

LLMPerf: GPU Performance Modeling meets Large Language Models

Minh-Khoi Nguyen-Nhat*

FPT Software AI Center

Hanoi, Vietnam

khoinnm1@fpt.com

Hoang Duy Nguyen Do

FPT Software AI Center

Hanoi, Vietnam

hoangdnd@fpt.com

Huyen Thao Le

FPT Software AI Center

Hanoi, Vietnam

huyenlt44@fpt.com

Thanh Tuan Dao

FPT University

Hanoi, Vietnam

tuandt28@fpt.edu.vn

Abstract—Performance modeling, a pivotal domain in program cost analysis, currently relies on manually crafted models constrained by various program and hardware limitations, especially in the intricate landscape of GPGPU. Meanwhile, Large Language Models (LLMs) have demonstrated their effectiveness in addressing diverse programming challenges. Our work establishes a connection between LLMs and performance modeling, employing the LLM as a performance estimator. Through experimental exploration with carefully designed large-scale OpenCL datasets, we highlight the potential capability as well as the main difficulties of using LLMs in handling performance modeling tasks for OpenCL device source programs. As the first study for this line of work, our LLM-based performance model achieves a mean absolute percentage error of 24.25% for a large-scale generated validation set. On a set of publicly available OpenCL programs, our model achieves a mean absolute percentage error of 46.1%.

Index Terms—GPU performance modeling, large language model, OpenCL, parallel computing

I. INTRODUCTION

Performance modeling involves designing mathematical models to serve as evaluators or simulators for estimating the performance of applications. This task requires intensive analysis not only of the program characteristics themselves but also of the underlying systems and hardware architectures. The term "performance" can be understood in various ways, including program execution time, program throughput, or other relevant performance metrics. A GPU performance modeling task is a specific type of performance modeling focused on GPUs, which is a massively parallel environment. In this paper, we address the problem of predicting program execution time, specifically targeting OpenCL kernels due to their broad applicability across various machines, making the method more accessible for diverse hardware. The ability to accurately predict execution time is valuable for a wide range of tasks aimed at enhancing program performance, such as workload distribution and program optimization.

GPU performance modeling is generally categorized into two major approaches: analytical models [1]–[3] and statistical models [4], [5]. CGPredict [3] addresses the problem by leveraging the execution trace and memory trace of a single-threaded program to develop an analytical model specifically designed for the NVIDIA Kepler Architecture. These analytical approaches often rely on handcrafted models developed by domain experts, resulting in models with low error rates

due to their careful design. However, these models require significant human effort and are prone to changes in system architecture [6]. To overcome these costly approaches, many statistical approaches developed. Overall, they aim to automate the model construction process by parameterizing the models and fitting their parameters to a training dataset collected from actual execution [4]. Since previous statistical methods rely on traditional machine learning techniques, they operate on a limited scale of data and are sensitive to feature choices. Additionally, many statistical models require runtime information, either in the form of single-threaded runtime data [7] or sample data points [8], [9], to achieve optimal performance. The reliance on dynamic features necessitates an available machine for kernel analysis and introduces runtime data collection overhead, thereby limiting the model's usability.

These limitations motivate the development of a statistical model that universally generalizes to any GPU program while considering only static information about the program and execution settings. In our setting, we fix the underlying system and only consider fundamental execution settings, including execution layout (global workgroup size, local workgroup size) and the kernel argument's value. Large Language Models (LLMs), recent advances in natural language processing, are well-suited for this purpose. LLMs, trained on extensive textual data, excel at understanding and generating human-like text and have demonstrated effectiveness in code-related tasks such as code auto-completion [10] and bug fixing [11]. By leveraging the code interpretability of LLMs and the awareness of interactions between code, hardware resources, and workload characteristics from a performance modeling perspective, our LLMPerf model aims to improve runtime predictions through more accurate reasoning about both code semantics and system-level performance factors.

Here, we formulate three questions that we think will be fundamental for future research:

- 1) How do we make new AI models (such as LLMs) useful for performance modeling?
- 2) How much can these models infer about the program without executing it?
- 3) How do we effectively represent the program and its execution environment so that the new AI models can learn them?

There are several key challenges to have accurate runtime estimation for arbitrary GPU source code. First, the massively parallel nature of the GPU programming architecture poses many difficulties in performance analysis. Second, the GPU architecture evolves quickly. This fact requires the performance models to attend to the low-level architectural changes and cover a wide range of performance factors. These performance factors are not always visible and require serious effort to analyze. Third, the GPU code has different performance behaviors with different inputs. It is not uncommon to see a library use different algorithms for the same problem with different input sizes [12]. Consequently, a dataset with high expressiveness is essential for model performance, requiring attention to not only program diversity but also input sizes and workgroup sizes. We address this by introducing a novel method for automatically generating a dataset that maintains expressiveness, leading to the creation of the first large-scale OpenCL performance dataset from real-life kernels. This dataset is then utilized to develop a pioneering LLM-based approach for estimating OpenCL execution times based solely on static information.

Our main contributions in this paper include:

- 1) A framework for automated collection of the execution time of an arbitrary one-dimensional GPU kernel at any input size.
- 2) A first large-scale OpenCL performance dataset from real life kernels
- 3) A first large language model that estimates OpenCL execution time from only static information.

Our paper is structured as follows: we describe our large-scale dataset creation framework for GPU kernel performance data in Section II. Section III introduces our proposed LLMPERF model that leverages LLMs for GPU kernel execution time prediction, followed by a comprehensive evaluation of LLMPERF’s accuracy, efficiency, and generalization capabilities in Section IV. Finally, Section V summarizes our key findings, limitations, and future research directions.

II. A LARGE-SCALE OPENCL KERNEL PERFORMANCE DATASET

Our dataset consists of two main components: a kernel source code corpus and the execution configuration for each kernel. The source code corpus, presented in section II-A, contains mainly kernel source code collected from various sources. The execution configuration, presented in Section II-B contains the necessary information to execute the kernel on a target device. This information includes the global work size, the local work size, the input to the kernel function, and the corresponding host code of the kernel.

A. Source Code Corpus

We adapt an OpenCL kernel corpus from BenchPress [13] with slight modifications in pre-processing: stripping double underscore prefixes and formatting using clang-format. The corpus comprises over 15000 OpenCL kernel source codes crawled from GitHub datasets.

The number of dimensions in a kernel’s execution space is an important performance factor. Table I shows the number of kernels belonging to each dimensional category in our corpus. Clearly, our corpus contains 10830 one-dimensional kernels (70%), accounting for the majority. This observation aligns with the intuition that most real-world problems are inherently one-dimensional, and higher-dimensional problems can be effectively reduced to a 1D representation. Consequently, we constrain our models to take 1D kernels as input, simplifying the learning scope while still capturing most real-life scenarios.

TABLE I: Dimensionality Distribution on all kernels in corpus

| Dimension | Count | Percentage (%) |
|-----------|-------|----------------|
| 1D | 10830 | 69.86 |
| 2D | 3879 | 25.02 |
| 3D | 793 | 5.12 |

B. Kernel Execution Configuration

This section describes how we generate the kernel inputs and the kernel execution configurations so that the kernel source code can be compiled and executed in an automatic manner.

In reality, the performance is affected by the OpenCL program, the compiler, the operating system, and the host/device architectures. Currently, we only consider the OpenCL programs and keep other factors fixed. Therefore, our models assume the performance variation only stems from the OpenCL programs. Henceforth, we refer to the combination of these three elements (*kernel source code*, *kernel inputs*, and *the kernel execution configuration*) as a **launch configuration**. This section describes our methods to generate the launch configurations in our dataset.

1) *Simple input argument selection*: The kernel source lacks input argument information (sizes and values) passed from the host code. We generate this information, selecting input arguments based on the global work size `gsize`:

- 1) If an argument is an array (e.g., `__global float4*`), create an array of `gsize` random values.
- 2) If an argument is a scalar (e.g., `int2`), create a scalar equal to `gsize`.

We adopt CLDrive [14] to implement this simple approach, generating valid input for 10250 out of 15502 kernels, assuming similar performance to the original input. It is important to clarify that a valid input in this context guarantees that the kernel can execute and produce its output without any compilation or runtime errors. Our underlying assumption is that the performance of the kernel with the generated input is similar to that of the kernel with the original input.

2) *Memory-analysis based input argument selection*: The simple approach can generate a large number of launch configurations. However, there are two main limitations. First, input array sizes and scalars are not always equal to the global work size. Second, scalar inputs can have varying values and affect performance, such as when they are part of branch divergence conditions.

To address these limitations and generate kernels with wider performance characteristics, we propose an enhanced input argument selection strategy. Our approach analyzes the kernel source code to identify global memory access patterns and generate more diverse and realistic combinations of input array sizes, global work sizes, and local work sizes. From analyzing 100 random kernels, we observe that most input array sizes can be inferred by statically analyzing memory access patterns, as GPU programs tend to have regular memory accesses to exploit parallelism. We identify four common patterns: data partitioning, offset and stride, boundary check, and complex forms combining the previous patterns.

- **Data partitioning:** When the input size is large, it is common to partition the data into multiple chunks and assign each chunk to a work-item. The number of elements being processed by a work-item is often expressed by some constant k . A typical example of this pattern is the summation program 1, where a scalar k specifies the number of elements on which each work-item should perform the summation operation.

```
kernel void summation(__global float* in,
                      __global float* out, int N, int k) {
    int id = get_global_id(0);
    float sum = 0.0f;
    for (int i = 0; i < k; i++) {
        int cId = id * k + i;
        if (cId < N) {
            sum += in[cId];
        }
    }
    out[id] = sum;
}
```

Listing 1: Data partitioning

- **Offset and stride:** The memory indexes accessed in an array might not be consecutive, but rather interleaved with respect to the global ID, workgroup ID or local ID. For the most cases, the accessing index can be expressed in the form of two variables: offset and stride. A common example of this pattern is the convolution program 2 that perform the convolution on a filter and a 1D-array. In this case, the offset is os and the stride is st .

```
kernel void convolution(__global float* in,
                       __global float* out, int os, int st) {
    int id = get_global_id(0)*st+os;
    out[id] = in[id - 1] * 0.5f
              + in[id] * 1.0f
              + in[id + 1] * 0.5f;
}
```

Listing 2: Offset and stride

- **Boundary check:** It is a common practice to have a boundary check in the kernel to guarantee no violated memory access. In this context, the input scalar value represents the size of the array that must be checked, preventing out-of-bounds errors and ensuring the integrity of the program's execution. Program 3 shows a simple case where the kernel applies a boundary check for the array a of size N .

```
kernel void initZero(__global float* a, int N)
{
    int id = get_global_id(0);
    if (id < N) {
        a[id] = 0.0f;
    }
}
```

Listing 3: Boundary check

- **Complex form:** The last pattern is a combination of the three previous patterns.

A common characteristic of the patterns is that each array index (hence the array size) is an affine function in the global ID, workgroup ID and local ID. We leverage this insight to propose an approach of fixing scalar values and finding a linear relationship between array size and global size. This allows decoupling the correlation between input size and global size, leading to more diverse input values for kernels. Specifically, given a global size $gsize$ for kernel K with scalar input argument list $S = \{s_1, \dots, s_m\}$, where s_i is the i -th scalar input, and $A = \{a_1, \dots, a_n\}$, where a_i is the i -th array input, we need to find:

$$size_{a_i} = c_{a_i} \cdot gsize + d_{a_i} \quad \forall a_i \in A \quad (1)$$

considering fixed scalar values $\{value_{s_1}, \dots, value_{s_m}\}$.

To find (c_{a_i}, d_{a_i}) , we sample two data points $(gsize^{(1)}, size_{a_i}^{(1)})$ and $(gsize^{(2)}, size_{a_i}^{(2)})$, which corresponds to the sizes of a_i when running kernel K with two sample global sizes $gsize^{(1)}$ and $gsize^{(2)}$. To obtain these points, we first use Clang AST to instrument each memory access and record the memory access locations. We then collect the memory access locations using two global work sizes $gsize^{(1)}$ and $gsize^{(2)}$. We set the array sizes based on the smallest and largest recorded memory access locations. Obtained (c_{a_i}, d_{a_i}) , we substitute $gsize$ into $size_{a_i} = ceil(c_{a_i} \cdot gsize + d_{a_i})$ to find the target array size $size_{a_i}$, where ceiling function $ceil$ takes a decimal number and rounds up it if its fractional part is greater than 0.

We choose input scalar values from a set of practically predefined values based on the observed patterns. For instance, the boundary checks often set the scalars at a multiple of $gsize$; offsets and strides often use a small constant such as 1, 2, 256, etc.. Therefore, we define a set of potential candidate values $cand_i$ for each s_i , and try all possible combinations in $cand_1 \times \dots \times cand_m$. In our experiments, we set $cand_i = \{1, 4, gsize, 16, 32, 256\}$ for all $s_i \in S$. The algorithm 1 details this process. On average, a kernel has 2 input scalar arguments. We find that this approach has a reasonable cost of running time.

3) *Kernel Execution Settings Selection Strategy:* In this part, our goal is to find appropriate global work sizes and local work sizes so that the performance variation is most beneficial for the AI models to generalize the performance.

For the local work size (denoted as $lsize$), since a multiple of warp size to ensures that work-items in a work-group are distributed evenly into the warps. We enforce

Algorithm 1 OpenCL Kernel Input Arguments Selection Algorithm

Require: Kernel K, Global size gsize

```

1: cands  $\leftarrow \{1, 4, gsize, 16, 32, 256\}$ 
2: scalars  $\leftarrow \text{GetKernelScalarList}(K)$ 
3: arrays  $\leftarrow \text{GetKernelArrayList}(K)$ 
4:  $K' \leftarrow \text{InsertArrayHook}(K)$ 
5: combs  $\leftarrow \text{cands}^{|\text{scalars}|}$ 
6: validSettings  $\leftarrow \{\}$ 
7: exeSettings  $\leftarrow \{(gsize1, lsize1), (gsize2, lsize2)\}$ 
8: for all comb  $\in$  combs do
9:   P1  $\leftarrow \text{RunKernel}(K', \text{combs}, \text{exeSettings}[0])$ 
10:  P2  $\leftarrow \text{RunKernel}(K', \text{combs}, \text{exeSettings}[1])$ 
11:  arraySizes  $\leftarrow [0 \times |\text{arrays}|]$ 
12:  for all b  $\in B$  do
13:    slope, intercept  $\leftarrow \text{LinearInterpolation}(P1[b],$ 
      P2[b])
14:    arraySizes[b]  $\leftarrow \text{slope} \times gsize + \text{intercept}$ 
15:  end for
16:  if all(size > 0 for size in arraySizes) then
17:    validCombs.append({comb, arraySizes})
18:  end if
19: end for
20: return validCombs

```

this constraint and randomize $lsize$ within the range of $[warp_size, MAX_LOCAL_SIZE]$.

For each $lsize$, we vary $gsize$ to get the set of desired execution settings. Given a $lsize$ and considering all possible values of $gsize$, the corresponding performance of a GPU kernel can generally be categorized into three types: **idle SMs**, **under-utilized**, and **fully utilized** sections [15], [16]. Such categorization mainly stems from the ability of latency hiding in GPUs [15]. Note that we can directly infer the $gsize$ value from the number of workgroups (denoted as N_{WG}) and $lsize$ using the following equation:

$$N_{WG} = \frac{gsize}{lsize} \quad (2)$$

Furthermore, since running the kernels at the values of $gsize$ that are divisible by $lsize$ is more reasonable in reality, we use N_{WG} to control the kernel parallelism for a given $lsize$ and derive $gsize$ from Equation 2.

Let N_{SM} denote the number of Streaming Multiprocessors (SMs) in the GPU. When $N_{WG} < N_{SM}$, not all SMs are active, leading to idle SMs during execution. The kernel execution time is typically the execution time of a single workgroup plus the overhead of workgroup launching. When N_{WG} is very large (e.g., $N_{WG} > 40 \times N_{SM}$), assuming round-robin SM scheduling, all SMs are busy during execution, and latency hiding is fully achieved in each SM. In this case, the kernel execution time is typically proportional to N_{WG} . When $N_{WG} > N_{SM}$ but is not sufficiently large to achieve fully utilized performance, the kernel is under-utilized. In this stage, all SMs are active, but the workload per SM is insufficient to

hide all latencies. The GPU throughput increases as N_{WG} grows. Among the three types, under-utilized execution configurations contain the most performance information due to varying parallelism per SM and characterizing the OpenCL workgroup scheduler. Therefore, we give the highest priority to sampling N_{WG} for this type. Idle SMs generally contain more performance variation than fully-utilized, as fully utilized execution times linearly increase with workload. Based on these considerations, we use an empirical sampling ratio of 25 : 60 : 15 for idle SMs, under-utilized, and fully-utilized types, respectively. To detect the fully utilized point, we use a common constant $k \times N_{SM}$ to avoid significant profiling overhead.

Handling data imbalance.

There are two main limitations with the aforementioned analysis: it does not consider the kernel execution time and the exact boundaries of the three performance types significantly depend on the kernel characteristics. Consequently, the generated data suffer from data imbalance with regard to the execution time distribution of the generated launch configurations. This is a non-trivial hurdle for the AI models to learn the performance from the samples.

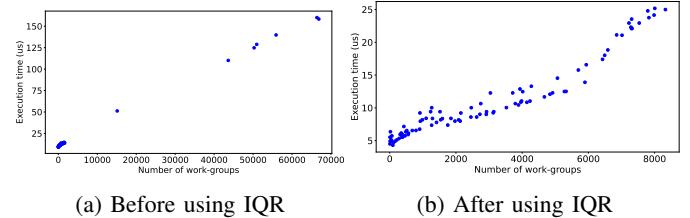


Fig. 1: Example of execution time data before and after using IQR

Figure 1a shows the execution time distribution of kernel *division_assignment_scalar* in our corpus. The graph shows the execution times with $lsize = 128$ and varying $gsize$. The majority of the data lies on the bottom-left corner of the graph, i.e., with very small execution times, while only a few data points that have the medium and long execution times.

To handle the imbalance problem, we propose a refinement algorithm to generate more balanced data points with regard to the execution time. The idea is based on an outlier detection method called the Interquartile Range(IQR) [17]. The algorithm assumes the data has a normal distribution. It estimates the central of the distribution based on frequent data points (small execution times) and the two edges of the distribution based on the infrequent data points (medium and long execution times). The algorithm then estimates the range of execution times missing from the distribution. We fit a linear regression model on the data generated and use this regression model to derive the corresponding N_{WG} to sample from the missing execution times. We can observe the improvement of the execution time distribution in Figure 1b.

III. LLMPERF MODEL

This section provides information of our model architecture, input representation, and training process that we use to train the model for execution time estimation. We emphasize that our model does not use any runtime information. It functions without any requirement of the OpenCL or GPU execution environment. In the inference phase, the only input it requires is the source code of the OpenCL kernel.

A. Prompt Engineering

Our model is a Large Language Model (LLM), thus we need to represent the input as a prompt, which is a text structure that the model will take as input. Designing an effective prompt is crucial, as the model learns better with a well-structured prompt. We leverage the prompt technique from the popular instruction-tuning dataset Alpaca [18] with modifications to better fit our use case. Specifically, for one data sample, the prompt is formatted as follows:

```
Predict time for the following OpenCL kernel:  
{code}  
Given input:  
{inputs}  
Given kernel are executed with global size equals cl  
    ::NDRange({global_size}), local work-group size  
    equals cl::NDRange({local_size})
```

In the prompt above, `code` section contains the kernel source code. The `input` section contains the input argument settings generated by concatenating the description of each input argument in the kernel on a separate line. We have separate description templates for array and scalar input arguments:

```
Argument at position {id} is {name}, which is {  
    qualifier} buffer of type {type} with size {  
    value}
```

```
Argument at position {id} is {name}, which is {  
    qualifier} scalar of type {type} with value {  
    value}
```

Here, `id`, `name`, `qualifier`, `type`, and `value` provide the argument index, argument name, argument qualifier (e.g., `global`, `local`, `const`, and `private`), argument data type (e.g., `int`, `float`), and argument size (if the argument is an array) or argument value (if the argument is a scalar), respectively.

This prompt format captures the essential static information needed for our model to predict execution time, with a focus on array sizes rather than element values. This choice is made for three key reasons: (1) OpenCL programs typically process large datasets in a SIMD manner, where the amount of data is the dominant performance factor; (2) while array element values can impact performance, especially in cases of memory or execution divergences, such scenarios are generally avoided by OpenCL programmers; and (3) including entire array values would make the prompt excessively long, risking context length limits in LLMs and complicating the optimization process.

B. Model architecture

Large Language Models (LLMs) are pre-trained on vast textual data using self-supervised learning objectives and often require further fine-tuning on task-specific datasets to adapt their knowledge. We choose CodeGen [19] as the pre-trained model and consider execution time estimation a fine-tuning task. CodeGen is a powerful LLM family developed by Salesforce Research for code-related tasks, trained on a vast corpus of source code and natural language data.

To better fit the regression task, we remove the language prediction head from CodeGen and introduce a new regression head to allow direct execution time prediction for a given kernel. This regression head takes N vectors of dimension d , concatenates all N vectors into one feature vector of dimension Nd , and projects the feature vector to the target execution time using a Multi-Perceptron (MLP) layer. We call this model LLMPERf. LLMPERf is trained on the generated prompt dataset to predict the binary logarithm of execution time using Mean Squared Error (MSE) loss. We chose to predict the logarithm of execution time to reduce sensitivity to large execution times and stabilize the training process.

IV. EVALUATION

TABLE II: Datasets information with number of samples after preprocessed inside brackets

| | 200K | 230K | 400K |
|--------------|-----------------|-----------------|-----------------|
| Train | 177892 (171790) | 206369 (198689) | 348970 (338434) |
| Val | 19922 (19655) | 23500 (23233) | 39975 (38797) |
| Total | 197814 (191445) | 229869 (221922) | 388945 (377231) |

In this section, we provide an evaluation of our LLMPERf model. We describe our performance datasets in Section IV-A. Section IV-B details the experimental setup and results on our large-scale dataset.

Section IV-C assesses the model on well-established OpenCL benchmarks, showcasing its ability to handle a wide range of kernels beyond the training data.

A. OpenCL performance dataset

In order to demonstrate the effectiveness of our data generation techniques, we create three versions of datasets. We use NVIDIA Tesla V100S PCIe 32 GB ($N_{SM} = 80$) to generate the performance data. We use the OpenCL C++ interface [14], [20] to create host programs and run kernels with OpenCL 2.2.

TABLE III: Experiment of LLMPERf on three large-scale datasets

| Model | Train MAPE | Val MAPE | Val MAPE (400K) |
|-------------------|------------|----------|-----------------|
| LLMPERf-350M-200K | 23.29 | 42.68 | 80.53 |
| LLMPERf-350M-230K | 21.78 | 37.39 | 51.48 |
| LLMPERf-350M-400K | 1.86 | 43.65 | 43.77 |
| LLMPERf-2B-200K | 6.55 | 34.2 | 64.9 |
| LLMPERf-2B-230K | 7.09 | 23.61 | 54.44 |
| LLMPERf-2B-400K | 3.92 | 24.25 | 24.25 |

200K. This dataset contains 200K data points generated using the simple version of the input argument selection strategy. We set $k = 80$. For each kernel source code, we generate 200 launch configurations (4 random $lsize$ and 50 random $gsize$). We generate one input set for each of 200 launch configurations.

230K. We employ the memory analysis based input argument selection strategy to generate 30K additional data points.

400K. Lastly, we apply IQR to the kernel execution setting selection of the 230K dataset to create a total of 400K data points with a more balanced data distribution.

B. Experiments on large-scale datasets

1) *Experiment settings: Dataset Setting:* LLMPef is trained on all three datasets. We split each dataset into training and validation sets with a ratio of 9 : 1. We ensure that no kernel appears in both the training and validation sets. Any prompts exceeding the model’s context length are discarded. For the target values, execution times are converted to microseconds. Our datasets are described in Table II.

Model Setting: We use the Multi variant of the CodeGen model, which is pre-trained on a combination of natural language and code corpora. To study the effect of model capacity, we use two model sizes: 350M (LLMPef-350M) and 2B (LLMPef-2B) parameters. The maximum number of tokens (maximum context length) that the model can process is 2048.

Training Setting: The AdamW optimizer [21] is used with a linear learning rate scheduler, 1000 warmup steps, a learning rate of 10^{-6} , $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$, and a weight decay rate of 0.01. Early stopping [22] is employed to terminate the training process if the loss does not decrease after 15 epochs.

Evaluation metric: We use Mean Absolute Percentage Error (MAPE) to evaluate the performance of LLMPef in predicting one particular kernel execution time. MAPE measures the average absolute percentage deviation between the predicted and true (measured) execution times. It is calculated as:

$$\text{MAPE} = \frac{100\%}{N} \sum_{i=1}^N \left| \frac{\hat{y}_i - y_i}{y_i} \right| \quad (3)$$

where N is the total number of samples, \hat{y}_i is the predicted execution time, and y_i is the true execution time for sample i .

2) *Experiments result:* Table III illustrates our results of LLMPef-350M and LLMPef-2B on 200K, 230K, and 400K datasets. After training, we add an additional evaluation of each model on the validation set of the 400K dataset to consistently evaluate the models’ quality on the most general and unseen dataset among the three validation sets. LLMPef-2B-400K demonstrates superior performance on the 400K validation set, achieving a MAPE of 24.25%, indicating strong generalization and accuracy when trained on the highest quality dataset. In comparison, LLMPef-2B-200K and LLMPef-2B-250K exhibit higher MAPEs of 34.2% and 23.61%, respectively, though they still outperform their corresponding

LLMPef-350M models in both training and validation sets. This suggests that the LLMPef-2B models have a greater capacity for learning and generalizing compared to the LLMPef-350M models.

LLMPef-350M consistently shows higher validation MAPEs, with 80.53% for LLMPef-350M-200K, 51.48% for LLMPef-350M-250K, and 43.77% for LLMPef-350M-400K, primarily due to its smaller size and limited learning capacity. LLMPef-350M-400K, despite a low training MAPE of 1.86%, suffers from significant overfitting with a validation MAPE of 43.65%. The dataset generation techniques also impact model performance: LLMPef-2B-200K, which had the highest validation MAPE of 64.9%, indicates a lack of diversity in performance data, while LLMPef-2B-250K’s lower MAPE of 23.61% benefits from improved input argument generation. LLMPef-2B-400K achieves the best result by effectively utilizing a comprehensive dataset, thus overcoming previous limitations and achieving a validation MAPE of 24.25%.

C. Experiments on public OpenCL benchmark

Due to the generation technique, in our training dataset, most of our samples employ the input size - global size correlation, where input size equals to global size, only a small amount of them release this correlation. Therefore, we want to assert the model’s performance with diverse performance patterns to see its generability. Here, we use the SHOC [23] and Rodinia [24] benchmark suite to evaluate our best model (LLMPef-2B-400K). Note that we do not use these benchmarks in our training process.

Usually, the program input size options for each kernel provided by its benchmark are limited, so we vary the input sizes with the same techniques as those used in original benchmarks to further observe the trend of execution time. For each input size, we randomly create data, measure the kernel execution time N times (usually $N = 100$) and take the average as target execution time. We select kernels in these two benchmarks that satisfy our assumptions (are 1D kernels and do not exceed the context length of the model) and record their execution times. We use LLMPef-2B-400K

TABLE IV: MAPE of LLMPef-2B-400K on Real Benchmarks

| Benchmark-Program (Kernel) | MAPE (%) |
|------------------------------------|--------------|
| SHOC-BFS (BFS_kernel_warp) | 53.62 |
| SHOC-MD (compute_lj_force) | 34.02 |
| SHOC-Reduction (reduce) | 62.44 |
| SHOC-Scan (bottom_scan) | 63.0 |
| SHOC-Scan (reduce) | 65.85 |
| SHOC-SPMV (spmv_csr_scalar_kernel) | 56.76 |
| SHOC-SPMV (spmv_csr_vector_kernel) | 32.51 |
| SHOC-SPMV (spmv_ellpackr_kernel) | 42.76 |
| SHOC-Triad (triad) | 42.8 |
| Rodinia-BFS (BFS_1) | 35.43 |
| Rodinia-Kmean (kmeans_kernel_c) | 43.21 |
| Rodinia-NN (NearestNeighbor) | 20.93 |
| Average | 46.11 |

model to evaluate all 12 satisfied kernels. The detailed MAPEs

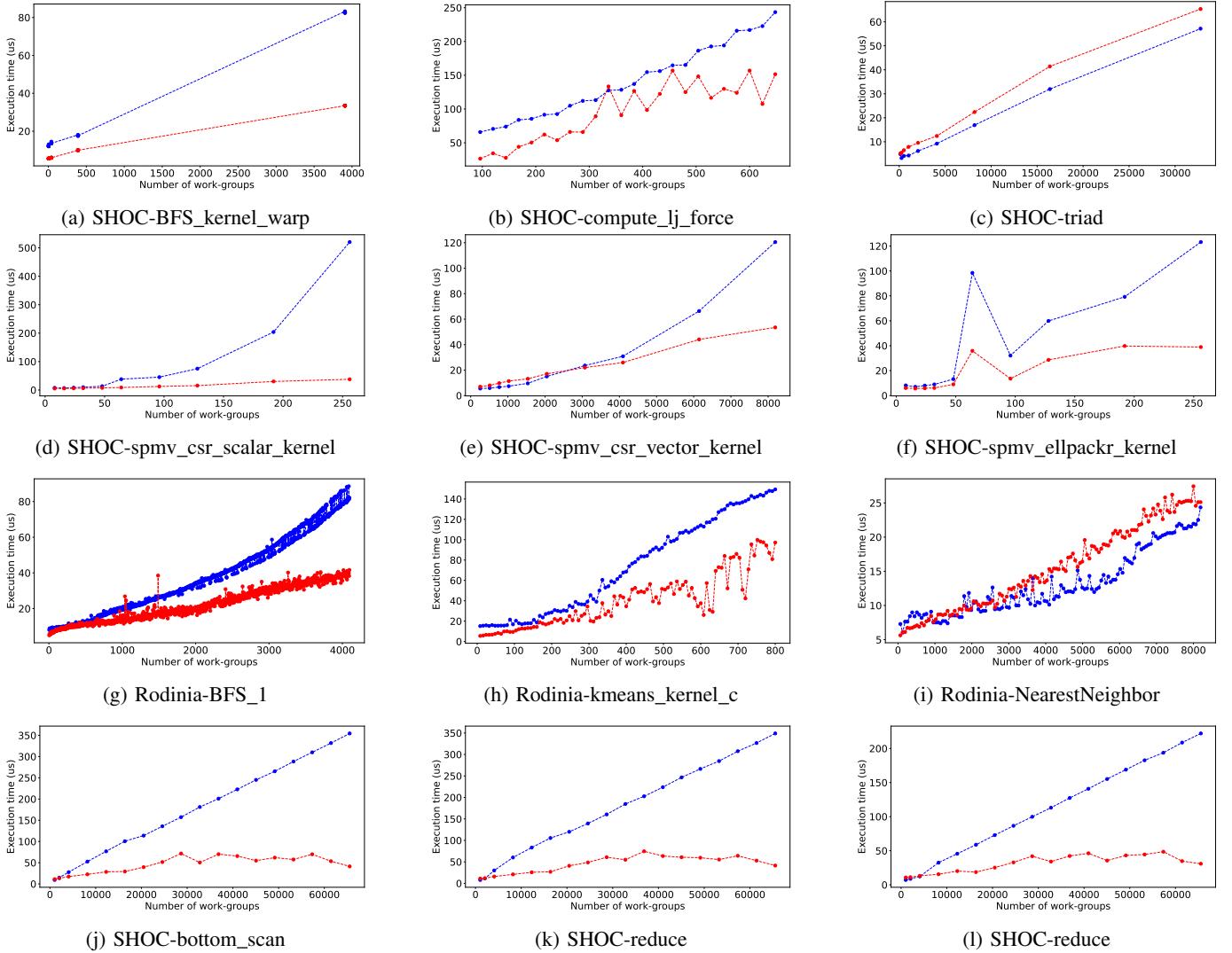


Fig. 2: Result of LLMPERF on real benchmark kernels by data (input) size. Red points and blue points are prediction time and its corresponding target time.

for each kernel are showed in Table IV. With an average MAPE of 46.11%, the model demonstrates reasonably good overall accuracy in predicting kernel execution times only using the kernel source codes. The model excels at kernels like NN (20.93%) and MD (34.02%). However, it struggles with kernels such as two reduce kernels (65.85% and 62.44%), bottom_scan (63.0%), bfs kernel (52.62%) and spmv_csr_scalar_kernel (53.62%).

While MAPE provides a quantitative measure, visualizing predicted and target execution times offers deeper insights into the model’s performance. Figures 1-9, except for spmv_csr_scalar_kernel, roughly predict the relative magnitude between data sizes, albeit slightly lower or higher than the target. For kernels with simple performance patterns (no loops or branching) like triad or NearestNeighbor, the model performs exceptionally well. Notably, LLMPERF still performs reasonably on kernels with

complex patterns, such as loops in BFS_kernel_warp, compute_lj_force, spmv_ellpackr_kernel, kmeans_kernel_c, and barriers (thread synchronization) in spmv_csr_vector_kernel. Even for kernels such as spmv_ellpackr_kernel, spmv_csr_vector_kernel, and BFS_1, where performance depends on array element values (violating our assumption and excluding this information from the prompt), the model demonstrates strong generalization despite limited information provided. This indicates that LLMPERF is robust in understanding kernel performance when sufficient data is available (such as correlations between input size and global size), effectively handling a range of patterns from simple to complex, even with partial information.

However, in cases where a kernel does not have the input size-global size correlation, such as spmv_csr_scalar_kernel, it cannot

capture the performance pattern despite sharing similarities with `spmv_ellpackr_kernel` and `spmv_csr_vector_kernel`. Additionally, when only data size changes, while global size remains fixed, as in Scan-bottom_scan (Figure 2j), Scan-reduce (Figure 2k), and Reduction-reduce (Figure 2l), the model lags in capturing the relationship between input size and performance. This limitation likely arises from the relatively small number of varied input size samples ($25K$) compared to the cases where global size and input size are equal ($365K$) in the training set. According to the result with input size-global size correlation kernels, we believe increasing the varied input size samples using memory analysis-based input argument generation could potentially overcome this limitation, which we leave for future work.

V. CONCLUSION

In this paper, we introduce the first LLM model that can predict the performance of OpenCL kernels solely based on their source codes. We also introduce the first large-scale OpenCL performance dataset that we will release publicly. We introduce techniques using compiler-based memory analysis and statistical analysis to enrich the performance dataset. We then introduce LLMPERF and evaluate our proposed model on various scenarios. Although the results that we achieve are humble, we believe a large scale exploration will lead to much better findings, which will be conducted in our future research. We believe this is the first step to construct a meaningful relationship between NLP models and the system performance.

ACKNOWLEDGEMENT

We would like to express our sincere gratitude to Moreh Company for providing the computing resources essential for this project.

REFERENCES

- [1] S. Hong and H. Kim, “An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness,” in *Proceedings of the 36th annual international symposium on Computer architecture*, 2009, pp. 152–163.
- [2] A. Li, Y. Tay, A. Kumar, and H. Corporaal, “Transit: A visual analytical model for multithreaded machines,” in *Proceedings of the 24th international symposium on high-performance parallel and distributed computing*, 2015, pp. 101–106.
- [3] S. Wang, G. Zhong, and T. Mitra, “Cgpredict: Embedded gpu performance estimation from single-threaded applications,” *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 16, no. 5s, pp. 1–22, 2017.
- [4] Y. Zhang, Y. Hu, B. Li, and L. Peng, “Performance and power analysis of ati gpu: A statistical approach,” in *2011 IEEE Sixth International Conference on Networking, Architecture, and Storage*. IEEE, 2011, pp. 149–158.
- [5] D. Moolchandani, A. Kumar, and S. R. Sarangi, “Performance and power prediction for concurrent execution on gpus,” *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 19, no. 3, pp. 1–27, 2022.
- [6] S. Madougou, A. Varbanescu, C. de Laat, and R. van Nieuwpoort, “The landscape of gpgpu performance modeling tools,” *Parallel Computing*, vol. 56, pp. 18–33, 2016.
- [7] A. Karami, S. A. Mirsoleimani, and F. Khunjush, “A statistical performance prediction model for opencl kernels on nvidia gpus,” in *The 17th CSI International Symposium on Computer Architecture & Digital Systems (CADS 2013)*. IEEE, 2013, pp. 15–22.
- [8] L. Braun, S. Nikas, C. Song, V. Heuveline, and H. Fröning, “A simple model for portable and fast prediction of execution time and power consumption of gpu kernels,” *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 18, no. 1, pp. 1–25, 2020.
- [9] Y. Arafa, A.-H. Badawy, A. ElWazir, A. Barai, A. Eker, G. Chennupati, N. Santhi, and S. Eidenbenz, “Hybrid, scalable, trace-driven performance modeling of gpgpus,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–15.
- [10] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, and G. Lample, “Llama: Open and efficient foundation language models,” 2023.
- [11] X. Chen, M. Lin, N. Schärlí, and D. Zhou, “Teaching large language models to self-debug,” *arXiv preprint arXiv:2304.05128*, 2023.
- [12] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Z. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, “Pytorch: An imperative style, high-performance deep learning library,” *CoRR*, vol. abs/1912.01703, 2019. [Online]. Available: <http://arxiv.org/abs/1912.01703>
- [13] F. Tsimpourlas, P. Petoumenos, M. Xu, C. Cummins, K. Hazelwood, A. Rajan, and H. Leather, “Benchpress: A deep active benchmark generator,” in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2022, pp. 505–516.
- [14] C. Cummins, P. Petoumenos, Z. Wang, and H. Leather, “Synthesizing benchmarks for predictive modeling,” in *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2017, pp. 86–99.
- [15] V. Volkov, *Understanding latency hiding on GPUs*. University of California, Berkeley, 2016.
- [16] T. T. Dao, J. Kim, S. Seo, B. Egger, and J. Lee, “A performance model for gpus with caches,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 7, pp. 1800–1813, 2014.
- [17] H. Vinutha, B. Poornima, and B. Sagar, “Detection of outliers using interquartile range technique from intrusion dataset,” in *Information and decision sciences: Proceedings of the 6th international conference on ficta*. Springer, 2018, pp. 511–518.
- [18] R. Taori, I. Gulrajani, T. Zhang, Y. Dubois, X. Li, C. Guestrin, P. Liang, and T. B. Hashimoto, “Stanford alpaca: An instruction-following llama model,” 2023. [Online]. Available: https://github.com/tatsu-lab/stanford_alpaca
- [19] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, “Codegen: An open large language model for code with multi-turn program synthesis,” *ICLR*, 2023.
- [20] “Opencl c++ bindings,” <https://github.khronos.org/OpenCL-CLHPP/>, accessed: April 2, 2024.
- [21] I. Loshchilov and F. Hutter, “Decoupled weight decay regularization,” *arXiv preprint arXiv:1711.05101*, 2017.
- [22] N. Morgan and H. Bourlard, “Generalization and parameter estimation in feedforward nets: Some experiments,” *Advances in neural information processing systems*, vol. 2, 1989.
- [23] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tippuraju, and J. S. Vetter, “The scalable heterogeneous computing (shoc) benchmark suite,” in *Proceedings of the 3rd workshop on general-purpose computation on graphics processing units*, 2010, pp. 63–74.
- [24] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, “Rodinia: A benchmark suite for heterogeneous computing,” in *2009 IEEE international symposium on workload characterization (IISWC)*. Ieee, 2009, pp. 44–54.