

# *e*-CLAS: Effective GPUDirect I/O Classification Scheme

Jungmyoung Kim and Jaechun No(✉)

Department of Computer Science and Engineering, Sejong University,
209 Neungdong-ro, Gwangjin-gu, Seoul, Korea
jano@sejong.ac.kr

**Abstract.** GPUs are widely used in various computational areas, due to their massive multithreading platform capable of high-level parallelism. Especially, to speedup I/O processing, GPUDirect I/O has been developed to enable direct data access between GPU memory and either NVMe or NVMe-oF. However, GPUDirect I/O does not always enhance the I/O performance of applications, because of reducing the benefit of page cache. In this paper, we present an effective I/O scheme, called *e*-CLAS, that can categorize I/Os through machine learning algorithms, to maximize the benefit of GPUDirect I/O while managing NVMe device storage at cost-effective way. The performance evaluation shows that our scheme effectively works to increase I/O performance, by appropriately classifying I/Os generated from applications.

**Keywords:** GPUDirect · I/O classification · NVMe · Direct I/O · machine-learning

## 1 Introduction

As GPU technology has been rapidly developed, several areas requiring high computation power, such as deep learning or HPC applications, increasingly facilitate GPUs due to their potential of high processing capability through massive multithreading and parallel executions [1–5, 8]. In order to leverage GPUs in user applications, users should link with API provided at GPU runtime library (e.g. CUDA).

In such an environment, to execute I/O processing, the necessary data should be transferred from storage to host memory and then moved to GPU memory by using the runtime library such as CUDA. Such a data transmission can restrict I/O processing throughput by going through multiple layers or external device (e.g. PCI-e), incurring the significant I/O performance degradation. GPU Direct I/O [6, 7] has been developed to overcome such a drawback in I/O path, by providing the direct data access between GPU memory and either NVMe (Non-Volatile Memory access) or NVMe-oF.

However, providing the direct I/O path in GPUDirect would not always enhance I/O performance [5] because of host kernel bypass capabilities, reducing the benefit of page cache such as data prefetching. There is a need to differentiate the I/O path

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024
O. Gervasi et al. (Eds.): ICCSA 2024, LNCS 14814, pp. 3–16, 2024.
https://doi.org/10.1007/978-3-031-64608-9_1

required in applications between GPUDirect Scheme (GDS interchangeably) and CPU-GPU Scheme (GPU or GPU I/O interchangeably), according to I/O characteristics such as data size or access pattern.

In this paper, we propose a machine-learning based I/O classification scheme, called *e*-CLAS(***e***ffective GPUDirect I/O **CLAS**sification Scheme), capable of categorizing I/Os through machine learning algorithms, to maximize the benefit of GPUDirect I/O while managing NVMe device storage at cost-effective way.

In this paper, we first compare the I/O performance of GDS and GPU (typical GPU I/O) by various benchmark tools and leverage the comparison results to organize ML-based classes, together with application I/O parameters and system status information including file size, I/O access pattern, and memory usages of CPU, GPU and memory.

To choose the appropriate data for GDS, *e*-CLAS extracts features affecting I/O performance of applications facilitating GPUs for computations and I/Os, and then classifies the proper data for GDS to separately store them to NVMe.

As a result, I/O operations on those selected data would utilize GDS capability by allowing the direct access between NVMe where the desired data are stored and GPU memory, using cuCIM library. In *e*-CLAS, the ML-based models built for categorizing I/Os have been implemented by using multiple ML algorithms including Support vector machine, Random Forest and Logistic regression. Among those models, the one enabling the best performance can automatically be selected, to exploit I/O speedup.

The rest of paper is organized as follows: In Sect. 2, we present the performance comparison between GDS and GPU to describe our motivations. In Sect. 3, we represent the overall structure of *e*-CLAS, and demonstrate its performance evaluation in Sect. 4. We conclude our paper in Sect. 5.

## 2 Motivation

In Figs. 1 and 2, we analyzed the I/O bandwidth of GDS on the local environments, while comparing it with GPU (CPU-GPU Scheme). In these figures, GDS creates the direct I/O path between the local NVMe device and the GPU memory. Such an I/O path is activated by going through either the network adaptor or the storage-based DMA engine to transfer data to GPU memory, while avoiding CPU bounce buffer.

The experimental platform used for the evaluation includes a 3.10 GHz 6-core Intel Xeon CPU i5-10500, NVIDIA RTX A4000 GPU, four 8 GB of Samsung DDR4 (total 32 GB), a 1 TB of Western Digital WD5000AZLX, and a 1 TB of KLEVV CRAS c710 NVMe. We used Ubuntu 20.04 with kernel 5.4.0 for our evaluation.

The evaluation results of Figs. 1 and 2 were obtained by using gdsio and fio, respectively. The file sizes are varied from 1 GB to 16 GB, while changing I/O sizes between 4 KB and 16 KB. As the I/O sizes increase, the performance gap between GDS and GPU becomes significant due to multiple data paths to reach GPU memory for computations, except for results with 16 GB of file size.

In Fig. 1, with 4 KB of file size, the I/O throughput of GPU shows about 15% of I/O degradation compared to that of GDS because of multiple I/O paths until the necessary data reaches GPU memory. However, such a difference is smaller than that with the large I/O size because the effect of bypassing cache do not help enhance GDS performance.

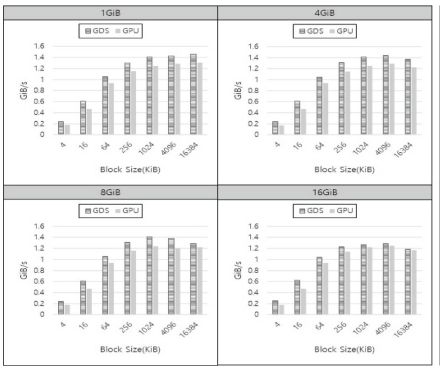


**Fig. 1.** gdsio read performance comparison

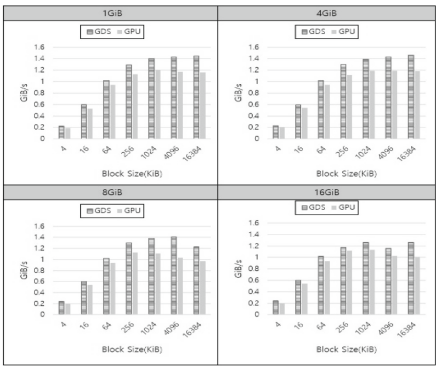


**Fig. 2.** fio read performance

With large I/O sizes, the beneficial of GDS becomes apparent, except for results with 16 GB of file size where I/O access latency to storage is large compared to that of other file sizes. We can observe the similar I/O execution behavior in Fig. 2, measured with *fio* benchmark.

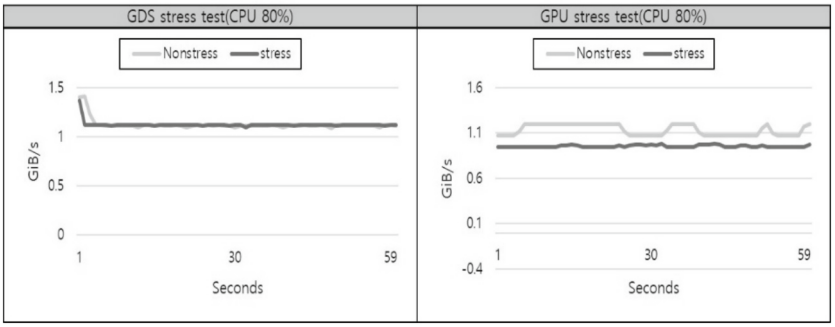


**Fig. 3.** Performance comparison using stress tool

Figure 3 illustrates how the existing I/O stress impacts I/O performance of GDS and GPU. In this comparison, we used Linux stress tool. In the figure, we can notice that the existing I/O stress would cause more performance variation with the typical GPU I/O than with GDS, since GPU I/O going through more I/O layers can be significantly influenced by the I/O stress. In GDS, fewer I/O layers are involved in I/O operations, resulting in more consistent performance compared to GPU I/O.

To collect appropriate performance results for implementing *e*-CLAS, we performed a lot of performance comparisons, although not all of them can be presented in this paper due to page limitations. Based on the evaluation, we attempt to achieve the following main objectives:

1) We try to categorize I/Os based on various application execution parameters, such as file and I/O sizes, access pattern, and current resource statuses including CPU, GPU and memory.
2) To choose the appropriate I/O scheme for applications, we utilize various machine learning (ML) algorithms. The ML algorithms being used for *e*-CLAS can be easily integrated with the system, for both training and evaluating the associated I/O models.
3) *e*-CLAS can effectively manage the NVMe storage capacity, by transferring only the data needed for GDS into it. Also, it stores the I/O categorization result for applications in the repository, to avoid redundant model evaluation or data transmission for further processing.

As the potential of GPUs has become evident in massive computational areas, extensive research has been conducted in various ways. For instance, Silberstein et al. [2] attempted to integrate GPU code to host's filesystem through POSIX-like API, while extending the host buffer cache to GPU memory, to retain GPU advantages such as massive parallelism. In the scheme, the GPUfs library caching filedata in both CPU and GPU memory is linked with user GPU code, to enable data accesses to the host filesystem.

Zhang [9] tried to provide a way of directly connecting GPU memory to SSD devices by mapping physically shared system memory blocks. By doing this way, their scheme can mitigate the overhead of data movement while reducing user/kernel switching.

Vesely et al. [10] tried to support standard POSIX semantics for GPU, instead of using special system calls. SPIN [5] proposed a way of integrating GPUDirect into file I/O layer, to leverage the benefit of page cache, such as prefetching, while maintaining POSIX file consistency.

Also, there are several GPU works for optimizing irregular memory accesses [11–14]. For example, Segura et al. [11] proposed IRU that can reorder irregular data accessed by threads, to improve memory coalescing in graph-based GPGPU applications. Li et al. [12] worked on optimizing irregular memory accesses by providing both fine-grained and coarse-grained L1 cache line management, without requiring additional memory structure for tag storage. Guo et al. [13] also tried to resolve the problem of mismatching between the size of accessed data and the wide GPU cache line, by utilizing run-time memory access profiling.

Chen et al. [14] argued that the current GPU cache might not enhance application performances retaining a lot of multiple irregular access patterns, due to its aim for the speedup architecture through multithread-based structure. They proposed GPGPU's cache policy that can control the number of active warps at run-time, to avoid cache contention.

ActivePointers [15] was proposed to eliminate CPU interference with active pointer in handling both page fault and GPU address translation occurring mmap system calls. It provides a pointer-based interface to directly access to storages from GPU. Also, GaiaGPU [16] was developed to combine CPU and GPU memories to extend OS page cache, while integrating it with OS file I/O layer. It employs mmap system calls to extend page cache space into GPU memory while addressing false sharing overhead through a lazy release consistency model.

# 3 Implementation Details

## 3.1 System Architecture

The main objective of *e*-CLAS is to categorize applications that can accelerate I/O performance through GPUDirect I/O. To achieve this, we classify the appropriate datasets generated by those applications using machine learning and store the data in advance on NVMe device storage, allowing for direct access when possible.

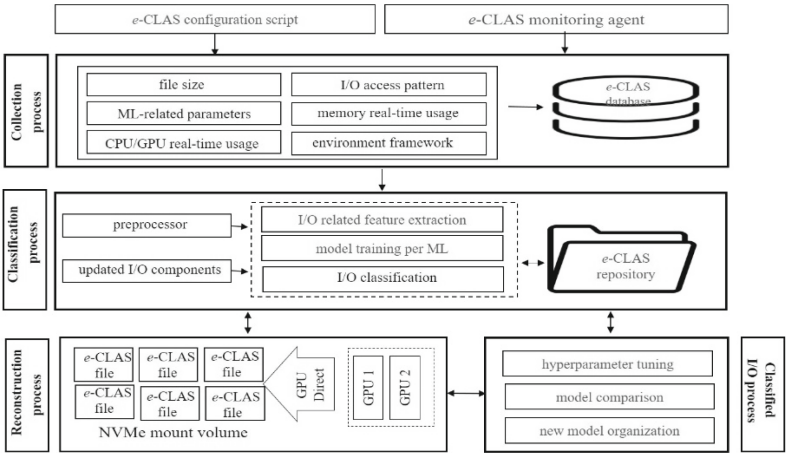


**Fig. 4.** *e*-CLAS Structure

Figure 4 illustrates *e*-CLAS structure. The *e*-CLAS is composed of four processes: collection process, classification process, reconstruction process, and classified I/O process. The collection process obtains the necessary information to categorize the I/O scheme enhancing application performance, such as the dataset for evaluating model training, the system environment parameters including the real-time states of host memory, CPU, and GPUs. The dataset for evaluating models would be continuously updated as new applications are executed on the *e*-CLAS.

The classification process extracts I/O characteristics (features) needed for categorizing the proper I/O scheme through machine learning (ML). *E*-CLAS can organize diverse I/O models to search for the proper applications whose dataset is appropriate for performing GPUDirect I/O. The ML algorithm necessary for organizing I/O models is either specified in the collection process as a part of *e*-CLAS configuration, or automatically selected by comparing the performance comparison ratio during the model training in the classification process. The organized models are stored in *e*-CLAS repository to be used for further processes.

In the reconstruction process, *e*-CLAS updates the existing models by optimizing steps, such as hyperparameter tuning. During this process, newly recognized, effective I/O characteristics are used to update and reorganize *e*-CLAS models, which are then stored for further use.

Based on the output of the reconstruction process, data sets not residing in NVMe device storage are copied to NVMe storage during the classified I/O process, enabling the utilization of GPUDirect in I/O steps available in the corresponding applications. As a result, if the applications running on the host generate the appropriate datasets for GPUDirect, the datasets would be directly accessed between NVMe storage and GPU memory.

To avoid redundant steps in organizing and analyzing the required I/O models for applications, the previously obtained performance comparison ratio for each application is stored in the *e*-CLAS repository for reuse in subsequent executions.

### 3.2 *e*-CLAS Collection Process

Table 1 shows an example of datasets used for training *e*-CLAS I/O models. Currently, we collect the dataset by executing *gdsio* and *fio* benchmarks. Because the system environment can significantly impact I/O performance, we generate datasets by introducing various system statuses, while changing the system environment in the collection process.

**Table 1.** An example of dataset for training *e*-CLAS I/O model

| | file size | I/O access pattern | GPU usage | memory usage | CPU usage | framework | I/O class |
|---|---|---|---|---|---|---|---|
| 1 | 0.2 | 1 | 0 | 16 | 0 | 0 | 0 |
| 3 | 4.0 | 1 | 32 | 42 | 85 | 0 | 0 |
| 4 | 2.9 | 1 | 39 | 17 | 1 | 1 | 0 |
| 2214 | 3.7 | 1 | 42 | 38 | 84 | 0 | 1 |

```
file_size   = os.path.getsize(name)
steam1      = os.popen(“free| grep ^Mem | awk '{printf $3*100/$2}'")
stream2     = os.popen("nvidia-smi | grep % | awk '{print $13}' | cut -d '%' -f1")
dockerframe = "docker“
randpat     = "randomread“
if pattern == randpat:
      A_new = np.array([[file_size3, 0, 1, inout1, inout, inout2]])
……
```

**Fig. 5.** An example of *e*-CLAS configuration

Figure 5 shows the *e*-CLAS configuration script used for categorizing the proper I/Os. The initial I/O parameters for determining resource statuses at the time of collection would be specified in the configuration file. After identifying the I/O parameters used for analyzing associated I/O performance, the monitoring agent updates the real-time statuses of the selected system component, to provide a more accurate performance comparison ratio for the classification.

The I/O parameters can be easily added to the configuration file, without involving the model reorganization. The current parameters for choosing the appropriate I/O scheme include file size, machine learning algorithm for model training, I/O access pattern, and usages of GPU, CPU and memory.

### 3.3 I/O Categorization

The classification process selects the appropriate features for performing *e*-CLAS and organizes the necessary models for producing the accurate performance comparison ratio. Once the classification module obtains the datasets from the collection module, it preprocesses the dataset in such a way that it takes the abnormal values away to prevent I/O errors. The preprocessed I/O dataset is used for categorizing I/O schemes through model training.

We use Random Forest (RF), Support Vector Machine (SVM) and Logistic Regression (LR) to organize *e*-CLAS models. However, user can specify more ML algorithms to provide the higher performance ratio, by specifying them in the configuration file. Since different algorithms can lead to different performance ratios when applied to the same dataset, the choice of ML algorithm depends on various factors such as the nature of the dataset, computational resources where applications execute, or performance requirements. *E*-CLAS attempts to consider those factors by flexibly providing them to the system and to compare their performance ratios, to select the most suitable one for application executions. The trained models and the performance comparison ratios are stored in the *e*-CLAS repository for further uses.

Table 2 shows an example of the performance comparison ratio obtained from the *e*-CLAS classification process. In this example, the evaluation values including precision, recall, and F1-score are used, and also SVM shows the highest ratio.

**Table 2.** An example of performance comparison ratio

| | | precision | recall | F1-score | support |
|---|---|---|---|---|---|
| RF | I/O | 0.99 | 0.94 | 0.97 | 533 |
| | accuracy | | | 0.94 | 555 |
| | weighted avg. | 0.97 | 0.94 | 0.95 | 555 |
| SVM | I/O | 0.99 | 0.99 | 0.99 | 533 |
| | accuracy | | | 0.98 | 555 |
| | weighted avg. | 0.98 | 0.98 | 0.98 | 555 |
| LR | I/O | 0.99 | 0.92 | 0.96 | 533 |
| | accuracy | | | 0.92 | 555 |
| | weighted avg. | 0.97 | 0.92 | 0.94 | 555 |

### 3.4 I/O Reconstruction Process

The reconstruction process is to update *e*-CLAS models by applying either a new ML algorithm, or accepting additional I/O parameters from the configuration script. In this case, *e*-CLAS extracts new features affecting the I/O performance through hyperparameter tuning and organizes new models to update the performance. Figure 6 shows an example of *e*-CLAS reconstruction process to update the models.

At the initial step, *e*-CLAS mounts its volume to store the selected datasets through the classified scheme. If the dataset is already available outside *e*-CLAS storage, the associated dataset is duplicated into *e*-CLAS directory in advance, to make them available for *e*-CLAS process.

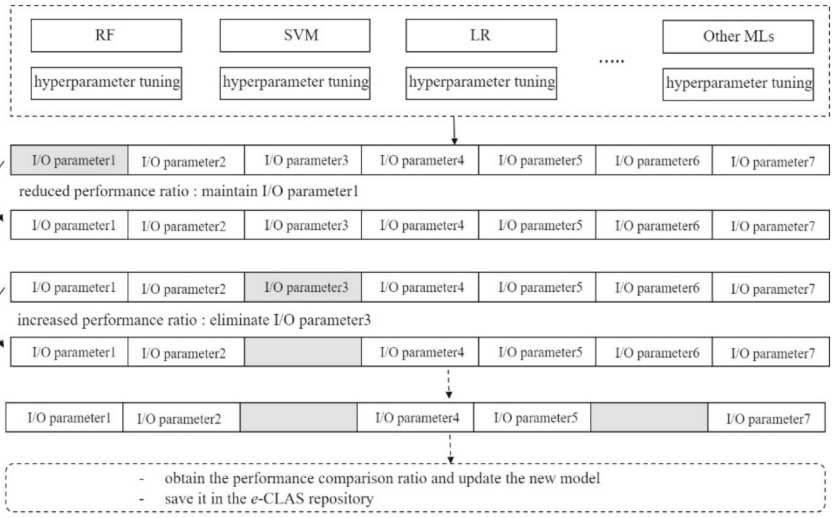


**Fig. 6.** *e*-CLAS reconstruction process

In order not to repeatedly execute the classification process of *e*-CLAS, the previously measured performance comparison ratio of the application is stored in the repository, and reused for the next execution, unless user triggers to update the new result. The procedures for categorizing I/O schemes are as follows:

1. Preprocess the datasets by eliminating parameter errors. If multiple ML candidates are determined in the *e*-CLAS configuration, select the better candidate by comparing their registered performance ratio stored in the repository.
2. Execute the optimization steps including hyperparameter tuning by updating I/O parameters and then choose the effective features significantly influencing the performance ratio enhancement. Perform steps repeatedly until effective I/O parameters are chosen.
3. Train and evaluate the *e*-CLAS models by leveraging the selected features, and store the trained models per ML in the repository.

4. Mount *e*-CLAS volume to perform the classified I/O process, by leveraging the models residing in the repository and the real-time system environment provided by the system agent. When applications begin, access the repository to check the proper I/O scheme and perform the selected I/O scheme. The real-time application and environment data are updated in the database to be used for the further model training and analysis.

Table 3 shows an example of I/O parameters used for hyperparameter tuning and model training in the *e*-CLAS classification process.

**Table 3.** I/O parameters used in *e*-CLAS

| Attributes | Parameters |
|---|---|
| Hyperparameter tuning | GridSearchCV |
| ML algorithm | Support vector machine(SVM)<br>Random forest(RF)<br>Logistic regression(LR) |
| RF parameter | n_estimators,<br>max_features,<br>max_depths |
| SVM parameter | C, gamma |
| LR parameter | max_iter, C |

## 4 Performance Evaluation

Figures 7 and 8 are the performance comparisons of *e*-CLAS and GPU I/O (CPU-GPU I/O), while reading 1 GB of data to GPU local memory. We measured the performance on the local environment in Fig. 4 and on the docker environment in Fig. 8.

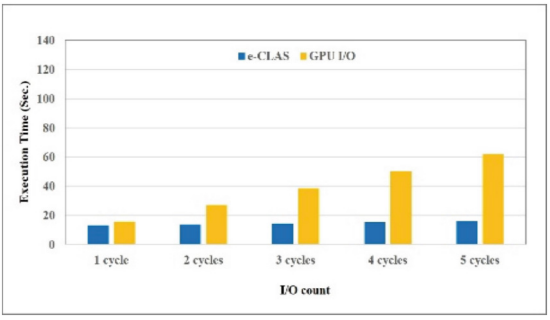


**Fig. 7.** Time comparison of seq. read operation on the local environment

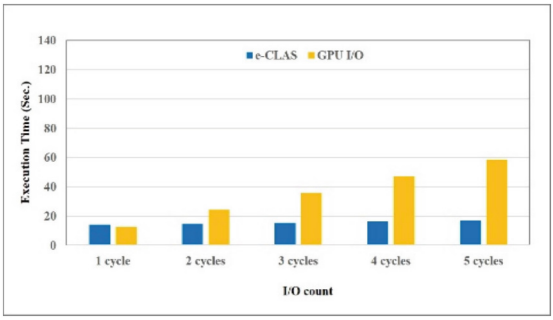


**Fig. 8.** Time comparison of seq. read operation on the docker environment

On the local environment, reading the file using *e*-CLAS takes about 13 s. and the time for reading using GPU I/O takes about 16 s. After the first step of reading data, *e*-CLAS approximately adds 0.75 s. for each read operation, while GPU I/O adding about 11 s. The reason for generating more I/O overhead in the existing GPU I/O is that every I/O step of GPU I/O needs to retrieve files from the host storage to CPU and to transfer those files to GPU memory for further computation.

Whereas in *e*-CLAS, although the necessary files should be copied to NVMe device storage at the first step, those files can be accessed from NVMe to GPU memory directly in the next I/O steps. Therefore, *e*-CLAS produces less I/O overhead in accessing the desirable files from NVMe to GPU, to utilize those files in the GPU computation.

Also, in GPU I/O, while I/O cycles increase the performance gap between 1 cycle and 5 cycles becomes large, nearly tripling in magnitude. On the other hand, the gap of *e*-CLAS within the same cycle is only about 22%. This suggests that GPU I/O is more affected by the system status during the performance measurement.

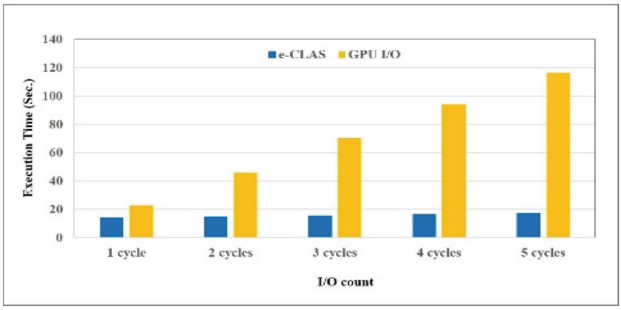


**Fig. 9.** Time comparison for random read operation on the local environment

Figures 9 and 10 show the random read performance comparison of *e*-CLAS and GPU I/O, by measuring *gdsio* benchmark on the local (Fig. 9) and the docker (Fig. 10) environments, respectively. Like in the previous experiments in Figs. 7 and 8, the performance of *e*-CLAS shows the better throughput compared to that of GPU I/O. Furthermore, the

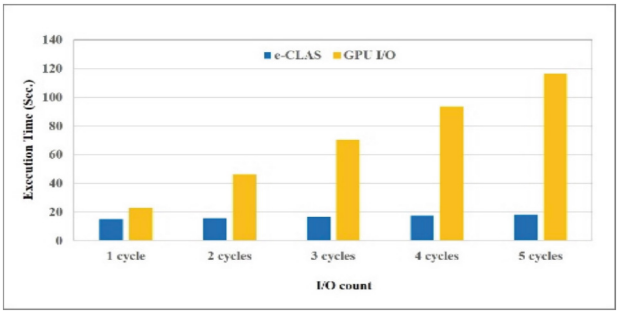


**Fig. 10.** Time comparison for random read operation on the docker environment

performance difference between the two schemes is more significant than that in the previous experiment, primarily due to the random read access pattern.

Even in the random access pattern, *e*-CLAS shows minimal differences as the number of I/O cycles increases, whether the performance was measured on the local environment or on the docker environment. However, with GPU I/O, the performance gap between 1 cycle and 5 cycles is nearly four times larger, even exceeding the gap shown in Fig. 7, due to the increased influence of system statuses.

In Fig. 11, we measured the throughput of cuCIM read operation, because we used cuCIM library for the performance comparison. In this evaluation, we vary the size of files from 1GiB to 4GiB. As can be seen, the read bandwidth in *e*-CLAS is higher than that of GPU I/O, although the performance gap is smaller than in the previous experiment.

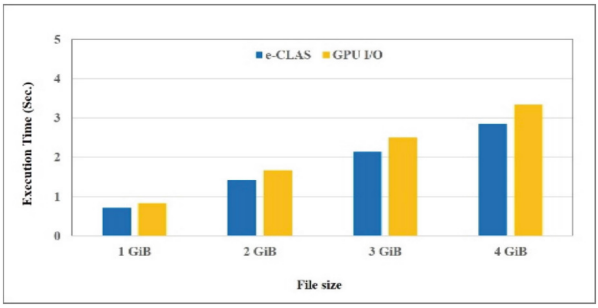


**Fig. 11.** cuCIM read performance

In Fig. 12, we compared the F1-score of ML algorithms used in *e*-CLAS, including RF, SVM, and LR. As can be seen, the SVM shows the best accuracy in organizing *e*-CLAS models. Even though, we set SVM models as default in the current implementation, users can easily switch to the other algorithm producing the better performance. Also, the *e*-CLAS can automatically detect the better models once their performance is registered in the repository.

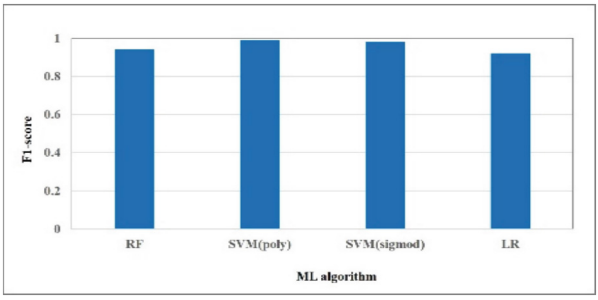


**Fig. 12.** F1-score of ML algorithms used in *e*-CLAS

Figures 13 and 14 are the comparison where two I/O schemes are installed on the remote server where clients access the desired data through NFS server. In the figures, Figs. 13 and 14 are the throughput of sequential read operations and random read operations, respectively. In this experiment, we can notice that even with the network overhead produced by NFS protocol, two I/O schemes generate the similar behavior, leading to the larger performance gap in GPI I/O as I/O counts increase.

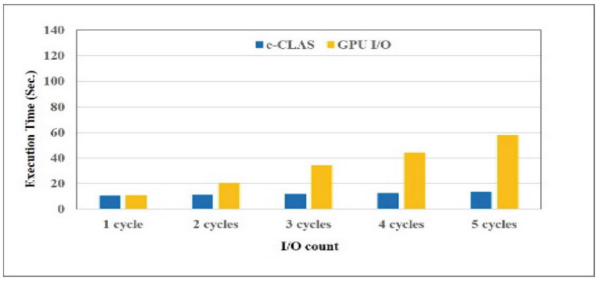


**Fig. 13.** Time comparison for seq. read operation where two I/O schemes are installed on the NFS server.

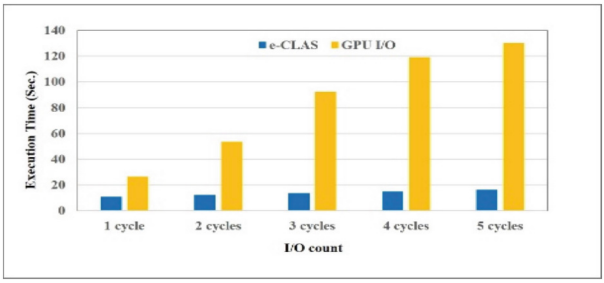


**Fig. 14.** Time comparison for random read operation where two I/O schemes are installed on the NFS server.

## 5 Conclusion

In this paper, we presented a machine-learning based I/O classification scheme, called *e*-CLAS, that categorizes I/Os using machine learning algorithms, to optimize GPUDirect I/O scheme (GDS) while efficiently managing NVMe device storage. To select the suitable data for each I/O scheme, *e*-CLAS extracts performance-affecting features from applications that facilitate GPUs for computations and I/Os. It then classifies the relevant data for GDS to separately store them to NVMe. We compared our scheme to the typical GPU I/O and demonstrated that *e*-CLAS can improve the I/O throughput on a large scale. As a future work, we would verify our scheme with more benchmarks in various ways to prove its effectiveness.

**Acknowledgements.** This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea Government (MSIT) (No. 2022R1A2C1004156).

## References

1. Ravikumar, K., Appelhans, D., Yeung, P.K.: GPU acceleration of extreme scale pseudo-spectral simulations of turbulence using asynchronism. In: Proceedings of the International Conference for High Performance Computing Networking, Storage and Analysis (SC 2019), Denver (2019)
2. Silberstein, M., Ford, B., Keidar, I., Witchel, E.: GPUfs: integrating a file system with GPUs. ACM Trans. Comput. Syst. **1**(1) (2013)
3. Hong, C., Spence, I., Nikolopoulos, D.: FairGV: fair and fast GPU virtualization. IEEE Trans. Parallel Distrib. Syst. **28**(12) (2017)
4. Suzuki, Y., Kato, S., Yamada, H., Kono, K.: GPUvm: GPU virtualization at the hypervisor. IEEE Trans. Comput. **65**(9) (2016)
5. Bergman, S., Brokhman, T., Cohen, T., Silberstein M.: SPIN: seamless operating system integration of peer-to-peer DMA between SSDs and GPUs. In: USENIX Annual Technical Conference (ATC 2017), Santa Clara (2017)
6. https://doc.nvidia.com/gpudirect-storage/overview-guide/index.html
7. Bayati, M., Leeser, M., Mi, N.: Exploiting GPU direct access to non-volatile memory to accelerate big data processing. In: IEEE Conference on High Performance Extreme Computing (HPEC) (2020)
8. Eduardo, V., Bona, L., Zola, W.: Speculative encryption on GPU applied to cryptographic file systems. In: 17th USENIX Conference on File and Storage Technologies (FAST 2019), Boston (2019)
9. Zhang, J., Donofrio, D., Shalf, J., Kandemir, M., Jung, M.: NVMMU: a non-volatile memory management unit for heterogeneous GPU-SSD architecture. In: PACT, pp. 13–24 (2015)
10. Vesely, J., Basu, A., Bhattacharjee, A., Loh, G., Oskin, M., Reinhardt, S.: Generic system calls for GPUs. In: ACM/IEEE 45th Annual International Symposium on Computer Architecture (2018)
11. Segura, A., Arnau, J., Gonzalez, A.: Irregular accesses reorder unit: improving GPGPU memory coalescing for graph-based workloads. J. Supercomput. **79**, 762–787 (2022)
12. Li, B., Wei, J., Sun, J., Annavaram, M., Kim, N.: An efficient GPU cache architecture for applications with irregular memory access patterns. ACM Trans. Architect. Code Optimiz. **16**(3) (2019)

13. Guo, H., Huang, L., Lu, Y., Ma, S., Wang, Z.: DyCache: dynamic multi-grain cache management for irregular memory accesses on GPU. IEEE Access **6**, 38881–38891 (2018)
14. Chen, X., Chang, L.-W., Rodrigues, C., Lv, J., Wang, Z., Hwu, W.: Adaptive cache management for energy-efficient GPU computing. In: IEEE/ACM International Symposium on Microarchitecture (MICRO) (2014)
15. Shahar, S., Bergman, S., Siberstein, M.: ActivePointers: a case for software address translation on GPUs. In: ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA) (2016)
16. Gu, J., Song, S., Li, Y., Luo, H.: Gaia GPU: sharing GPUs in container clouds. In: IEEE International Conference on Big Data and Cloud Computing (2018)