In *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI-22*, Lud De Raedt (Ed.). International Joint Conferences on Artificial Intelligence Organization, 5546–5555. https://doi.org/10.24963/ijcai.2022/775 Survey Track.

NVIDIA. 2023. NVIDIA Nsight Graphics. https://developer.nvidia.com/nsight-graphics. Accessed: 2024-01-01.

Kenneth O'neal, Philip Brisk, Ahmed Abousamra, Zack Waters, and Emily Shriver. 2017. GPU Performance Estimation Using Software Rasterization and Machine Learning. *ACM Trans. Embed. Comput. Syst.* 16, 5s, Article 148 (sep 2017), 21 pages. https://doi.org/10.1145/3126557

Dinglan Peng, Shuxin Zheng, Yatao Li, Guolin Ke, Di He, and Tie-Yan Liu. 2021. How could Neural Networks understand Programs?. In *Proceedings of the 38th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 139)*, Marina Meila and Tong Zhang (Eds.). PMLR, 8476–8486. https://proceedings.mlr.press/v139/peng21b.html

Inigo Quilez and Pol Jeremias. 2013. Shadertoy BETA. https://www.shadertoy.com/. Accessed: 2024-01-01.

Andreas Sembrant, Trevor E. Carlson, Erik Hagersten, and David Black-Schaffer. 2017. A graphics tracing framework for exploring CPU+GPU memory systems. In *2017 IEEE International Symposium on Workload Characterization (IISWC)*. 54–65. https://doi.org/10.1109/IISWC.2017.8167756

Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Neural Machine Translation of Rare Words with Subword Units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Katrin Erk and Noah A. Smith (Eds.). Association for Computational Linguistics, Berlin, Germany, 1715–1725. https://doi.org/10.18653/v1/P16-1162

Pitchaya Sitthi-Amorn, Nicholas Modly, Westley Weimer, and Jason Lawrence. 2011. Genetic Programming for Shader Simplification. *ACM Trans. Graph.* 30, 6 (dec 2011), 1–12. https://doi.org/10.1145/2070781.2024186

O. Sykora, P. Phothilimthana, C. Mendis, and A. Yazdanbakhsh. 2022. GRANITE: A Graph Neural Network Model for Basic Block Throughput Estimation. In *2022 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE Computer Society, Los Alamitos, CA, USA, 14–26. https://doi.org/10.1109/IISWC55918.2022.00012

A. M. Turing. 1937. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society* s2-42, 1 (01 1937), 230–265. https://doi.org/10.1112/plms/s2-42.1.230 arXiv:https://academic.oup.com/plms/article-pdf/s2-42/1/230/4317544/s2-42-1-230.pdf

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems*, I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

Rui Wang, Xianjin Yang, Yazhen Yuan, Wei Chen, Kavita Bala, and Hujun Bao. 2014. Automatic Shader Simplification Using Surface Signal Approximation. *ACM Trans. Graph.* 33, 6, Article 226 (nov 2014), 11 pages. https://doi.org/10.1145/2661229.2661276

Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Commun. ACM* 52, 4 (apr 2009), 65–76. https://doi.org/10.1145/1498765.1498785

Y. Yang, C. Barnes, and A. Finkelstein. 2022. Learning from Shader Program Traces. *Computer Graphics Forum* 41, 2 (2022), 41–56. https://doi.org/10.1111/cgf.14457 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.14457

Yazhen Yuan, Rui Wang, Tianlei Hu, and Hujun Bao. 2018. Runtime Shader Simplification via Instant Search in Reduced Optimization Space. *Computer Graphics Forum* 37, 4 (2018), 143–154. https://doi.org/10.1111/cgf.13482 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.13482

Yi Zhai, Yu Zhang, Shuo Liu, Xiaomeng Chu, Jie Peng, Jianmin Ji, and Yanyong Zhang. 2023. TLP: A Deep Learning-Based Cost Model for Tensor Program Tuning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 833–845. https://doi.org/10.1145/3575693.3575737

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009

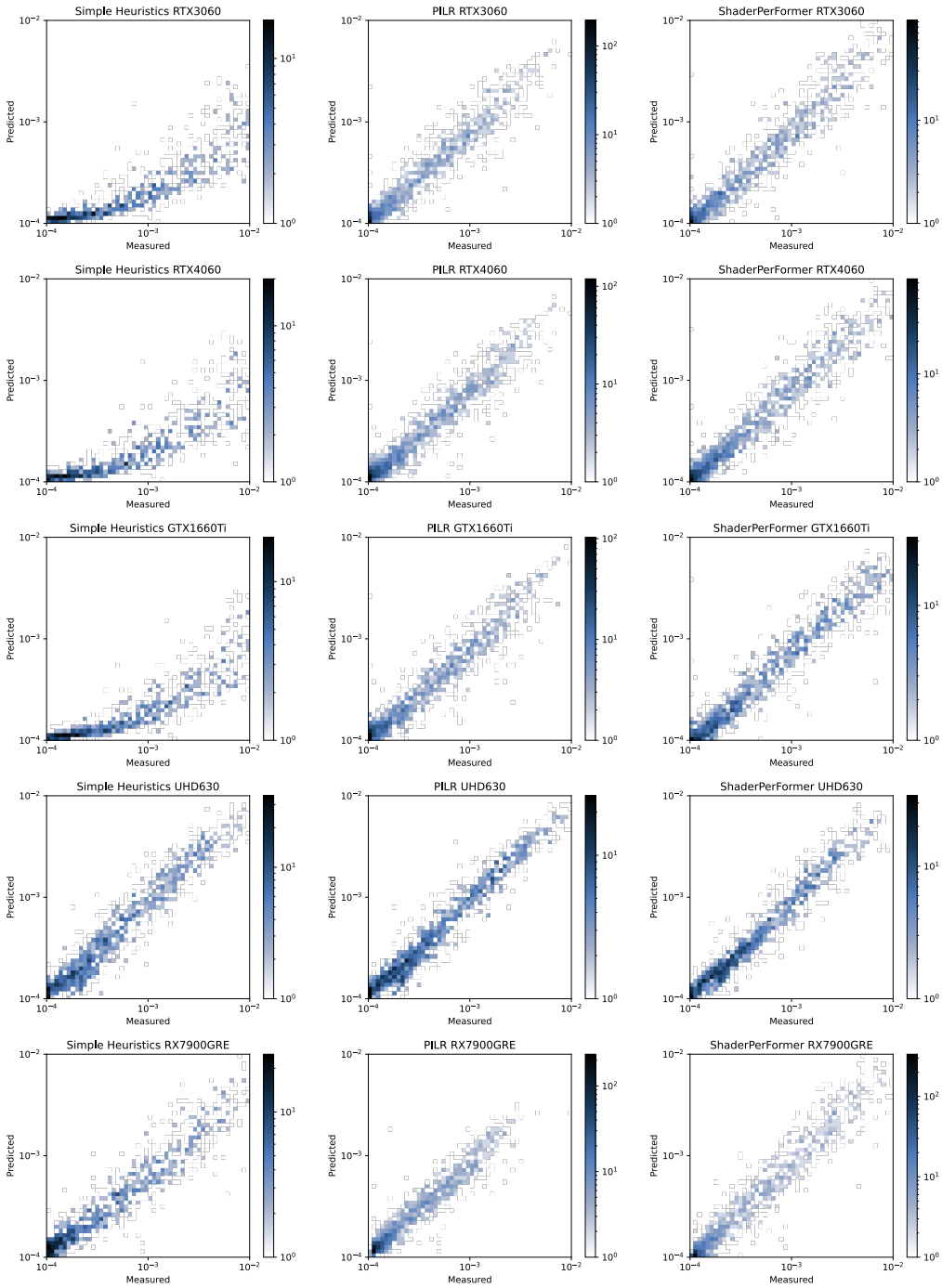


Fig. 7. Heatmap plot for predictions and measured results for baseline methods and ShaderPerFormer under all 5 GPU platforms. PILR represents Per Instruction Linear Regression.