ShaderNet: Graph-based Shader Code Analysis to Accelerate GPU's Performance Improvement

Lin Zhao
Nanyang Technological University
Singapore
lin018@e.ntu.edu.sg

Arijit Khan
Aalborg University
Denmark
arijitk@cs.aau.dk

Robby Luo
Advanced Micro Devices Co., Ltd.
China
robby.luo@amd.com

ABSTRACT

This paper demonstrates ShaderNet — our graph analytics framework with shader codes, which are machine-level codes and are important for GPU designers to tune the hardware, e.g., adjusting clock speeds and voltages. Due to a wide spectrum of 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 present a system, ShaderNet, which converts shader codes into graphs, and applies advanced graph mining and machine learning techniques to simplify shader graphs analysis in an effective and explainable manner. By studying shader codes' evolution with temporal graphs analysis and structure mining with frequent subgraphs, we demonstrate several key functionalities of our framework, such as a frame's scene detection, clustering scenes, and a new application's inefficient shaders prediction, which can accelerate GPU's performance tuning. Our code base and demonstration video are at: **https://lzlz15.github.io/D_E_M_O/**.

CCS CONCEPTS

• **Computing methodologies → Graphics processors**; **Machine learning algorithms**; • **Mathematics of computing → Graph algorithms**.

KEYWORDS

GPUs, Shader Code, Graph Analysis, Machine Learning on Graphs

ACM Reference Format:
Lin Zhao, Arijit Khan, and Robby Luo. 2022. ShaderNet: Graph-based Shader Code Analysis to Accelerate GPU's Performance Improvement. In *Joint Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA) (GRADES & NDA'22), June 12, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3534540.3534688

1 INTRODUCTION

In earlier decades, the graphics processing unit (GPU) was a specialized microprocessor to offload graphically intense applications that created a burden on the CPU. Afterwards, GPUs have become general-purpose, programmable, and many-core processors [5, 10, 26, 32]. **(1)** Online and multi-player video games with special effects, virtual reality, 3D graphics, and 4K screens demand serious computing power. GPUs have become essential to quickly render and maintain realistic images in these applications. **(2)** Computer-aided design (CAD) softwares that visualize objects in 3D rely on GPUs to draw such models in real time as one rotates or moves them. GPUs' parallel processing ability also makes them faster and easier to render videos and graphics in higher-definition formats. **(3)** GPU-accelerated analytics in high-performance computing (HPC) enables data science workflows over billions of records, such as gene mapping and clinical trials. **(4)** One of the most exciting applications of GPUs involve deep learning and machine learning. GPUs can process tons of data and train deep neural networks for image and video analytics, speech recognition, and natural language processing. **(5)** GPUs play a key role in database management systems (DBMS) such as GPU Databases [27], GPU-assisted query optimization [14, 15, 28, 39, 40], visual analytics [31], image data processing [34], and data management for machine learning [11].

Challenges in GPU tuning. From GPU designers' (e.g., AMD, Intel, NVIDIA, etc.) point-of-view, designing high-performance, low-power, and small-area chips is critical. Based on feedback from our industry partners, GPU chip manufacturers experience three major challenges. **(1)** While the hardware must be tuned for specific applications running on it, GPU designers may not obtain application source codes from app developers (e.g., game developers, CAD software developers, or algorithm developers). Application codes are often complied into binary files for security reasons. Thus, code development behind applications are not transparent to GPU designers. **(2)** As the landscape of GPU applications is growing wider, there are a large number of shader codes emerging from them. Engineers generally tune GPUs (i.e., adjusting clock speeds and voltages, VRAM and fan tuning, memory timing) based on a few benchmark applications or hand-crafted key frames selected by domain experts. Such manual tunings miss important information and can be suboptimal. In addition, the technical knowledge required for GPU tuning is high, since mis-configuration will result in an unstable system, e.g., system crashes, hangs, and/or graphical corruption [8]. **(3)** New applications are emerging and it becomes tedious for GPU designers to tune GPUs for every new application.

Our solution and contributions. To solve the above challenges, we build ShaderNet — our system analyzes the characteristics of different applications using graph mining and machine learning tools, to accelerate the GPU's tuning process. In this paper, we shall consider video games dataset for experiments. ShaderNet can be extended to other applications, since when these applications are complied to execute on the GPU, they are represented in machine-level language format [41], which can be input to ShaderNet.

Graphics applications call graphics APIs (e.g., OpenGL, Microsoft DirectX), which define a programmable graphics pipeline that is

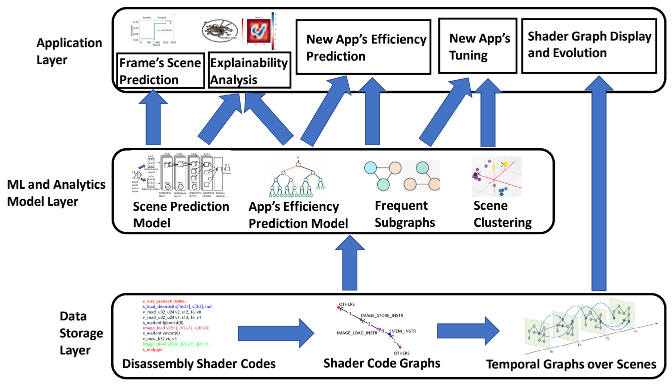


Figure 1: Architecture of ShaderNet

mapped onto the GPU [3]. A *shader code* is a piece of program from these graphics APIs that runs in the graphics pipeline, and instructs 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 [9, 12, 30]. When application codes are not available from application developers, we generate *disassembly shaders* by running those applications on GPUs using a graphics debugger, e.g., RenderDoc [19]. Next, we convert shader codes into directed, attributed graphs: the commands requiring relatively long time (e.g., accessing memory) are represented as nodes; successive short-running commands between two nodes are grouped as an edge between the two nodes.

We notice that our graphs can well-represent the structural characteristics of disassembly shader codes by comparing the code content similarity with the graph structural similarity. Besides, graphs can represent both syntactic and semantic structures of code, thus would be more effective than mainly syntactic notion of program text [7]. We employ various graph mining and machine learning algorithms for shader codes analysis in a holistic and explainable manner, including key frames identification, a frame's scene detection, clustering scenes, shader codes' structure mining with frequent subgraphs, and evolution via temporal graph analysis.

Given disassembly shaders of a new application, our system **(1)** predicts its low-efficiency shaders to guide GPU designers locating the weak points, so that they can develop targeted solution either in the software optimization, or in the hardware architecture; and **(2)** finds existing applications that are most similar to the new application. Since the GPU has been tuned for these similar existing applications, we can easily find the optimal configuration for the new application.

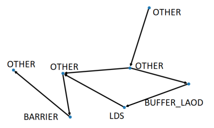
We discuss our demonstration plan, with visual, predictive, and mining query forms, as well as ShaderNet's performance in § 3. Our codebase is open-sourced [42] and a video demonstration is available at: **https://youtu.be/0WxvrJ6KuqY**.

Related work. Several toolkits are developed by GPU designers and third-parties to profile the GPU performance. AMD offers the Radeon^{TM} Developer Tool Suite including Radeon^{TM} GPU Profiler [4]. Intel VTune^{TM} can monitor GPUs to identify and fix performance bottlenecks. Nvidia uses a tool to access detailed information about GPU usage, such as hardware counters inside GPUs [2]. Other third-party tools, e.g., HPCToolKit [43] and [36] build call path profiles for GPUs to help developers access their code performance.

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

Disassembly shader instruction set
s_inst_prefetch 0x0003
s_bfe_u32 s1, s2
s_cbranch_scc1 label68
v_mad_u32_u24 v2, s12, 16, v0
tbuffer_load_format_x
v_mov_b32 v6, v3
ds_write2_b16
label_68
s_waitcnt vmcnt(0)
s_barrier
v_mov_b32 v1, v2
s_endpgm

nodes and node-labels	
0,	*OTHERS*
1,	*OTHERS*
2,	*OTHERS*
3,	*BUFFER_LOAD*
4,	*LDS*
5,	*BARRIER*
6,	*OTHERS*



edges and edge-distances	
(0,1),	2
(1,2),	1
(1,3),	2
(3,4),	2
(2,5),	2
(4,2),	1
(5,6),	2

Greenspector [1] uses Abstract Syntax Tree (AST) representation of code to analyze power and memory consumptions.

The aforementioned tools optimize GPUs from a single application's point of view. They concentrate on optimizing certain applications and require resource supports such as the application's source code. In contrast, our system, ShaderNet is aimed at GPU designers, who would like to optimize the new generation of GPU products for multiple applications, including the emerging ones. ShaderNet applies graph mining and machine learning techniques over a number of applications to accelerate the whole debugging process, by reducing the numbers of frames from one application and the total number of applications that need to be investigated. We also locate weak points in applications, which can lead the way to improving the overall hardware performance.

Graph analytics has been adopted in static code and call graphs analysis for program similarity, bugs and malware detection [1, 6, 13, 16, 17, 25, 35]. Frequently used graph representations in program analysis are abstract syntax tree (AST), control graphs (CG), control flow graphs (CFG), and program dependency graphs (PDG). Management and optimizations of codes, machine learning models and workflows, data frames and libraries have recently become popular in the data management community with the prevalence of data science [21–24, 29, 37]. Unlike these graphs constructed from codes in high-level programming languages, nodes and edges in our framework represent the hardware resource groups and the number of consecutive, short-running instructions, respectively, which are critical for machine-level shader codes analysis. To the best of our knowledge, ShaderNet [42], which we demonstrate in this work, is the first publicly available toolkit that adopts advanced graph mining and machine learning techniques for shader code analysis to accelerate the GPU's performance tuning process.

2 SYSTEM OVERVIEW

Solution architecture. Figure 1 presents ShaderNet's architecture and the interaction between its three layers. The core layers of ShaderNet – ML and analytics model layer and application layer – are implemented in Python 3.8. The Displayer, which permits visualization and end-user interaction at the application layer, is developed with the D3.js library (http://d3js.org/).

Table 2: Key software components of ShaderNet

Shader Graph's Structural Evolution	**Description**
node_count.py	Variation of node counts in shader graphs across scenes and plot the result
edge_count.py	Variation of edge counts in shader graphs across scenes and plot the result
cycle_count.py	Variation of cycle counts in shader graphs across frames and plot the result
degree_count.py	Variation of node degrees in shader graphs across frames and plot the result
Scene Prediction and Explainability	**Description**
merge_allGraphs_PerFrame.py	Merge shader graphs from a frame to one disjoint graph for that frame
GCN_with_CG.py	Predict which scene the unknown frame belongs to using graph convolutional neural network (GCN) and explainability method saliency map with contrastive gradients (CG)
New App's Tuning	**Description**
gSpan.py	Frequent subgraphs mining
fsm_file_to_edgelist_hash.py	Convert results from gSpan to graph edge list and node list
graph_similarity_measure.py	Measure pair-wise similarity value between graphs in each scene dataset
prepare_dataset_culstering.py	Select the highest similarity one from pair-wise measurement and form dataset for clustering
Kmeans_PCA.py	Cluster games into groups using K-Means with PCA model. The result is viewed in a plot
New App's Inefficient Shaders Prediction	**Description**
randomForest.py	Classify frequent sub-structures into high/low efficiency

Data storage layer. Disassembly shader codes are stored in .sp3 files. An example of a disassembly shader code is given in Table 1. Instructions are divided into a few groups (shown with different colors) based on the GPU functional blocks in each line. Only the long-running commands in a shader code (e.g., accessing memory) are represented as nodes; the GPU functional blocks are used as node labels. Consecutive short-running commands (e.g., arithmetic computations) between two nodes are grouped together as one weighted, directed edge. The count of the short-running commands across an edge is assigned as the edge's weight. The graphs are stored in edge-list and vertex-list format. Next, shader code graphs are loaded as Python objects via the NetworkX library (https://networkx.org/). We store distinct shader graphs per frame and across scenes — these form our snapshot-based temporal graph datasets to study the evolution of shader code structures over different frames and scenes within an application (e.g., a game).

ML and analytics model layer. The second layer stores ML models and pre-processed mining information that are useful for interactive and online queries in the application layer. We mine and store frequent subgraphs (e.g., via the gSpan algorithm [38]) from the representative frame's shader code graphs for each scene. The minimum support value is set to 25% of the number of graphs in each representative frame. The minimum number of nodes in the resulting subgraphs is set to 5. In this way, smaller subgraphs with more appearances are removed, since they do not add much value in our overall analysis. We further cluster the scenes by using frequent subgraphs as features. Before applying a clustering algorithm, e.g., K-Means, Gaussian Mixture, Birch, etc., we conduct Principle Component Analysis (PCA)-based dimensionality reduction to overcome the curse of dimensionality.

We additionally obtain labels (e.g., efficient/ inefficient) of frequent shader subgraphs from our industry collaborators. Such labeling is relative easy for domain experts due to small sizes of frequent subgraphs. Next, we build and store a random forest model using labeled frequent subgraphs for predicting a new application's inefficient shaders.

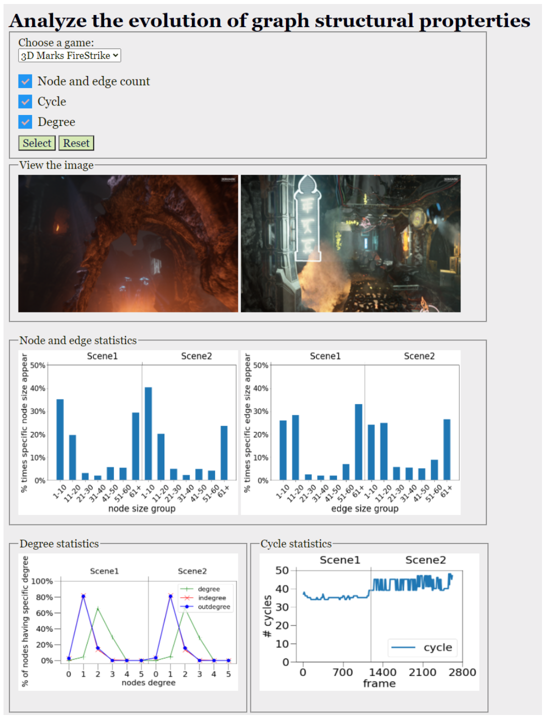


Figure 2: Shader code graphs' structural evolution

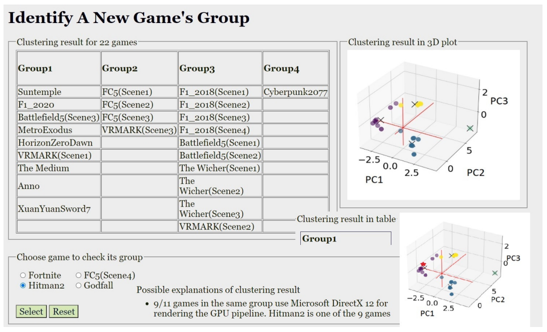
Finally, we also pre-train and store a graph convolutional neural network (GCN) [18, 20] based classifier for the scene prediction of an unknown frame.

Application layer. This layer supports predictive queries, visualization, and interaction with end-users. Our displayer module shows temporal evolution of shader code graphs and their structural properties (e.g., node degree distribution, count of cycles) across different frames and scenes. We find that within the same scene, shader graphs across frames are roughly the same. However, across different scenes, frames and shader graphs can be quite different. This facilitates automatic identification of the key frames within each scene. Using our pre-trained models and mined information, we also predict an unknown frame's scene, a new application's inefficient shaders, and we find existing applications (for which the GPU has already been tuned) that are most similar to the new application. We provide interpretation of our scene prediction results using saliency map with contrastive gradients (CG) [33], that is, ShaderNet identifies the most important GPU functional blocks having higher positive contributions to the scene prediction.

Software. Table 2 shows main software components of ShaderNet related to four key applications: shader graph's structural evolution, an unknown frame's scene prediction with explainability, a new application's tuning based on clustering with known applications, and a new application's inefficient shaders prediction.

3 DEMONSTRATION PLAN

We collected about 4 millions disassembly shaders, corresponding to 64 scenes of 29 games, from our industry partners in the GPU



Identify A New Game's Group

Clustering result for 22 games

Group1	Group2	Group3	Group4
Suntemple	FC5(Scene1)	F1_2018(Scene1)	Cyberpunk2077
F1_2020	FC5(Scene2)	F1_2018(Scene2)	
Battlefield5(Scene3)	FC5(Scene3)	F1_2018(Scene3)	
MetroExodus	VRMARK(Scene3)	F1_2018(Scene4)	
HorizonZeroDawn		Battlefield5(Scene1)	
VRMARK(Scene1)		Battlefield5(Scene2)	
The Medium		The Wicher(Scene1)	
Anno		The Wicher(Scene2)	
XuanYuanSword7		The Wicher(Scene3)	
		VRMARK(Scene2)	

Clustering result in table

Group1

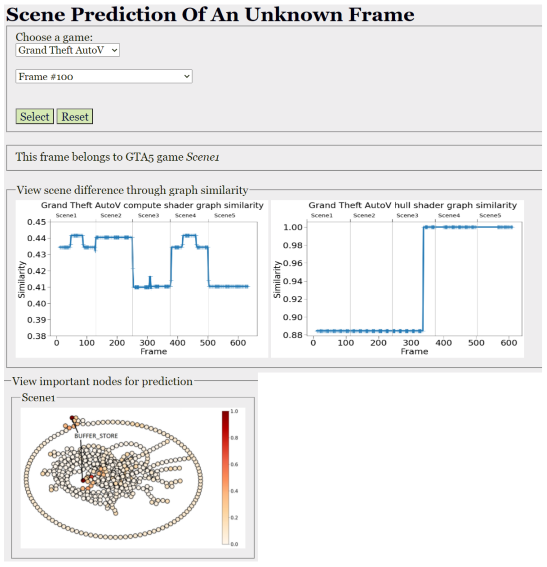
Choose game to check its group

Fortnite FC5(Scene4) [✓ Hitman2] Godfall

Possible explanations of clustering result

- 9/11 games in the same group use Microsoft DirectX 12 for rendering the GPU pipeline. Hitman2 is one of the 9 games

Figure 3: New game scenes' clustering



Scene Prediction Of An Unknown Frame

Choose a game: Grand Theft AutoV

Frame #100

This frame belongs to GTA5 game *Scene1*

Select
Reset
View scene difference through graph similarity
Grand Theft AutoV compute shader graph similarity
Grand Theft AutoV hull shader graph similarity
Scene1
Scene2
Scene3
Scene4
Scene5
Similarity
Frame
View important nodes for prediction
Scene1
BUFFER_STORE

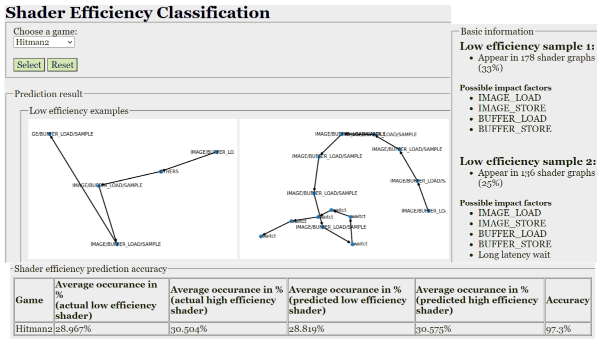
Figure 4: Unknown frame's scene prediction

manufacturing domain. Many of these game datasets contain several scenes and multiple frames per scene. A scene in the video refers to a series of motions that happen in a single location and continuous time. Within a scene, there are multiple (600∼1200) frames. Each frame refers to a still image. Within a frame, there are a number of draw calls which instruct the graphics APIs to draw objects. We construct a shader graph corresponding to every draw call, each having small numbers (e.g., 10∼600) of nodes and edges (§2). The number of graphs per frame can be around 8K. When we consider all shader codes within a frame, the corresponding graph size can be modest (≈11K nodes and 14K edges). Besides chain-like structures, complex patterns such as branches and cycles also exist in our graph datasets.

For demonstration with these games, we shall present a web application on a laptop (**demonstration video** available at [42]).

Visualization. The monitor window in Figure 2 shows shader code graphs' structural evolution over different scenes for a user-selected game (e.g., 3D Marks Firestrike). The top portion shows two different scenes, the middle and bottom portions display variation of graph structural properties (e.g., node, edge, and cycle counts, node degree distribution).

Mining. In the monitor window in Figure 3, the user can interactively select scenes from different games, cluster them based on



Shader Efficiency Classification

Choose a game: Hitman2

Basic information

Low efficiency sample 1:

- Appear in 178 shader graphs (33%)

Possible impact factors

- IMAGE_LOAD
- IMAGE_STORE
- BUFFER_LOAD
- BUFFER_STORE

Low efficiency sample 2:

- Appear in 136 shader graphs (25%)

Possible impact factors

- IMAGE_LOAD
- IMAGE_STORE
- BUFFER_LOAD
- BUFFER_STORE
- Long latency wait

Shader efficiency prediction accuracy

Game	Average occurance in % (actual low efficiency shader)	Average occurance in % (actual high efficiency shader)	Average occurance in % (predicted low efficiency shader)	Average occurance in % (predicted high efficiency shader)	Accuracy
Hitman2	28.967%	30.504%	28.819%	30.575%	97.3%

Figure 5: New game's efficiency prediction

mined frequent subgraphs, and visualize clustering results (tabular and 3D formats). Our industry collaborators verified that the clustering results are effective in identifying important factors, e.g., scenes from the same game, game engines and graphics APIs employed. We include such reasons in the monitor window. Selecting one scene, which is the nearest to the centroid of each group, for performance analysis and tuning will be both effective and efficient.

Prediction. We demonstrate two types of prediction results: Figure 4 shows the scene prediction result for a user-selected frame from various games. For instance, the user interactively selects frame #100 from Grand Theft Auto V, and our prediction result correctly reports it from scene 1. On average, our GCN-based scene prediction accuracy reaches ≈ 90% for most of the games. Figure 5 displays efficiency prediction results for a new scene (selected interactively from several options). Recall that we use a random forest classifier built on top of annotated (efficient/inefficient) frequent subgraphs. We find the average accuracy to be 0.96 in predicting a new application's inefficient shader patterns.

Explainability. We display explainability results associated with the prediction. Figure 4 shows the most important GPU functional blocks having higher positive contributions to the scene prediction, identified by saliency map with contrastive gradients (CG). In Figure 5, we depict a few examples of high-efficiency and low-efficiency subgraphs (i.e., shader code fragments) from the selected scene and provide reasonings behind low-efficiency shaders.

4 CONCLUSIONS

Our demonstration of ShaderNet shows an interesting application of graph mining and machine learning for shader codes analysis that can accelerate GPU's performance tuning. By analyzing machine-level shader codes, our framework can reduce the numbers of frames from one application and the total number of applications that need to be investigated, thereby accelerating GPU's performance improvement. In future, we shall consider other portions in the shader code, e.g., input data, in addition to algorithm structures, and disassembly machine codes from different applications, for more comprehensive analyses.

5 ACKNOWLEDGEMENT

Lin Zhao acknowledges support from Nanyang Technologies University, Advanced Micro Devices Co., Ltd., and Singapore Economic Development Board. Arijit Khan acknowledges support from the Novo Nordisk Foundation grant NNF22OC0072415.

REFERENCES

[1] 2021. greenspector. https://greenspector.com/en/home/.

[2] 2021. NVIDIA Performance Analysis Tools . https://developer.nvidia.com/performance-analysis-tools.

[3] 2021. Pipelines and Shaders with Direct3D 12. https://docs.microsoft.com/en-us/windows/win32/direct3d12/pipelines-and-shaders-with-directx-12.

[4] 2021. Radeon™ Developer Tool Suite . https://gpuopen.com/.

[5] J. A. H. Aguilar, J. C. Bonilla-Robles, J. C. Zavala Díaz, and A. Ochoa. 2019. Real-time Video Image Processing through GPUs and CUDA and its Future Implementation in Real Problems in a Smart City. *Int. J. Comb. Optim. Probl. Informatics* 10, 3 (2019), 33–49.

[6] R. Alanazi, G. Gharibi, and Y. Lee. 2021. Facilitating Program Comprehension with Call Graph Multilevel Hierarchical Abstractions. *J. Syst. Softw.* 176 (2021), 110945.

[7] M. Allamanis, M. Brockschmidt, and M. Khademi. 2018. Learning to Represent Programs with Graphs. In *ICLR*.

[8] AMD. [n. d.]. How to Tune GPU Performance Using Radeon Software. https://www.amd.com/en/support/kb/faq/dh2-020.

[9] V. M. D. Barrio, C. González, J. Roca, A. Fernández, and R. Espasa. 2005. Shader Performance Analysis on a Modern GPU Architecture. In *MICRO*.

[10] A. Benhamida, M. Kozlovszky, and S. Szénási. 2019. GPU Usage Trends in Medical Image Processing. In *SACI*.

[11] C. Chai, J. Wang, Y. Luo, Z. Niu, and G. Li. 2022. Data Management for Machine Learning: A Survey. *IEEE Transactions on Knowledge and Data Engineering* (2022).

[12] 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.

[13] L. DeRose and F. Wolf. 2002. CATCH - A Call-Graph Based Automatic Tool for Capture of Hardware Performance Metrics for MPI and OpenMP Applications. In *Euro-Par*.

[14] H. Doraiswamy and J. Freire. 2020. A GPU-friendly Geometric Data Model and Algebra for Spatial Queries. In *SIGMOD*.

[15] H. Doraiswamy, H. T. Vo, C. T. Silva, and J. Freire. 2016. A GPU-based Index to Support Interactive Spatio-temporal Queries over Historical Data. In *ICDE*.

[16] F. Eichinger, K. Böhm, and M. Huber. 2008. Mining Edge-Weighted Call Graphs to Localise Software Bugs. In *ECML/PKDD (Lecture Notes in Computer Science)*.

[17] H. Gascon, F. Yamaguchi, D. Arp, and K. Rieck. 2013. Structural Detection of Android Malware using Embedded Call Graphs. In *AISec*.

[18] W. L. Hamilton, Z. Ying, and J. Leskovec. 2017. Inductive Representation Learning on Large Graphs. In *NeurIPS*.

[19] B. Karlsson. 2021. RenderDoc. https://renderdoc.org/docs/index.html.

[20] T. N. Kipf and M. Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *ICLR*.

[21] D. J. L. Lee, D. Tang, K. Agarwal, T. Boonmark, C. Chen, J. Kang, U. Mukhopadhyay, J. Song, M. Yong, M. A. Hearst, and A. G. Parameswaran. 2021. Lux: Always-on Visualization Recommendations for Exploratory Dataframe Workflows. *PVLDB* 15, 3 (2021), 727–738.

[22] Y. Li, Y. Shen, W. Zhang, J. Jiang, Y. Li, B. Ding, J. Zhou, Z. Yang, W. Wu, C. Zhang, and B. Cui. 2021. VolcanoML: Speeding up End-to-End AutoML via Scalable Search Space Decomposition. *PVLDB* 14, 11 (2021), 2167–2176.

[23] S. Macke, A. G. Parameswaran, H. Gong, D. J. Lin Lee, D. Xin, and A. Head. 2021. Fine-Grained Lineage for Safer Notebook Interactions. *PVLDB* 14, 6 (2021), 1093–1101.

[24] E. Mansour, K. Srinivas, and K. Hose. 2021. Federated Data Science to Break Down Silos [Vision]. *SIGMOD Rec.* 50, 4 (2021), 16–22.

[25] A. Nair, A. Roy, and K. Meinke. 2020. funcGNN: A Graph Neural Network Approach to Program Similarity. In *ESEM*.

[26] C. A. Navarro, N. Hitschfeld-Kahler, and L. Mateu. 2014. A Survey on Parallel Computing and its Applications in Data-Parallel Problems Using GPU Architectures. *Communications in Computational Physics* 15, 2 (2014), 285–329.

[27] J. Paul, S. Lu, and B. He. 2021. Database Systems on GPUs. *Found. Trends Databases* 11, 1 (2021), 1–108.

[28] B. Peng, P. Fatourou, and T. Palpanas. 2021. SING: Sequence Indexing Using GPUs. In *ICDE*.

[29] D. Petersohn, W. W. Ma, D. J. L. Lee, S. Macke, D. Xin, X. Mo, J. Gonzalez, J. M. Hellerstein, A. D. Joseph, and A. G. Parameswaran. 2020. Towards Scalable Dataframe Systems. *PVLDB* 13, 11 (2020), 2033–2046.

[30] X. Ren and M. Lis. 2021. CHOPIN: Scalable Graphics Rendering in Multi-GPU Systems via Parallel Image Composition. In *HPCA*.

[31] C. Root and T. Mostak. 2016. MapD: a GPU-powered Big Data Analytics and Visualization Platform. In *SIGGRAPH*.

[32] C. Rossant, N. P. Rougier, J. Comba, and K. P. Gaither. 2021. High-Performance Interactive Scientific Visualization With Datoviz via the Vulkan Low-Level GPU API. *Comput. Sci. Eng.* 23, 4 (2021), 85–90.

[33] K. Simonyan, A. Vedaldi, and A. Zisserman. 2014. Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps. In *ICLR*.

[34] A. Suprem, J. Arulraj, C. Pu, and J. E. Ferreira. 2020. ODIN: Automated Drift Detection and Recovery in Video Analytics. *PVLDB* 13, 11 (2020), 2453–2465.

[35] 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.

[36] B. Welton and B. P. Miller. 2019. Diogenes: Looking for an Honest CPU/GPU Performance Measurement Tool. In *SC*.

[37] D. Xin, H. Miao, A. G. Parameswaran, and N. Polyzotis. 2021. Production Machine Learning Pipelines: Empirical Analysis and Optimization Opportunities. In *SIGMOD*.

[38] X. Yan and J. Han. 2002. gSpan: Graph-Based Substructure Pattern Mining. In *ICDM*.

[39] Z. Yao, R. Chen, B. Zang, and H. Chen. 2022. Wukong+G: Fast and Concurrent RDF Query Processing Using RDMA-Assisted GPU Graph Exploration. *IEEE Trans. Parallel Distributed Syst.* 33, 7 (2022), 1619–1635.

[40] E. T. Zacharatou, H. Doraiswamy, A. Ailamaki, C. T. Silva, and J. Freire. 2017. GPU Rasterization for Real-Time Spatial Aggregation over Arbitrary Polygons. *PVLDB* 11, 3 (2017), 352–365.

[41] P. Zandbergen. 2022. Machine Code and High-level Languages: Using Interpreters and Compilers. https://study.com/academy/lesson/machine-code-and-high-level-languages-using-interpreters-and-compilers.html/.

[42] L. Zhao, A. Khan, and R. Luo. 2022. ShaderNet Code and Datasets. https://lzlz15.github.io/D_E_M_O/.

[43] 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 GPU-accelerated Applications with HPCToolkit. *Parallel Comput.* 108 (2021), 102837.