https://doi.org/10.2352/EI.2023.35.11.HPCI-243
© 2023, Society for Imaging Science and Technology

Towards Real-time Formula Driven Dataset Feed for Large Scale Deep Learning Training

Edgar Josafat Martinez-Noriega and Rio Yokota; National Institute of Advanced Industrial Science and Technology and Tokyo Institute of Technology; Japan

Abstract

Recently, a new deep learning architecture, the Vision Transformer, has emerged as the new standard for classification tasks, overtaking the conventional Convolutional Neural Network (CNN) models. However, these state-of-the-art models require large amounts of data, typically over 100 million images, to achieve optimal performance through transfer learning. This requirement is met by using proprietary datasets like JFT-300M or 3B, which are not publicly available. To overcome these challenges and address privacy concerns, Formula-Driven Supervised Learning (FDSL) has been introduced. FDSL trains deep learning models using synthetic images generated from mathematical formulas, such as Fractals and Radial Contour images. The main objective of this approach is to reduce the Input/Output (I/O) bottleneck that occurs during training with large datasets. Our implementation of FDSL generates instances in real-time during training, and uses a custom data loader based on EGL (Native Platform Graphics Interface) for fast rendering via shaders. The evaluation of our custom data loader on the FractalDB-100k dataset comprising 100 million images revealed a loading time that is three times faster compared to the PyTorch Vision loader.

Introduction

The field of deep learning has witnessed a major breakthrough in recent years with the emergence of the Vision Transformer as the new standard for tackling classification tasks. Unlike its predecessor, the Convolutional Neural Network (CNN), which relies on convolution operations, the Vision Transformer places its focus on the self-attention mechanism. This innovative approach has proven to be highly effective, as evidenced by the success of natural language processing models such as GPT [1] and BERT [2]. The training of Vision Transformer models faces a major challenge, the requirement of a massive amount of data. To achieve state-of-the-art performance using transfer learning, these models typically need over 100 million images [3, 4]. To fulfill this requirement, large datasets such as JFT-300M or 3B are often utilized [5]. However, the issue with these datasets is that they are typically proprietary and not accessible to the public. To address this issue, as well as concerns about privacy, a new approach called Formula-Driven Supervised Learning (FDSL) has been proposed [6]. Unlike traditional approaches, which rely on large proprietary datasets, this innovative method utilizes synthetic images generated from mathematical formulas, such as fractals and radial contour images, to train Vision Transformer models [7]. The performance of this method has been compared to real image datasets and found to be effective, but these results are based on smaller datasets containing only 10 million images [8]. To achieve state-of-the-art performance using Vision Transformer models, a dataset of the scale of JFT-300M is necessary. However, training on such a large dataset with a high-performance computing system like the ABCI supercomputer [9] presents several challenges, including efficient management of high I/O traffic. To tackle the challenges of training large datasets, a widely-used technique is Data Distribution, which employs multiple GPUs. This method involves each GPU holding a copy of the model and training on a different subset of the data over multiple iterations. Popular deep learning frameworks like PyTorch Vision [11] offer a data loader that evenly distributes the data among all GPUs. The loader retrieves the raw image files from the file system, however, when training on massive datasets such as ImageNet-1k (1.2 million images) or ImageNet-21k (14.19 million images), the heavy burden on the file system can become problematic. To handle the increased load, hundreds of nodes with multiple GPUs may be used in the data distribution parallel approach. Moreover, GPUs were originally engineered to accelerate the production of high-speed 3D graphics. Utilizing native graphics APIs, such as OpenGL or DirectX, significantly boosts the rendering process. The original FractalDB [6] was created using Python routines, resulting in a rendering time that took several hours to generate the entire dataset. By harnessing the power of GPUs, we can streamline the process of generating the entire dataset and reduce the load on the network, by utilizing the GPU to render each image.

The goal of our work is to overcome Input/Output (I/O) bottlenecks during training on large Formula-Driven datasets. To achieve this, we propose a new approach that generates instances dynamically during training, reducing the I/O burden. Each worker or GPU has the necessary information to render individual instances of the dataset, eliminating the need to retrieve data from disk. By generating instances on the fly, we can reduce the amount of data loaded into memory at any given time, which improves training efficiency.

Our proposed method of utilizing GPUs for searching and generating Fractal images resulted in a much faster dataset creation process, up to 185 times faster than the previous CPU implementation. This approach enabled us to create larger versions of FractalDB, with over 300 million images, comparable in size to JFT-300M. Additionally, our loader outperformed the PyTorch loader by over 3 times in experiments involving full training on large datasets.

Related Work

Large datasets have been proposed to train Vision Transformers such as Instagram-3.5B [16], YFCC from Yahoo [17], JFT-300M/3B from Google[5], and LION-5B[18]. However, most of these datasets are close to those research groups and are

not open to the public, only LION-5B is distributed under a permissive license. Nevertheless, the use of these large datasets are not exempt from ethical concerns, such as societal biases, privacy issues, and copyright violations. These problems are of great concern and can result in models trained on these datasets being unfair or violating privacy protection laws. The lack of curation in these datasets further exacerbates these ethical issues. Thus, FDSL has been proposed to overcome these issues. Several efforts have been made in FDSL field. For example, the research conducted by Nakashima *et al.* [8] that pre-training a Vision Transformer (ViT) model on FractalDB can lead to similar accuracy results on downstream tasks as a ViT pre-trained on ImageNet. Kataoka *et al.* [7] later expanded FractalDB to ExFractalDB and RCDB, which focus more on contours than textures. These datasets were used to pre-train ViT on ImageNet-21k and compare results with synthetic datasets ExFractalDB-21k and RCDB-21k, which have the same number of categories and images. On another hand, there have been several proposals to alleviate the burden of I/O on training deep learning models in a cluster environment. Dryden *et al.* [19] introduced NoPFS, a machine learning I/O middleware designed to solve the I/O bottleneck in a scalable, flexible, and user-friendly way. NoPFS employs clairvoyance, allowing it to accurately predict when and where a sample will be accessed based on the seed used to generate the random access pattern for training with Stochastic Gradient Descent (SGD). Nguyen *et al.* [20] suggested exploring the possibility of partitioning the dataset among multiple workers in deep learning workloads and performing a limited exchange of samples in each training epoch. They showed that, with proper adjustment, the validation accuracy achieved through global shuffling can still be maintained with partial distributed exchange. Aizman *et al.* [21] introduced AIStore, a scalable and user-friendly storage solution, and WebDataset, a storage format and library that adheres to industry standards and enables efficient access to massive datasets. Furthermore, Baradad *et al.* demonstrated the possibility of learning visual representations from synthetic images using the Deadleaves dataset. As well, they proposed training with a dataset of 21,000 programs, each generating a variety of synthetic images, where the programs are brief code snippets that can be easily altered and quickly run using OpenGL [22].

Method

This section provides an overview of FDSL methods, followed by a detailed description of the FractalDB dataset. We also explain the original implementation of the IFS routine for Fractal image creation and how we optimized it using a parallel approach on the GPU.

Formule Driven Supervised Learning (FDSL)

FDSL, or Formula-Driven Synthetic Learning, is a cutting-edge training methodology in the field of machine learning. In this approach, synthetic images and their corresponding labels are generated from mathematical formulas, offering a unique way to generate large, diverse, and dynamic datasets for training. Adopting this strategy comes with several advantages, including the elimination of ethical concerns. By training on synthetic datasets generated from mathematical formulas, we can sidestep issues such as societal biases and the handling of sensitive information, such as copyrights and personal data [10, 12, 13]. This not only enhances the ethical standards of the technology being developed but also provides a more secure and controlled environment for training machine learning models.

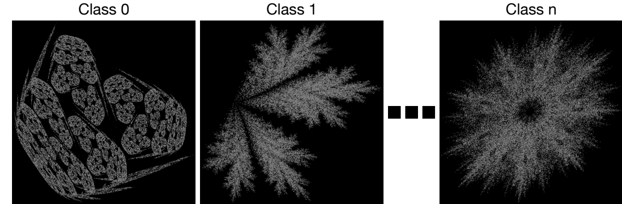

Figure 1. Intra-Class Fractal samples.

The synthetic images generated come in a diverse range of shapes and patterns, including polygons, geometric shapes, and fractals [6, 7, 8, 14]. The complexity and intricacy of these shapes can be fine-tuned through adjusting various parameters such as smoothness, orientation, texture, fill rate, and other factors that influence the final image. Labels can be assigned to these images based on a combination of any of these parameters, enabling the creation of a labeled dataset of any size without the need for manual annotation by human operators.

FractalDB

The original FractalDB is a collection of fractal images generated using the Iterated Function System (IFS) [15], consisting of 1,000 to 10,000 image-label pairs. Fractal geometry was selected as a method for generating the dataset because of its ability to render complex patterns and shapes for each unique set of parameters. More formally, we can define a complete metric space $\mathscr{X}$ by

$$\text{IFS} = \{\mathscr{X}; w_1, w_2, \cdots, w_N; p_1, p_2, \cdots, p_N\}, \quad (1)$$

where transformation functions are defined by $w_i : \mathscr{X} \to \mathscr{X}$, p_i are probabilities, and the number of transformations is defined by N. Thus, a fractal $S = \{x_t\}_{t=0}^{\infty} \in \mathscr{X}$ can be generated in the 2D Euclidean space $\mathscr{X} = \mathbb{R}^2$. This assumption is made that each transformation in practice is a type of affine transformation, characterized by six parameters $\theta_i = (a_i, b_i, c_i, d_i, e_i, f_i)$ for operations such as shifting and rotation:

$$w_i(x; \theta_i) = \begin{bmatrix} a_i & b_i \\ c_i & d_i \end{bmatrix} x + \begin{bmatrix} e_i \\ f_i \end{bmatrix}. \quad (2)$$

The parameters $(a_i, b_i, c_i, d_i, e_i, f_i, p_i)$ are randomly selected at first and adopted if the filling rate of the image pattern generated from the parameters surpasses a threshold, calculated as the number of fractal dot pixels divided by the total number of image pixels. The instances within each category are created extensively using three methods that consider the category's configurations to preserve its shape: slight adjustments to parameters, rotation, and patch drawing. A sample of the main shape for each class in FractalDB is shown in Figure 1.

Accelerating FractalDB Creation through GPU Rendering

The original FractalDB code is publicly available. This code is written in Python and it provides the ability to search the parameters described in 2. A sample of the main routine is shown in Listing 1. As we can observe, the IFS implementation loops over

Listing 1. *Original IFS Python routine to search and generate FractalDB.*

```python
def ifs(self,iter):
    rand = np.random.random(iter)
    s_func = self.s_func
    func = self.func
    p_x,p_y = self.p_x, self.p_y
    for i in range(iter-1):
        for j in range(len(s_func)):
            if rand[i] <= s_func[j]:
                n_x = p_x * func[j]["a"] +
                      p_y * func[j]["b"] +
                      func[j]["e"]
                n_y = p_x * func[j]["c"] +
                      p_y * func[j]["d"] +
                      func[j]["f"]
                break
        self.xs.append(n_x),self.ys.append(n_y)
        p_x = n_x
        p_y = n_y
```

many iterations (points) given the parameters for each fractal and its probability, computing the next point based on the previous one. The original settings contemplate a threshold set to 80%, and the number of points for the search to 100,000. After all the points are calculated, the rendering is performed using a basic perspective projection to a fixed image resolution. If the fixed rate is acceptable, the the parameters are save in csv files (descriptors for each category). Thus, we can generate the fractal images for each class utilizing different patches or single points. Nevertheless, the generation of these fractals are performed in a serial execution and the complete generation of FractalDB could take much time. For example, to generate FractaDB-1k using this approach, including 1,000 instances per class, total 1 Million images takes 2 days (5.4 images/sec) to complete even using ABCI supercomputer system.

IFS CUDA implementation

We implemented the IFS routine for fractal search and generation using CUDA. A complete implementation is shown in Listing 2. We utilize as many threads as possible to iterate over many points. From the original routine shown in Listing 1, we compute the outer loop using the *index* variable which is an specific ID for each thread in the kernel. We also transfer the fractal parameters to shared memory. This provides faster access and computing when each thread is calculating the next point. As well, we utilize cuRAND library for efficient generation and high-quality pseudo-random numbers. This is utilized when calculating the probability for each point. Finally, we avoid the movement of the data back to the CPU, more concretely we keep the position of each point inside of the GPU memory that will be share later to OpenGL for rendering purposes.

CUDA-OpenGL interoperability

To minimize the unnecessary communication overhead, CUDA offers graphics interoperability to share memory resources between CUDA and the rendering context such as OpenGL and Direct3D. This approach eliminates the need to transfer data back and forth between the CPU and GPU, resulting in improved performance in both computation and rendering. However, ensuring the congruence between CUDA and OpenGL memory space is important for proper implementation of this feature.

Headless rendering

We employ high-speed 3D rendering APIs such as OpenGL or Direct3D to enable quick and sophisticated computer graphics processing by the GPU. Communication with these APIs from and to the GPU involves an intermediate stage that is the windowing system. Different OSs have varying implementations of windowing systems, for example, X11 or Wayland for Linux-based systems, and Desktop Windows Manager (DWM) for Windows. Therefore, we communicate via these windowing systems by using specialized libraries such as GLFW. Subsequently, we display or project the rendered image onto a display or screen. Typically, a physical display is linked to the GPU when employing standard rendering methods. However, when performing server-side rendering or when operating on high-performance clusters, headless rendering without a display is preferred. EGL (Native Platform Graphics Interface) is utilized to access OpenGL APIs in a headless manner. An illustration of both approaches is presented in Figure 2.

Listing 2. *Paralle IFS CUDA implementation using shared memory.*

```cuda
__global__ void ifs(float2* d_poss, int numPoints,
  mapping *d_mappings, int numMappings){
  int index = blockIdx.x * blockDim.x + threadIdx.x;
  int stride = blockDim.x * gridDim.x;
  curandState state;
  curand_init((unsigned long long) clock(), index, 0, &
    state);
  extern __shared__ mapping maps[];
  if(threadIdx.x == 0){
    for(int i = 0; i < numMappings; i++)
        maps[i] = d_mappings[i];
  }
  __syncthreads();
  int target = index % numMappings;
  float2 currentP, n_Poss;
  currentP.x = maps[target].x;
  currentP.y = maps[target].y;
  for(int i = index; i < numPoints; i += stride){
    d_poss[i].x = currentP.x ;
    d_poss[i].y = currentP.y ;
    float currentProb = curand_uniform(&state);
    float totalProb = 0.0f;
    for(int j = 0; j < numMappings; j++){
        totalProb += maps[j].p;
        if(currentProb < totalProb){
            target = j;
            break;
        }
    }
    n_Poss.x = maps[target].a * currentP.x +
               maps[target].b * currentP.y +
               maps[target].x;
    n_Poss.y = maps[target].c * currentP.x +
               maps[target].d * currentP.y +
               maps[target].y;
    currentP = n_Poss;
  }
}
```

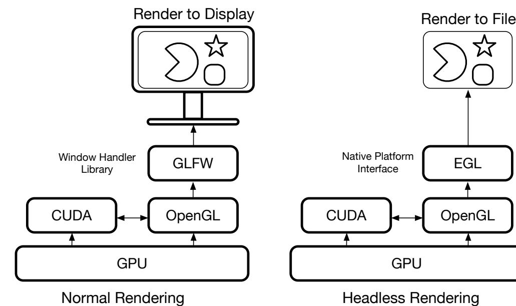

Figure 2. *Normal and headless rendering layer overview.*

Accessing Datasets on Distributed Training

Currently, deep neural network training is typically conducted using widely-used frameworks like PyTorch, TensorFlow, Keras, and ONNX. These frameworks provides various tools and utilities to train deep learning models. One of the core functionalities on these frameworks is the data loading utility, which is used to retrieve data efficiently. Specially PyTorch, which his package for Vision includes a built-in utility with predefined datasets and transforms for computer vision tasks. Moreover, PyTorch pro-

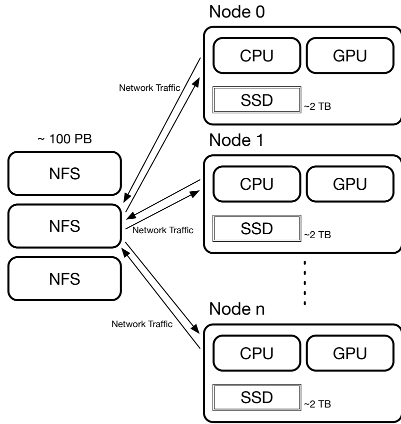

Figure 3. *Accesing datasets in a distributed environment.*

vides another utility which enables efficient distributed training of deep learning models called Data Distibuted Parallel (DDP). Using the before mentioned PyTorch utilities, distributed training can be performed across multiple nodes and GPUs in an environment as illustrated in Figure 3. In a distributed environment, training images are retrieved from an NFS system that allows for storage of several hundred petabytes. Additionally, each node in the system includes a solid-state drive with storage capacity in the range of several terabytes. According to the study by Nguyen *et al.* [20], several supercomputers listed on the TOP500 list utilize this type of storage for deep learning training, mainly to cache the entire dataset during training, thereby enabling faster access to files and reducing training time. However, the capacity of these storage units is insufficient to accommodate large datasets.

WebDatasets

Proposed by Aizman *et al.* [21] in conjuction with AIStore, the WebDataset is a Python library that enables efficient handling of large datasets in standard such as TAR. It uses the concept of streaming the data, which means that the data is loaded in small batches as it is needed for training, rather than being loaded all at once. This allows for the efficient use of disk and memory resources, especially when dealing with very large datasets. Additionally, this module also includes a number of features to help with data augmentation and processing, such as parallel processing and applying random transforms to the data. While the library is tailored for compatibility with PyTorch and TensorFlow, it also works with any framework supporting Python's Iterator protocol. Despite being similar to distributed PyTorch utilities in terms of file retrieval for training, the usage of WebDataset reduces the number of queries per worker to the NFS system by housing the entire dataset in small TAR containers.

Real-time feed loader trough GPU rendering

We have employed CUDA and OpenGL to accelerate the creation of FractalDB by implementing a parallel version of the IFS routine. Our approach can be used to generate larger Fractal datasets, such as Fractal-100k (100 million images) or Fractal-300k (300 million images), although using NFS system for hosting may not be optimal for training on multiple nodes. To address this, we proposed real-time rendering of Fractal images during training, using only the descriptors (csv files with the parameters)

Listing 3. *Custom loader pseudocode for real-time Fractal rendering.*

```
from torch.utils.data import Dataset
Class FractalRenderer(Dataset):
    def __init__(self):
        load_csv()
        construct_instances()
    def __len__(self):
        return len(total_instances)
    def render_instance(self,index)
        points = computeIFSCUDA(index)
        image = render_EGL(points)
        return image
    def __getitem__(self,idx): -> Tuple[Any, Any]:
        sample = render_instance(idx)
        return sample, target
```

for each Fractal to generate the corresponding image. FractalDB-10k (10 million images) requires approximately 271 GB of storage space, while the descriptors take up only 40MB. We developed a custom loader that includes all the necessary acceleration routines for fast Fractal generation, including the IFS parallel implementation, OpenGL rendering with CUDA interoperability, and headless rendering. The proposed custom loader is based on inheriting the PyTorch Dataset class to maintain consistency with the PyTorch loader pipeline. The *FractalRenderer* class loads Fractal descriptors during the object's creation and defines the number of instances per class to determine the dataset size. This enables the creation of a dataset list used in the PyTorch pipeline to retrieve samples and targets for training. The *__getitem__* function can retrieve an image by providing an index or ID for each Fractal, and the *render_instance* function can generate a Fractal image using OpenGL and the fast IFS computation. This design is optimized for the PyTorch pipeline, which uses *__getitem__* to batch multiple images from the loader. A pseudocode of this approach is provided in Listing 3.

Evaluation

In this section, we present the results of our experiments using various data loaders and the creation of multiple versions of FractalDB. We provide a detailed description of our experimental environment and setup, as well as a comparison between the original CPU implementation and our GPU approach for Fractal image creation. Furthermore, we report the outcomes of an image retrieval test and full training using large FractalDB.

Experimental Environment

During our development, experiments, and evaluation we utilized the AI Bridging Cloud Infrastructure (ABCI) [9] supercomputer, which is specialized for AI. The supercomputer is composed of two types of nodes: those hosting A100 GPUs and those equipped with V100 GPUs. The Volta-equipped nodes, which were more widely available in number, consist of 1,088 compute nodes with 2 Intel Xeon Gold 6148 CPUs per node (total of 40 cores), 384 GiB of DRAM, 4 NVidia V100 GPUs, and InfiniBand EDR NICs. Each node is also equipped with 1.6TB of local storage and shares a 35PB Lustre parallel filesystem. The Ampere type consist of 120 compute nodes with 2 Intel Xeon Platinum 8360Y Processor per node (total of 76 cores), 512 GiB of DRAM, 8 NVidia A100 GPUs, and InfiniBand HDR. Additionally, each node has 2.0TB of local storage and shares a connection to the Lustre NFS.

Creation of Large FractalDB by GPU Rendering

This section presents the performance evaluation of our fast IFS and rendering method for creating FractalDB on ABCI su-

Performance comparison on the creation of FractalDB-1k.

	Images/sec	Total Time
Python-CPU	5.4	2d 0h 30m
CUDA-GPU	1002.5	29m 1s

Large Fractal Datasets.

Type	Total Images (millions)	Size
FractalDB-1k	1	24GB
FractalDB-10k	10	271GB
FractalDB-21k	21	560GB
FractalDB-100k	100	2.70TB
FractalDB-300k	300	8.10TB

percomputer. We compared our GPU implementation against the original Python CPU implementation and reported the results in Table 1. Our GPU implementation achieved a 185x higher image throughput compared to the CPU implementation due to parallel computation of IFS on a large number of threads on the GPU. With this high performance, we could create FractalDB-1k in just 29 minutes as opposed to 2 days in the original implementation. Larger versions of FractalDB were also created as shown in Table 2. This approach allowed us to produce FractalDB-300k, which contains 300 million images, similar in size to JFT-300M. This will enable the study on large datasets on Vision Transformers and newer architectures.

Large FractalDB Training using Real-time Feed Loader

We evaluated loading images using different loaders, including the standard PyTorch loader with NFS and SSD locations, WebDatasets, and our GPU renderer. We generated several versions of the FractalDB, ranging from one thousand to one million images. The experiments were conducted on A100 nodes on ABCI, with 8 MPI jobs (for each GPU inside the node) and a batch size of 64. We set the number of workers per MPI job to 8. Figure 4 presents the time required to load all the images. The results show that the SSD provides the fastest image retrieval, even for 1 million images which can be loaded in under 13 seconds due to the high bandwidth and low latency. The NFS system loader has good performance for small numbers of images but becomes slow for over 100,000 images. The WebDatasets loader has lower performance for small numbers of images compared to the other approaches, but it surpasses the NFS system loader for larger image sizes due to less bottleneck generated from loading TAR files instead of individual image files. The GPU renderer loader provides comparable performance for retrieving small numbers of images, and maintains similar performance to WebDatasets for larger numbers of images. However, loading the whole dataset from the SSD is the best strategy for training, providing almost an order of magnitude faster retrieval. Nevertheless, the larger FractalDB-100k or FractalDB-300k datasets do not fit in the SSD, as shown in Table 2.

Next, we assess the impact of image loading on a real training scenario, using the relatively small DeiT-Tiny Vision Transformer architecture [24]. We employed 8 ABCI A100 nodes, totaling 64 GPUs, with a maximum batch size of 1,024 limited by the GPU memory. The number of workers per job was maintained

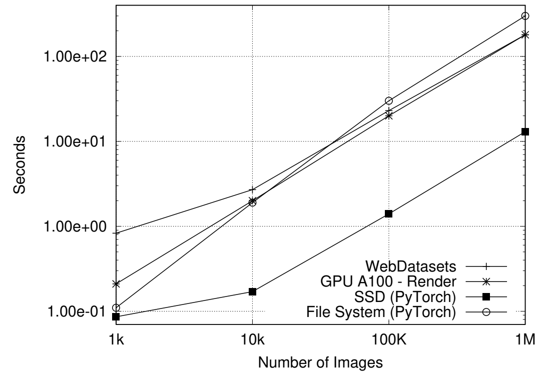

Figure 4. Image retrieval test. The test is running on one A100 ABCI node. We set up the BS = 64, and the number of workers to 8.

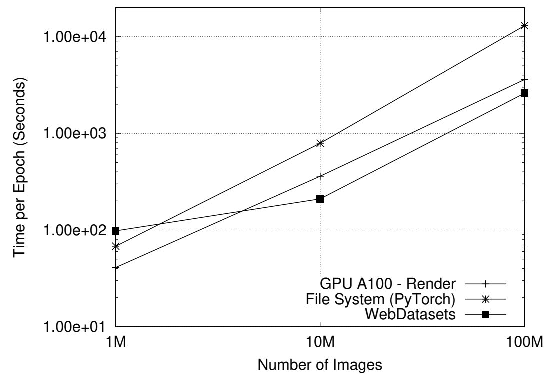

Figure 5. Total time for one epoch on training uing DeiT-Tiny-224 architecture on large versions of FractalDB. We use 64 GPUs, BS = 1024, and the number of workers set to 8.

at 8. We exclude the SSD case, as the entire dataset can not fit in the device. Figure 5 illustrates the results of this experiment. Our GPU renderer loader exhibits the best performance for the 1 million image case, in contrast to the WebDataset case which shows lower performance. In the previous experiment, it was observed that loading a large number of images was similarly efficient for both the GPU renderer and WebDatasets loaders. However, during full training, the GPU renderer outperformed WebDatasets. This may be due to other operations, such as pre-processing routines, that occur during training prior to loading the data onto the GPU. The bottleneck of accessing data from the NFS system was observed for more than 10 million images, and the GPU renderer and WebDatasets were found to have a shorter training time per epoch. For the 100 million images case, WebDatasets had the best performance, with a training time of 1 epoch in 42 minutes, followed by the GPU renderer in 59 minutes, while using the NFS file system took 3 hours. As we can observe, for the large dataset case, using our GPU renderer provides more than 3x of loading performance compared to the PyTorch loader.

Conclusion

This paper proposes a GPU-accelerated approach for efficient creation and search of FractalDB, enabling the generation of large-scale versions of the dataset. Our study shows that using the GPU renderer loader during the training of 100 million images with multiple GPUs resulted in a 3x reduction in training time compared to the PyTorch loader. We compared our ap-

proach to WebDatasets, an alternative for loading large datasets in a distributed environment using TAR containers. Our GPU renderer showed lower performance when training on large datasets compared to WebDatasets due to our simplistic rendering implementation at the EGL level. Future improvements such as asynchronous memory transfers from the GPU to the CPU and implementing the complete transformation pipeline on the GPU are planned. We believe that our contribution to a quick rendering and loading pipeline will be beneficial not only for FDSL but also for any other method that employs computer-generated graphics for synthetic data.

This paper is based on results obtained from a project, JPNP20006, subsidized by the New Energy and Industrial Technology Development Organization (NEDO).

References

[1] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever. Language Models are Unsupervised Multitask Learners. In International Conference on Machine Learning (ICML), 2018.

[2] J. Devlin, M. Chang, K. Lee, and K. Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Lan- guage Understanding. In North American Chapter of the Association for Computational Linguistics (NAACL), 2019.

[3] Zhai. X, Kolesnikov. A, Houlsby. N, Beyer. L. Scaling vision transformers. In Proceedings of the IEEE/CVF, Conference on Computer Vision and Pattern Recognition 2022 (pp. 12104-12113).

[4] Kolesnikov. A, Beyer. L, Zhai. X, Puigcerver. J, Yung. J, Gelly. S, Houlsby. N. Big transfer (bit): General visual representation learning. InComputer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part V 16 2020 (pp. 491-507). Springer International Publishing.

[5] Sun. C, Shrivastava. A, Singh. S, Gupta. A. Revisiting unreasonable effectiveness of data in deep learning era. In Proceedings of the IEEE international conference on computer vision 2017 (pp. 843-852).

[6] Kataoka. H, Okayasu. K, Matsumoto. A, Yamagata. E, Yamada. R, Inoue. N, Nakamura. A, Satoh. Y. Pre-training without natural images. In Proceedings of the Asian Conference on Computer Vision 2020.

[7] Kataoka. H, Hayamizu. R, Yamada. R, Nakashima. K, Takashima. S, Zhang. X, Martinez-Noriega. EJ, Inoue. N, Yokota. R. Replacing Labeled Real-image Datasets with Auto-generated Contours. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition 2022 (pp. 21232-21241).

[8] Nakashima. K, Kataoka. H, Matsumoto. A, Iwata. K, Inoue. N, Satoh. Y. Can vision transformers learn without natural images?. In Proceedings of the AAAI Conference on Artificial Intelligence 2022 Jun 28 (Vol. 36, No. 2, pp. 1990-1998).

[9] National Institute of Advanced Industrial Science and Technology, ABCI Supercomputer, https://abci.ai, 2023, [January 2023].

[10] Asano. YM, Rupprecht. C, Zisserman. A, Vedaldi. A. Pass: An imagenet replacement for self-supervised pretraining without humans. In NeurIPS Track on Datasets and Benchmarks, 2021.

[11] PyTorch Core Team, PyTorch Vision Docs, https://pytorch.org/vision/stable/datasets.html, [January 2023].

[12] Yang. K, Qinami. K, Fei-Fei. L, Deng. J, Russakovsky. O. Towards fairer datasets: Filtering and balancing the distribution of the people subtree in the imagenet hierarchy. In Proceedings of the 2020 conference on fairness, accountability, and transparency 2020 Jan 27 (pp. 547-558).

[13] Carlini. N, Hayes. J, Nasr. M, Jagielski. M, Sehwag. V, Tramer. F, Balle. B, Ippolito. D, Wallace. E. Extracting Training Data from Diffusion Models. arXiv preprint arXiv:2301.13188. 2023 Jan 30.

[14] Kataoka. H, Matsumoto. A, Yamada. R, Satoh. Y, Yamagata. E, Inoue N. Formula-driven supervised learning with recursive tiling patterns. In Proceedings of the IEEE/CVF International Conference on Computer Vision 2021 (pp. 4098-4105).

[15] M. F. Barnsley. Fractals Everywhere. Academic Press. New York, 1988.

[16] Mahajan. D, Girshick. R, Ramanathan. V, He. K, Paluri. M, Li. Y, Bharambe. A, Van Der Maaten. L. Exploring the limits of weakly supervised pretraining. In Proceedings of the European conference on computer vision (ECCV) 2018 (pp. 181-196).

[17] Thomee. B, Shamma. DA, Friedland. G, Elizalde. B, Ni. K, Poland. D, Borth. D, Li. LJ. YFCC100M: The new data in multimedia research. Communications of the ACM. 2016 Jan 25;59(2):64-73.

[18] Schuhmann. C, Beaumont. R, Vencu. R, Gordon. CW, Wightman. R, Cherti. M, Coombes. T, Katta. A, Mullis. C, Wortsman. M, Schramowski. P. LAION-5B: An open large-scale dataset for training next generation image-text models. In Thirty-sixth Conference on Neural Information Processing Systems Datasets and Benchmarks Track.

[19] Dryden. N, Böhringer. R, Ben-Nun. T, Hoefler. T. Clairvoyant prefetching for distributed machine learning I/O. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis 2021 Nov 14 (pp. 1-15).

[20] Nguyen. TT, Trahay. F, Domke. J, Drozd. A, Vatai. E, Liao. J, Wahib. M, Gerofi. B. Why globally re-shuffle? Revisiting data shuffling in large scale deep learning. In 2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS) 2022 May 30 (pp. 1085-1096). IEEE.

[21] Aizman. A, Maltby. G, Breuel. T. High performance I/O for large scale deep learning. In 2019 IEEE International Conference on Big Data (Big Data) 2019 Dec 9 (pp. 5965-5967). IEEE.

[22] Baradad. M, Chen. CF, Wulff. J, Wang. T, Feris. R, Torralba. A, Isola. P. Procedural Image Programs for Representation Learning. In Advances in Neural Information Processing Systems 2022 Nov 26.

[23] TOP500, The List, https://www.top500.org/, 2023, [January. 2023]

[24] Touvron. H, Cord. M, Douze. M, Massa. F, Sablayrolles. A, Jégou. H. Training data-efficient image transformers & distillation through attention. In International conference on machine learning, 2021, (pp. 10347-10357). PMLR.

Author Biography

Edgar Josafat Martinez-Noriega obtained his Doctorate in Computer Science from the University of Electro-Communications, Tokyo in 2022. Following this, he has been employed as a Post-Doctoral Researcher at the National Institute of Advanced Industrial Science and Technology (AIST), working on the application of synthetic datasets for large-scale deep learning. His research focuses on parallel computing, computer graphics, and deep learning.

Rio Yokota is a professor at the Global Scientific Information and Computing Center, Tokyo Institute of Technology. His research focuses on high performance computing, linear algebra, and machine learning. He has developed several libraries, including ExaFMM for fast multipole methods, and Hatrix for hierarchical low-rank algorithms. He has received the Gordon Bell prize in 2009 using the first GPU supercomputer. Rio is a member of ACM, IEEE, and SIAM.