Graph Mining and Machine Learning for Shader Codes Analysis to Accelerate GPU Tuning

Lin Zhao[1], Arijit Khan[2], Robby Luo[3], and Chai Kiat Yeo[4]

[1] Nanyang Technological University, Singapore,
lin018@e.ntu.edu.sg
[2] Aalborg University, Denmark,
arijitk@cs.aau.dk
[3] Advanced Micro Devices Co., Ltd.,
robby.luo@amd.com
[4] Nanyang Technological University, Singapore,
asckyeo@ntu.edu.sg

Abstract. The graphics processing unit (GPU) has become one of the most important computing technologies. Disassembly shader codes, which are machine-level codes, are important for GPU designers (e.g., AMD, Intel, NVIDIA) to tune the hardware, including customization of clock speeds and voltages. Due to many use-cases of modern GPUs, engineers generally find it difficult to manually inspect a large number of shader codes emerging from these applications. To this end, we develop a framework that converts shader codes into graphs, and employs sophisticated graph mining and machine learning techniques over a number of applications to simplify shader graphs analysis in an effective and explainable manner, aiming at accelerating the whole debugging process and improving the overall hardware performance. We study shader codes' evolution via temporal graph analysis and structure mining with frequent subgraphs. Using them as the underlying tools, we conduct a frame's scene detection and representative frames selection. We group the scenes (applications) to identify the representative scenes, and predict a new application's inefficient shaders. We empirically demonstrate the effectiveness of our solution and discuss future directions.

1 Introduction

The graphics processing unit (GPU) was formerly a specialized microprocessor for offloading graphically intensive tasks, such as online video games that adopt real-time 3D graphics, special effects including virtual reality. Recently, GPUs are involved in many general-purpose, programmable, and multi-core applications [1, 2]. One of the most exciting applications of GPUs consists of deep learning and machine learning. GPUs can process tons of data and train deep neural networks for image and video analytics, speech recognition, natural language processing, and self-driving cars. In high-performance computing (HPC), scientific workflows, and Internet-of-Things (IoT) domains, modern GPUs are widely adopted to speed-up a variety of computing applications, e.g., gene mapping, virus tracing. The computer-aided design (CAD) software utilizes GPUs to visualize 3D objects in real time.

1.1 Challenges in Conventional GPU Tuning

From GPU designers' (e.g., AMD, Intel, NVIDIA) perspective, designing high-performance, low-power, and small-area chips is crucial for their products to seize market shares. Based on feedback from our industry collaborators, they experience three major challenges. **First**, since GPU designers are not the same as application designers (e.g., game developers), it is difficult for them to obtain the application source code and adjust their products accordingly. **Second**, as the field of GPU applications expands, a vast number of different programs are executed on the GPUs. However, engineers often tune GPUs (e.g., customization of clock speeds and voltages, VRAM tuning, memory timing, fan tuning) based on a few benchmark applications for hand-crafted key portions according to past experiences. This manual tweaking may result in missing critical information and sub-optimal performance. GPU tuning also requires an advanced level of technical knowledge. Incorrect tuning can make the system unstable, e.g., resulting in crashes, hangs, and/or graphical corruption [3]. **Third**, since new games and applications are emerging continuously, it is tedious to design a new tuning method for each new application.

1.2 Our Solution and Motivation

To solve the aforementioned challenges, we build a unified graph model that aims to accelerate the GPU performance tuning leveraging graph mining and machine learning tools. Our model analyzes the characteristics of the application's code executed on GPUs to find opportunities for speeding up the tuning process. In this work, we shall consider gaming programs and their disassembly shader codes, since graphic functions are still the primary and most complicated applications on GPUs. Our work can be extended to other applications on GPUs – when these applications are mapped onto the GPU, they are translated to machine-level disassembly codes by drivers [5].

The application's code is referred to *shader* (shader code), a piece of program to instruct the GPU how to render each pixel. Since shaders support complex, low-level, generic programming, and due to their proximity to the hardware, they are an important tool for GPU designers [7–10]. When shader codes are not available from application developers, we generate *disassembly shaders* by running those graphics applications on GPUs using a publicly available graphics debugger, such as RenderDoc [11]. Graph analytics has been adopted in code analysis in recent years [12–15]. In shader code analysis, it can simplify the codes to structural level. In §3.2, we demonstrate that such graph structure is a good proxy for the shader code . Thus, various graph mining and machine learning algorithms on graphs are beneficial to the analysis of shader codes. Our framework can process machine-level codes to assist in GPU's performance improvement by reducing the numbers of frames from one application and the total number of applications that need to be investigated. We also identify frequently occurring low-performance code structures that can lead the way for efficient debugging.

1.3 Our Contributions and Roadmap

Our contributions can be summarized as follows.

– We are the first to introduce a graph mining framework for analyzing the shader structures within video games (§3). §3.1 shows the process of preparing shader graph datasets from disassembly shader codes. In §3.2, we compare the code content similarity with the graph structural similarity and establish that graph representations can well encode the structural characteristics of disassembly shader codes. The statistics of the graph datasets and their temporal evolutions within a game are discussed in §3.3.
– In §4, we demonstrate the functionality which is to predict an unknown frame's scene via the Graph Convolutional Neural Network [17] with explanations.
– §5 demonstrates that when there is a new application, first we find similar existing applications via frequent subgraphs-based clustering methods. Since the GPU has already been tuned for these similar existing applications, one can quickly find the optimal tuning for the new application. We next introduce a method to predict shader's efficiency using frequent subgraphs, and identify a new application's low-efficiency shaders, thus locating its frequent weak points.

With results from our analysis, domain experts can accelerate the optimization process by only tuning hardware parameters using representative frames and representative applications. In addition, hardware engineers can understand common patterns (i.e., frequent subgraphs) in shader codes and hence, improve the hardware architecture in the next generation of products by adding specific hardware structures or re-designing the driver to process frequent sub-patterns more efficiently. Our code is open-sourced [18] and its demonstration [45] video is available at the YouTube - https://youtu.be/0WxvrJ6KuqY.

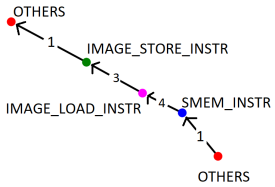
2 Related Work

With the advances in GPU architectures, researchers compared different generations of GPU designs [28]. Moya et al. [29] discussed the performance improvement due to various shader processing configurations. Many toolkits are developed by GPU designers and third-parties. AMD offers the **Radeon™Developer Tool Suite**, including Radeon™GPU Profiler, Memory Visualizer, GPU Analyzer, and Developer Panel [31], using which game developers obtain in-depth access to the GPU and they can profile the computing usage, analyze pipeline bottlenecks, and other inefficiencies while developing the games. NVIDIA also provides a few performance analysis tools for CUDA C/C++ optimization [30]. **Intel VTune™** is capable of monitoring CPUs, GPUs, and FPGAs to locate the most time-consuming part of the developer's code. Additionally, the third-party tools, e.g., the feed-forward measurement performance tool model in [32] and **HPCToolKit** [33] build the call path profiles for GPUs to let developers know their code performance translated from a high-level language model. The aforementioned tools optimize GPUs from a single application's point of view. They focus on a specific application's optimization and require the resource supports such as the application's source code. In contrast, our model is developed for GPU designers, who would like to optimize their GPUs for multiple applications, including the emerging ones. To the best of our knowledge, there is no publicly available work that uses graph mining and machine learning for shader

Table 1. An example of disassembly shader and the graph created

Disassembly shader instruction set
s_inst_prefetch 0x0003
s_load_dwordx8 s[16:23], s[2:3], null
v_mad_u32_u24 v2, s12, 16, v0
v_mad_u32_u24 v1, s13, 16, v1
s_waitcnt lgkmcnt(0)
image_load v[3:5], [v2,v1], s[16:23]
s_waitcnt vmcnt(0)
v_mov_b32 v6, v3
image_store v[3:6], [v2,v1], s[4:11]
s_endpgm

nodes and node-labels
0, *OTHERS*
1, *SMEM_INSTR*
2, *IMAGE_LOAD_INSTR*
3, *IMAGE_STORE_INSTR*
4, *OTHERS*



edges and edge-distances
(0,1), 1
(1,2), 4
(2,3), 3
(3,4), 1

code analysis. Adopting graph analysis for GPU performance improvement process from the code's perspective is a novel paradigm which we investigated.

3 Datasets and Characteristics

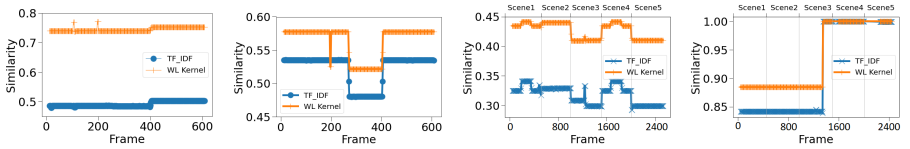
A *scene* in the video refers to a series of motions that happen in a single location and continuous time. The majority of online games have multiple scenes. Within a scene, there are multiple frames. Within a frame, there are a number of *draw calls* which instruct the graphics APIs to draw object. There are different types of shaders for each draw call. Primarily used shaders in the games that we consider are vertex shader (**VS**), pixel shader (**PS**), geometry shader (**GS**), compute shader (**CS**), hull shader (**HS**), local shader (**LS**), and export shader (**ES**). Each of them is one piece of shader code. We collected about 4 million disassembly shaders, corresponding to 64 scenes of 29 games, from our industry partners in GPU manufacturing. Our dataset includes both benchmarks (**3D Mark** and **VRMARK series**) and online games.

3.1 Graph Data Extraction

We describe the data preparation in two steps.

Step1: from video to shader code. There are a few open source tools that convert the video from a game into disassembly shaders such as AMD Radeon GPU Profiler [19] and RenderDoc [11].

Step2: from shader code to graph format. The long-time running instructions are represented as nodes along with node-labels based on the GPU functional blocks involved. For the rest of instructions such as arithmetic calculations, the number of such successive instructions between two nodes are counted and stored as the edge-distance. As table 1 shows, the instructions are categorized into a few groups (shown with different colors). In the node list, the first line s_inst_prefetch_0x003 is categorized in the group of *OTHERS* and is labeled as



(a) **GS** shader, **3DM** (b) **CS** shader, **3DM** (c) **CS** shader, **GTA5** (d) **HS** shader, **GTA5**

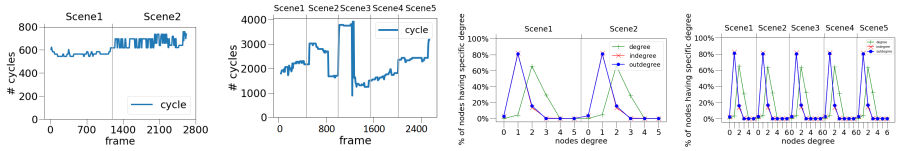
Fig. 1. Content (**TF-IDF**) vs. graph similarity (**WL kernel**)

node 0 (red color). The second line is categorized as *SMEM_INSTR* which uses scalar memory and it becomes node 1 (blue color). The consecutive short-running instructions are grouped as edges. For this reason, a few instructions are skipped until *IMAGE_LOAD_INSTR* (line 6, magenta color, this becomes node 2). Notice that this graph is a chain graph. Complex structures, e.g., branches, cycles, and complex patterns also exist in our graphs. In §3.3, we show the number of cycles over different frames.

3.2 Effectiveness of Graph Structure

A piece of shader code can be considered as an algorithm that specifies where to take input data and how they will perform computation and data storage. Looking purely at the text contents of shader codes, we use the widely-used **TF-IDF Vectorizer** [20] method to represent shader codes as vectors. We then compute the cosine similarity (a value in [0,1]) between a pair of vectors. Finally, we report the similarity between two consecutive frames which is the average similarity value over all pairs (one from the first frame and another from the second frame) from all distinct shader codes (considering a specific type of shader, e.g., compute shader) in these two frames. Figure 1 demonstrates text contents-based similarity values for consecutive frames from two games: 3D Mark Time Spy (**3DM TS**) and Grand Theft Auto 5 (**GTA5**), considering specific types of shaders, e.g., **GS**, **CS**, and **HS**. Figures 1(a)-(b) show that frame structures change slightly considering all 600 frames for **3DM TS** and both **GS** and **CS** shader types. Figures 1(c)-(d) show text contents-based similarity for **GTA5** using **CS** and **HS** as examples. From this five-scene dataset, a clear change at the boundary between scenes can be observed for **CS**. Within each scene, the similarity value may vary because of addition or deletion of a few pieces of shader codes.

Next, we convert shader codes into graphs (§3.1) and compute similarity between pairs of graphs via the *Weisfeiler-Lehman Graph Kernel* (**WL kernel**) [21]. The **WL kernel** maps an original graph into a sequence of graphs (kernel features), whose node attributes capture both structural and label information. We apply the **WL kernel** function from the **GraKel** library [22] to extract kernel features and compute the similarity between a pair of graphs. Analogously, we also report in Figure 1 the similarity between two consecutive frames by computing the average **WL kernel** similarity value over all pairs (one from the first frame and another from the second frame) from all distinct shader code graphs (considering a specific type of shader, e.g., compute shader) in these two frames. Furthermore, the pairwise graph similarity values are normalized between 0 and 1, by dividing it with $\sqrt{\mathit{\Pi}_i \mathbb{K}(G_i, Gi)}$, where $\mathbb{K}(G_i, Gi)$ is the **WL**



(a) **3DM FS**, # cycles over frames (b) **GTA5**, # cycles over frames (c) **3DM FS**, degree, in- and out-degree distributions (d) **GTA5**, degree, in- and out-degree distributions

Fig. 2. Evolution of structural properties over frames

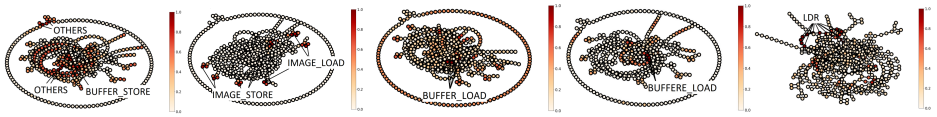
kernel-based self-similarity of graph G_i to itself, and we consider all G_i's from these two frames. We observe that the **WL kernel**-based similarity plots generally have the same trends as those based on **TF-IDF** text contents similarity. This indicates that the graph representation is a good proxy to reflect the contents in shader codes. Besides, graphs can represent both syntactic and semantic structures of code, compared to mainly syntactic notions in program text [16]. Therefore, we shall use graph structures of disassembly shader codes.

3.3 Graph Data Characteristics and Key Frames Selection

Each game consists of a large number of frames and draws for different rendering functions. We study the characteristics and evolution of shader code graphs across frames for each game and identify the *representative frames*. We show our results using 3D Mark Fire Strike (**3DM FS**, a benchmark game) and **GTA5** (an online game). We study graph changes across frames/scenes and show the variation of the number of nodes, edges, cycles, as well as the degree distribution. These provide an overview of the evolution of shader code graphs.

Figures 2 (a)-(b) depict the number of cycles existing in each frame for the two games. The x-axis shows successive frames and the y-axis indicates the total number of cyclic structures in that frame. **3DM FS** has more loop structures in Scene 2. In **GTA5**, majority of the frames in Scene 3 have more cycles, compared to that in frames from other scenes. It indicates that there are more branches and loops implemented in this part. Figures 2 (c)-(d) show the degree distribution of nodes from every scene. The x-axis presents various node degrees for each scene. The y-axis is the percentage of nodes having the specific degree, with respect to the total number of nodes in that scene. For each scene, node degrees vary from 1 to 5. The majority of nodes have two edges, with one in-edge and another out-edge, implying the chain-like structure. The in-degree and out-degree values vary from 0 to 5. Most nodes have balanced in- and out-degrees.

Structural changes across scenes. In Figure 1, **WL kernel** similarity results reveal that the shader code graph structures remain almost same with a small variation within one scene, but often have noticeable changes between two scenes. We can conclude that within the same scene, the algorithms adopted are roughly the same. However, across different scenes, the algorithms used can be quite different. Sudden changes in frame structures can be detected based on the changes in **WL kernel** similarity across successive frames. Within a specific scene, we select the frame as the *representative* that has the highest **WL kernel** similarity with all other frames in that scene. In this way, by studying this frame,



(a) Class=Scene 1 (b) Class=Scene 2 (c) Class=Scene 3 (d) Class=Scene 4 (e) Class=Scene 5

Fig. 3. GTA5: Scene prediction interpretability and important nodes

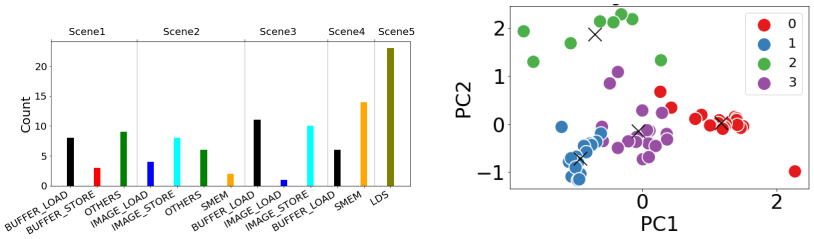
the GPU designers are able to cover most of the algorithmic patterns in that scene. If the algorithm structures do not vary too much within a scene, it is possible to reduce the number of times each algorithm is executed by storing the data into cache or local memory, and all these data can be retrieved in a very short time, compared to applying the algorithm again to render the image.

4 Predicting A Frame's Scene

In §3.3, we illustrate the similarities and difference of frame structures within scenes. Thus, it is possible to predict which scene a frame belongs to given an unknown frame, in a supervised manner. In particular, we employ a Graph Convolutional Neural Network (GCN) [17, 23] for classification of frames and thereby predict an unknown frame's scene. Our GCN architecture includes two graph convolutional layers with the first layer having 128 units and the second layer having 64 units. Both layers use the ReLU activation function. The next layer is a global mean pooling layer, followed by a softmax classifier.

All distinct shader code graphs in each frame are merged to form one single, disjoint, large graph. The resulting graph is labeled based on the frame's scene number. In **3DM FS**, there are two classes corresponding to two scenes . **GTA5** contains five classes, denoting five scenes. We apply 4-fold cross validation. The learning rate is set to 0.0005 and the dropout rate is 0.5. The number of epochs is 50 for **3DM FS** and 900 for **GTA5**. For **3DM FS**, training and test accuracy quickly reach 100%, indicating that the frames are easily separable across different scenes. For **GTA5**, after 900 epochs, the training and test accuracy reach 90.64% and 88.46%, respectively, with average ROC_AUC score 97.04%. This shows that the **GTA5** dataset, with five scenes, is relatively harder to classify.

Explainability study. We use the *Gradient-Weighted Class Activation Mapping* (**Grad-CAM**) interpretability method [24] to derive node importance in frame classification. In Figure 3, we show the graphs corresponding to five test frames, one from each scene of **GTA5**. Clearly, as the frames are different, their graph structures are different too. The nodes highlighted in the red color indicate those nodes having higher positive contributions to the activation of the specific class label output (i.e., scene number). Finally, we identify the top-20 nodes from each of these five frames (each belonging to a different scene) having the highest positive contributions. We count the node-labels present in those top-20 nodes from each test frame and report these counts corresponding to each scene in Figure 4(a). The x-axis indicates the node-labels (i.e., GPU functional blocks involved in the shader code instruction) from these important nodes. Same color bars represent that they are the same functional block. The figure shows that the most important GPU functional blocks considered to distinguish each scene by



(a) (b)

Fig. 4. (a) # GPU functional blocks in the top-20 nodes from each scene, having the highest positive contributions in scene prediction. (b) Scene clustering results with K-Means (shown with two principal components)

our GCN-based classification method are quite different. These results demonstrate the effectiveness and interpretability of our frame classification approach.

5 Tuning for A New Application

We first provide an overview of our method for analyzing existing and new game datasets. With the existing game datasets, we obtain representative frames from various scenes by studying graph similarity over frames (§3.3). Next, frequent subgraphs are extracted from representative frames (§5.1). The game datasets and scenes are clustered into different groups based on similarity of their frequent subgraphs (§5.2). Thus, only representative games from each group need to be investigated to cover the majority of graph patterns across existing games. This minimizes the tuning effort for engineers. Furthermore, frequent patterns from representative games are annotated as high-efficiency or low-efficiency by domain experts. The engineers can upgrade and tune the GPU architecture and hardware parameters by knowing the common patterns. When a new game is given, its representative frames and frequent subgraphs are extracted. Next, we verify if the new game is similar to any of the groups of existing games. If it belongs to one of them, the tuning result from similar existing games can be re-used. If not, we predict its low-efficiency shaders in a supervised manner.

5.1 Frequent Subgraphs Mining

We mine frequent subgraphs with the gSpan algorithm [25] from each of 64 scenes spanning across 29 different games. The *minimum support* (minsup) value is set to 25% of the number of graphs in each representative frame. A frequent subgraph pattern is a subgraph that appears in at least minsup graphs in a graph database. For even higher support values, the frequent patterns get smaller and less in number. For lower support values, gSpan does not terminate in one day. Among output frequent subgraphs, we consider those with the number of nodes at least 5. In this way, we eliminate smaller subgraphs which appear frequently; however, they are less useful in hardware tuning.

5.2 Scenes Clustering

We identify representative scenes and games for GPU performance tuning, thus improving the tuning efficiency by eliminating duplicated works on games having

similar characteristics. We consider all maximal frequent subgraphs as features for each scene to cluster them. Thus, we get 64 data points (one for every scene), each with dimensionality 825 (representing all maximal frequent subgraphs). For a specific scene S_i and dimension d_j, the value $S_i[d_j] \in (0, 1)$ is computed as the maximum pairwise **WL kernel** similarity between each frequent subgraph of S_i and the maximal frequent subgraph corresponding to d_j. Next, we apply K-Means [20], Gaussian Mixture [26], and Birch algorithms [27] for clustering these data points. Before clustering, Principal Component Analysis (PCA) is adopted to reduce the number of features so to avoid the curse of dimensionality. The optimal number of clusters is determined based on the Elbow method [43]. The Elbow method selects K where the within-cluster variance no longer decreases sharply (e.g., 4 for K-Means in our case).

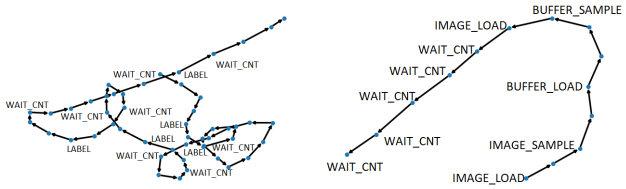
Experimental results. We observe that K-Means and Gaussian Mixture produce clustering with 93.75% overlap, while BIRCH have 95.31% overlap with K-Means clustering. Scenes with different grouping results are near the boundary of two groups based on the three clustering methods. We visualize the K-Means clustering in Figure 4(b). Each group is labeled with a different color. The cross sign refers to the centroid of that group. Scenes in the same group have similar frequent subgraph structures, which means that the frequent algorithmic patterns used in these scenes are also similar. Therefore, selecting one scene, which is the nearest to the centroid of each group, for performance analysis and tuning will be both effective and efficient.

5.3 A New Game's Shader Efficiency Prediction

The frequently occurring sub-patterns from shader codes are valuable also for the code efficiency analysis and improvement. The overall GPU efficiency can be improved if the frequent patterns can be efficiently processed. The engineers can upgrade and tune the GPU architecture and hardware parameters by knowing the common patterns.

Due to their larger sizes and relatively complex shapes, by looking at a complete piece of shader code, it is often difficult even for domain experts to predict its efficiency or to identify the frequently occurring weaknesses. However, we observe that our industry collaborators can quickly recognize if a frequent shader subgraph is inefficient or not, due to its small size. To this end, we aim at predicting the efficiency of frequent shader subgraphs in a supervised manner. The mined frequent subgraphs are annotated as high (Label 1) or low efficiency/ utilization (Label 0) by our industry collaborators based on GPU functional blocks and latency information. Moreover, we associate the counts of various GPU functional blocks in frequent subgraphs as their features. In this way, we get our data points (one for every frequent subgraph) having dimensionality 43 (representing all GPU functional blocks involved). The dataset is unbalanced with 88.02% in Label 1 and 11.98% having Label 0.

Experimental results. We employ the Random forest classifier, with 4-fold cross validation, to predict the efficiency of frequent shader subgraphs. We find the average accuracy to be 0.9865, AUC-PRC 0.9943, and AUC-ROC 0.9574.



(a) **Subgraph 1** (b) **Subgraph 2**

Fig. 5. Two examples of low-efficiency shader patterns

These results demonstrate that we can predict the efficiency of frequent shader subgraphs in a supervised manner. Moreover, given a new game, one can predict its performance by mining its own frequent subgraphs and using our supervised frequent subgraphs classification model.

Case study. Figure 5 presents two examples of low-efficiency shader patterns which are correctly predicted by our method. For the subgraph in Figure 5(a), there are many branches (indicated by LABEL) and long-time wait commands (indicated by WAITCT); it is likely that during jump or wait time, GPU resources are under-utilized. For the pattern in Figure 5(b), the structure, though simple, consists of many LOAD and STORE actions, which take longer time.

6 Conclusions

We proposed a unified graph model to denote shader codes as graphs, employed graph mining and machine learning to analyze disassembly shader code structures. We investigated a few research questions: Our single game analyses reveal that the algorithms used in each frame within a scene are similar, thus we can predict the scene for an unknown frame, and identify the representative frame from each scene. Our inter-game analyses demonstrate that we can group scenes and games based on their frequent sub-structures in the shader codes. As a result, only one or a few sample scenes are sufficient to cover the typical patterns in that group; hence, engineers can focus on the specific scenes and games for performance tuning. Finally, our novel shader efficiency prediction method based on frequent subgraphs accelerates the tuning process by identifying frequent algorithmic structures, including frequently occurring weaknesses. In future, we shall consider other portions of the shader code, such as input data, in addition to algorithmic structures, as well as disassembly machine codes from different applications, for a more comprehensive analysis. Besides, the frequent subgraph mining algorithm can be improved, e.g., SPMiner [42], to consider various attributes after additional information is added to the graphs, thereby extracting more sophisticated features.

7 Acknowledgement

This research is supported by Novo Nordisk Foundation grant NNF22OC0072415 and Singapore MOE tier-2 grant 2019-T2-2-042. Lin Zhao acknowledges support from Nanyang Technologies University, Advanced Micro Devices Co., Ltd.

References

1. S. Huang, S. Xiao, and W.-c. Feng. 2009. On the Energy Efficiency of Graphics Processing Units for Scientific Computing. In IPDPS
2. J.D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. 2008. GPU Computing. Proc. IEEE 96, 5 (2008), 879–899
3. AMD. [n.d.]. How to Tune GPU Performance Using Radeon Software. https://www.amd.com/en/support/kb/faq/dh2-020.
4. Foster, I., Kesselman, C., Nick, J., Tuecke, S.: The Physiology of the Grid: an Open Grid Services Architecture for Distributed Systems Integration. Technical Report, Global Grid Forum (2002)
5. P. Zandbergen. 2022. Machine Code and High-level Languages: Using Interpreters and Compilers. https://study.com/academy/lesson/machine-code-andhigh-level-languages-using-interpreters-and-compilers.html/
6. 2021. Pipelines and Shaders with Direct3D 12. https://docs.microsoft.com/enus/windows/win32/direct3d12/pipelines-and-shaders-with-directx-12.
7. C. S. de La Lama, P. Jääskeläinen, H. Kultala, and J. Takala. 2019. Programmable and Scalable Architecture for Graphics Processing Units. Trans. High Perform. Embed. Archit. Compil. 5 (2019), 21–38.
8. S. T. Fam and A. Sowerby. 2015. Shader Program Profiler https://patents.google.com/patent/US9799087B2
9. X. Ren and M. Lis. 2021. CHOPIN: Scalable Graphics Rendering in Multi-GPU Systems via Parallel Image Composition. In IEEE International Symposium on High-Performance Computer Architecture, HPCA. IEEE, 709–722.
10. S. I. Zaidi, S. T. Fam, P. Lotfi, V. R. Indukuru, J. Pan, A. M. Sowerby, and J.-L.Duprat. 2018. Shader Profiler.
11. B. Karlsson. 2021. RenderDoc. https://renderdoc.org/docs/index.html
12. H. Gascon, F. Yamaguchi, D. Arp, and K. Rieck. 2013. Structural Detection of Android Malware using Embedded Call Graphs. In AISec.
13. R. Alanazi, G. Gharibi, and Y. Lee. 2021. Facilitating Program Comprehension with Call Graph Multilevel Hierarchical Abstractions. J. Syst. Softw. 176 (2021), 110945
14. U. Tekin and F. Buzluca. 2014. A Graph Mining Approach for Detecting Identical Design Structures in Object-oriented Design Models. Sci. Comput. Program. 95 (2014), 406–425.
15. 2021. greenspector https://greenspector.com/en/home/
16. M. Allamanis, M. Brockschmidt, and M. Khademi. 2018. Learning to Represent Programs with Graphs. In ICLR.
17. T. N. Kipf and M. Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In ICLR.
18. L. Zhao, A. Khan, and R. Luo. 2022. Our Code and Datasets. https://github.com/forest2022/graphForest.
19. 2021. Radeon™ GPU Analyzer. https://https://gpuopen.com/rga/
20. J. Leskovec, A. Rajaraman, and J. D. Ullman. 2014. Mining of Massive Datasets, 2nd Ed. Cambridge University Press.
21. N. Shervashidze, P. Schweitzer, E. J. v. Leeuwen, K. Mehlhorn, and K. M. Borgwardt. 2011. Weisfeiler-Lehman Graph Kernels. J. Mach. Learn. Res. 12 (2011), 2539–2561
22. G. Siglidis, G. Nikolentzos, S. Limnios, C. Giatsidis, K. Skianis, and M. Vazirgiannis. 2020. GraKeL: A Graph Kernel Library in Python. Journal of Machine Learning Research 21, 54 (2020), 1–5.

23. W. L. Hamilton, Z. Ying, and J. Leskovec. 2017. Inductive Representation Learning on Large Graphs. In NeurIPS.
24. R. R. Selvaraju, M. Cogswell, A. Das, R. Vedantam, D. Parikh, and D. Batra. 2017. Grad-CAM: Visual Explanations from Deep Networks via Gradient-Based Localization. In ICCV.
25. X. Yan and J. Han. 2002. gSpan: Graph-Based Substructure Pattern Mining. In ICDM.
26. X. Lin, X. Yang, and Y. Li. 2019. A Deep Clustering Algorithm based on Gaussian Mixture Model. Journal of Physics: Conference Series 1302 (08 2019), 032012.
27. T. Zhang, R. Ramakrishnan, and M. Livny. 1997. BIRCH: A New Data Clustering Algorithm and Its Applications. Data Min. Knowl. Discov. 1 (06 1997), 141–182.
28. D. Schaa and D. R. Kaeli. 2009. Exploring the Multiple-GPU Design Space. In IPDPS.
29. V. Moya, C. Gonzalez, J. Roca, A. Fernandez, and R. Espasa. 2005. Shader Performance Analysis on a Modern GPU Architecture. In MICRO.
30. 2021. NVIDIA Performance Analysis Tools https://developer.nvidia.com/performance-analysis-tools
31. 2021. Radeon™ Developer Tool Suite https://gpuopen.com/
32. B. Welton and B. P. Miller. [n.d.]. Diogenes: Looking for an Honest CPU/GPU Performance Measurement Tool.
33. K. Zhou, L. Adhianto, J. Anderson, A. Cherian, D. Grubisic, M. Krentel, Y. Liu, X. Meng, and J. Mellor-Crummey. 2021. Measurement and Analysis of GPUaccelerated Applications with HPCToolkit. Parallel Comput. 108 (2021), 102837
34. C. A. Baddouh, M. Khairy, R. N. Green, M. Payer, and T. G. Rogers. 2021. Principal Kernel Analysis: A Tractable Methodology to Simulate Scaled GPU Workloads. In MICRO.
35. J.-C. Huang, L. Nai, H. Kim, and H.-H. S. Lee. 2014. TBPoint: Reducing Simulation Time for Large-Scale GPGPU Kernels. In IPDPS
36. M. Kambadur, S. Hong, J. Cabral, H. Patil, C. Luk, S. Sajid, and M. A. Kim. 2015.Fast Computational GPU Design with GT-Pin. In IISWC.
37. Z. Yu, L. Eeckhout, N. Goswami, T. Li, L. K. John, H. Jin, C.-Z. Xu, and J. Wu. 2015. GPGPU-MiniBench: Accelerating GPGPU Micro-Architecture Simulation. IEEE Trans. Computers 64, 11 (2015), 3153–3166.
38. D. Sadyrin, A. Dergachev, I. Loginov, Iu. N. Korenkov, and A. Ilina. 2019. Application of Graph Databases for Static Code Analysis of Web-Applications. In MICSECS.
39. J. Liu. 2020. Enabling Static Program Analysis Using A Graph Database. Ph.D. Dissertation. Wright State University.
40. A. Nair, A. Roy, and K. Meinke. 2020. funcGNN: A Graph Neural Network Approach to Program Similarity. In ESEM.
41. A. V. Phan, M. L. Nguyen, and L. T. Bui. 2017. Convolutional Neural Networks over Control Flow Graphs for Software Defect Prediction. In ICTAI.
42. A. Z. Wang, J. You, and J. Leskovec. 2020. Frequent Subgraph Mining by Walking in Order Embedding Space. In GRL Workshop
43. R. L. Thorndike. 1953. Who Belongs in the Family? Psychometrika 18 (1953), 267–276.
44. 2021. Unreal Engine. https://https://www.unrealengine.com/en-US/
45. L. Zhao, A. Khan, and R. Luo. 2022. ShaderNet: Graph-based Shader Code Analysis to Accelerate GPU's Performance Improvement (Demonstration). In GRADES-NDA Workshop