IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2017.DOI

Overlapped Data Processing Scheme for Accelerating Inference Procedure in Machine Learning

JINSEO CHOI[1] AND DONGHYUN KANG[1]

[1]Changwon National University, 20 Changwondaehak-ro Uichang-gu Changwon-si, Gyeongsangnam-do (51140), South Korea

Corresponding author: Donghyun Kang (e-mail: donghyun@gs.cwnu.ac.kr).

This research was supported by Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Education(NRF-2021R1I1A3047006)

ABSTRACT

With the enhancement of machine learning (ML) technologies, many researchers and developers have challenged how to solve the traditional problems using ML technologies with a rich set of hardware resources. However, despite the many efforts to study the internal processing of ML to maximize the resources provided, they are still wasted. In this paper, we first observe possible optimization opportunities by studying the data flows of the conventional inference procedure in detail. Subsequently, we propose `ol.data`, the first software-based data processing scheme that aims to (1) overlap training and validation steps in one epoch or two adjacent epochs, and (2) perform validation steps in parallel, which helps to significantly improve not only the computation time but also the resource utilization We implement a Convolution Neural Network (CNN) model using `ol.data` and compare it with the traditional approaches, Numpy (*i.e.*, baseline) and tf.data on three different datasets. The results show that `ol.data` improves the inference performance by up to 41.8% while using the highest CPU and GPU utilization.

INDEX TERMS Machine learning, TensorFlow, CPU/GPU utilization, Overlapping, Multiple threads

I. INTRODUCTION

FOR several years, machine learning (ML) has achieved great success in diverse computing environments [1]–[3]. Emerging hardware technologies (*e.g.*, Ampere [4], Apple M1 [5], TPU [6], etc.) and open-source libraries for supporting ML (*e.g.*, TensorFlow [2], [7], PyTorch [8], Caffe [9], Keras [10], etc.) had been helpful in accelerating the successful adoption of ML theories and models. For example, an inference procedure has currently becomes a standard feature in ranking, image classification, and recommendation systems [11], [12]. The procedure commonly runs on a machine that consists of CPUs and lots of GPUs; the CPUs transform data into *tensors*, and the GPUs perform *training* and *validation* steps to provide high-accuracy predictions. Unfortunately, such an inference procedure frequently wastes rich hardware resources (*e.g.*, CPU cycles and memory), as all steps need to be fully serialized before starting the next epoch; an epoch refers to iteration granularity in which the entire dataset is trained and validated once.

To solve the aforementioned issue, there were many interesting approaches [13]–[16]. Some researchers focused on the same issue and they proposed tf.data that processes some of the inference steps (*i.e.* data load and preprocessing) in a parallel way [13]. Unfortunately, tf.data still misses opportunities in exploiting parallelism inside one or multiple epochs. Meanwhile, recent research offloads time-consuming parts of the inference procedure into hardware logic (*i.e.*, FPGA) to sufficiently utilize the hardware resources [17]–[21]. For example, Ren *et al.* [17] focused on GPU-CPU hybrid training technology that offloads data to reduce the amount of data movement to/from GPU. Kwon *et al.* [20] introduced the prototype of the multi-FPGA acceleration for large-scale RNNs. However, we think that the offloading approaches are difficult to be gained wide adoption in various environments because they require not only new hardware but also the modification of the corresponding codes.

In this paper, we propose a new data processing scheme, called overlapped data (`ol.data`), to efficiently address the resource waste issue. The key motivation of `ol.data` is that an inference procedure (*e.g.*., data loading, preprocessing,

training, and validation) and the traditional method inevitably waste hardware resources by running each step using a single thread. To keep high CPU and GPU utilization, `ol.data` (1) logically isolates and overlaps the training and validation steps by using a *model copy* operation in adjacent epochs, and (2) allows multiple threads to simultaneously perform the validation steps. For evaluation, we implemented a Convolution Neural Network (CNN) model for image classification by extending tf.data based on TensorFlow framework which is widely used to implement ML models and extensively evaluated `ol.data` on three popular datasets, such as MNIST, fashion MNIST, and Extended MNIST. The major contributions of this paper are the following.

- We first describe data flows of the conventional inference procedure in detail and carefully observe how much CPU and GPU resources are utilized in the entire inference model.
- We propose `ol.data`, which reduces the idle time to start the next step by overlapping the CPU and GPU executions and saves the computing time consumed by the validation step using multiple threads.
- Finally, we show comprehensive evaluation results to confirm the effectiveness of `ol.data`. For comparison, we additionally implemented CNN models with Numpy and tf.data on TensorFlow framework, respectively. The experimental results show that `ol.data` reduces the elapsed time consumed by the inference procedure by up to 41.8% compared with tf.data.

The rest of this paper is organized as follows. Section II introduces how the inference procedure behaves to correctly make a prediction and describe and describes our motivation from new observation results. Section III explains the key techniques of `ol.data`. Section IV shows our evaluation and comparison with existing data processing approaches. Finally, Section V introduces related work and Section VI concludes this paper.

II. BACKGROUND AND MOTIVATION

In this section, we describe how to transform data into tensors and run the training and validation operations in detail. We introduce our new observations from the monitoring of CPU and GPU utilization during an inference procedure and the motivation for this work.

A. DATA FLOWS IN ML INFERENCE PROCEDURE

In ML domains, raw input data are transformed by a designated procedure and transferred from CPUs to GPUs to predict a correct answer.

Figure 1 presents the data processing flow in an entire inference procedure in one epoch. In general, an inference procedure can be divided into five steps: data-load, preprocessing for training, training, preprocessing for validation, and validation. (1) In the data-load step, raw data are loaded as an arguments of input through either networks or file systems; this step is performed only from the initial time

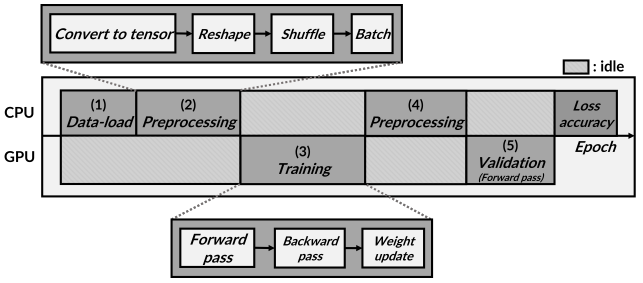


FIGURE 1: Data flow in the conventional inference procedure.

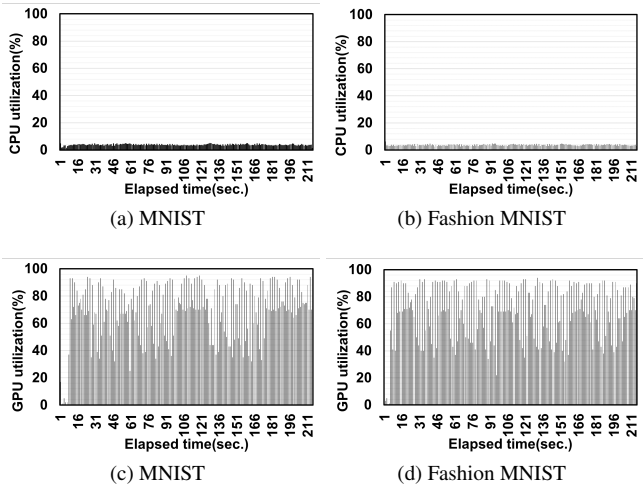


FIGURE 2: CPU and GPU utilization for CNN model (100 epochs).

to the start. (2) Next, Some of the loaded data from the previous step are transformed into tensors for training; approximately 80% entire data are commonly used in this step.. The tensors are reshaped and shuffled to avoid erroneous inferences and to achieve higher accuracy. Then, a set of tensors is delivered as training arguments for ML models (*e.g.*, CNN [22], RNN [23], and DNN [24]) by going through a batch operation. (3) To find the correct answers from the corresponding ML model using the tensors, the training step runs three operations: forward pass, backward pass, and weight update. (4) For preprocessing for the next validation step, the remaining data aapproximately bout 20% of the loaded data are transformed similarly as before. (5) Finally, the validation step makes acceptable predictions based on the weights learned in the training step. After finishing this step, the loss and accuracy values are calculated to confirm that the model results are acceptable.

Note that, the above steps are conducted in a serialized fashion because one thread is responsible for each step; the next step is conducted after finishing the prior step. As shown in Figure 1, the CPU and GPU resources are not used even though they are idle, because each step waits for the completion of the previous step. Thus, a rich set of hardware resources are unintentionally wasted.

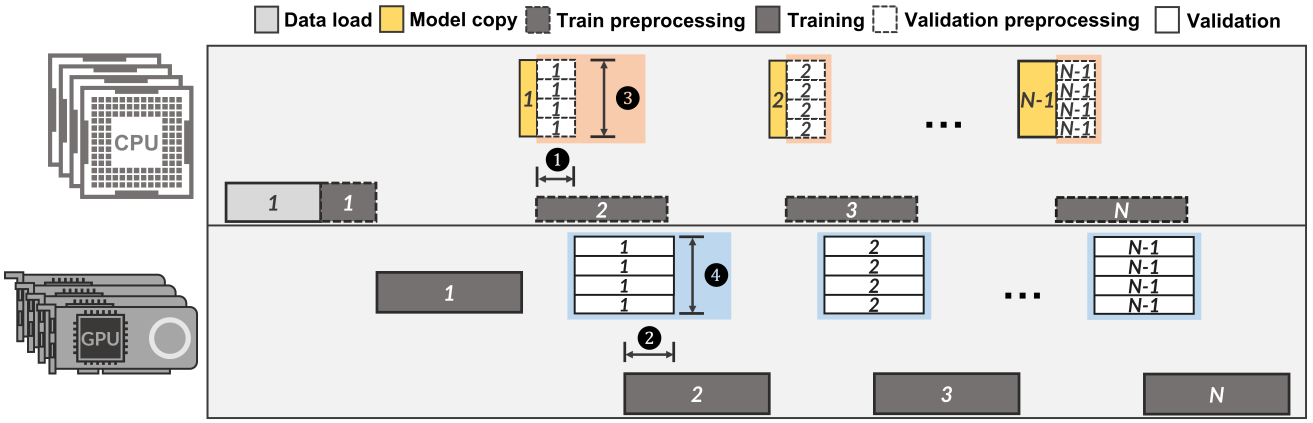


FIGURE 3: The process of ol.data. The number inside each rectangle means the corresponding epoch time.

B. CPU AND GPU UTILIZATION

In order to confirm our intuition as mentioned before, we conducted experiments to study CPU and GPU utilization. In this evaluation, we implemented an image classification model based on TensorFlow with Numpy and followed the default guidelines to build a model using the MNIST and fashion MNIST datasets, which are commonly used in image classification models [25], [26].

Figure 2 shows the evaluation results of utilization measured while running the model on a machine composed of CPUs and GPUs (we will describe the evaluation setup in detail later). Unfortunately, as shown in Figure 2(a) and Figure 2(b), the model utilizes approximately 3.7% of CPU resources on both datasets. This observation is very interesting in that the steps running on CPU do not fully utilize the available resources, even though there is no interference from other tasks. To understand why CPU has inferior utilization, we conducted a study and found that (1) the utilization collapse at the start time comes from an I/O overhead in the data-load step, and (2) most parts of poor utilization result from the execution of the order of the four operations in the preprocessing step on a single thread. In addition, some operations require considerable time and effort. For example, the shuffle operation first reads an array of NumPy to obtain the index of each tensor and randomly exchanges an order of all tensors. Figure 2 also shows a trend of the utilization frequently drops when the preprocessing step is triggered after completion of the GPU operations.

Meanwhile, utilization of GPU shows better than that of CPU but reveals dramatic fluctuations (see Figure 2(c) and Figure 2(d)). The reason behind this is that the training and validation step should wait for the completion of the preprocessing inside one epoch and this wait is repeated every epoch. In summary, the serialized step in the traditional inference procedure can waste a rich set of CPU and GPU resources. This limitation motivates the need for intelligent isolation and parallel scheme.

III. DESIGN

In this section, we propose a new data processing scheme called `ol.data` to ensure high CPU and GPU utilization while implementing an inference procedure. The design goal of `ol.data` is to accelerate the overall performance of an ML model while efficiently isolating the tasks running on the CPUs and GPUs. Figure 3 shows the data processing flow of `ol.data` in detail.

A. DATA-LEVEL ISOLATION

As mentioned in Section II-A, the steps belonging to the same inference model are serialized using a single thread even though they run on physically independent cores. The key reason is that a few tasks on the CPU or GPU share the data structures for managing model parameters (*e.g.*, *weight values*). In general, a task in the validation step employs the parameters which have been finally updated by the previous training task; since the model parameters are frequently updated during the execution of the training step, the value of the parameters in the middle of the execution is incomplete to use in the validation task.

To efficiently disable this data sharing, `ol.data` has an additional operation between the training and validation steps, called the *model copy* operation. This operation provides a simple and powerful solution in that it can isolate and overlap the training and validation tasks with a low overheads. As shown in Figure 3, `ol.data` performs three steps including data-load, preprocessing for training, and training step, in the same way as processing the traditional ones. Then, `ol.data` executes the model copy operation in which the whole data structures updated by the previous training task are copied for enabling the *data-level isolation*. Finally, `ol.data` additionally creates new tasks (*i.e.*, threads) and runs them so as to separate the flow of the remaining steps in the current epoch from the main task. The created tasks simultaneously perform validation based on the copied model parameters after finishing the corresponding preprocessing step (we will describe how `ol.data` deals with them in detail later). Meanwhile, the main task starts the next epoch by triggering the preprocessing step for training regardless

of the process in the validation tasks. Such a concurrent execution never interferes with the flow and calculation of the separated validation tasks running on CPU and GPU because `ol.data` efficiently takes the data-level isolation. As a result, `ol.data` allows improving the overall performance by the overlapping of two or more steps. In other words, `ol.data` can concurrently run two preprocessing steps; one is for the training of the next epoch and the other is for the validation of the current epoch (❶). The execution of the validation step can be overlapped with that of the training step as much as possible (❷). We believe that such a parallelism improves the overall performance with high CPU and GPU utilization.

Meanwhile, if the training step for the next epoch completes before the validation step in the current epoch, the model copy operation is postponed until finishing the validation step; it helps to reduce the management overhead of the shared data structures.

B. ENHANCED VALIDATION STEP

As the aforementioned before, a long completion time for the validation step might be harmful in that the model copy operation is delayed. To reduce the inevitable wait, we enhanced the validation step to further optimize `ol.data`. The key idea of the enhancement is to (1) split the data used in the validation step and (2) efficiently schedule the tasks derived from the related steps using multiple threads. Therefore, the enhanced validation step has extra operations for making acceptable predictions.

Figure 3 also shows how `ol.data` enhances the validation step along with the corresponding preprocessing step. (1) After the model copy operation, `ol.data` creates concurrent tasks (*i.e.*, threads) by up to the predefined number of threads; we limit the number of overlapped threads to avoid interference overheads across the tasks. (2) `ol.data` equally splits the whole data used by the preprocessing step into smaller chunks to allocate them to the created tasks one by one. The data belonging to each chunk is preprocessed by the corresponding task in parallel (❸). Therefore, `ol.data` can reduce the time consumed by the preprocessing step compared with the conventional step in a batch. (3) The tasks concurrently perform the validation step using the data from the preprocessed step to calculate predictions. (❹). Note that in the enhanced validation step, each task is performed only to make predictions for the allocated data in the corresponding chunk based on the copied data structures. As a result, the execution of the enhanced validation step can have an opportunity to be overlapped on the GPU as much as possible. However, `ol.data` can be forced to remain in this step for some time to complete all tasks, because the completion times of all tasks are different. (4) Finally, `ol.data` reports the loss and accuracy of this validation step by computing the average of all tasks; it is required because the calculated loss and accuracy derived from a task are not visible to the other tasks. We believe this operation incurs negligible collection overhead.

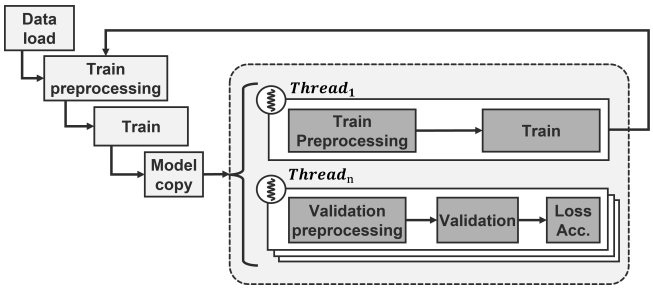


FIGURE 4: Data flow in `ol.data` procedure

Figure 4 shows how to best leverage training and validation steps with multiple threads. As shown in Figure 4, the idea of `ol.data` is simple and straightforward, but it has opportunities to optimize the overall performance with high CPU and GPU utilization because it efficiently provides the data-level isolation and enhanced validation step.

IV. EVALUATION

In this section, we shows our experimental results in detail. Specially, our evaluation aims to answer the following questions:

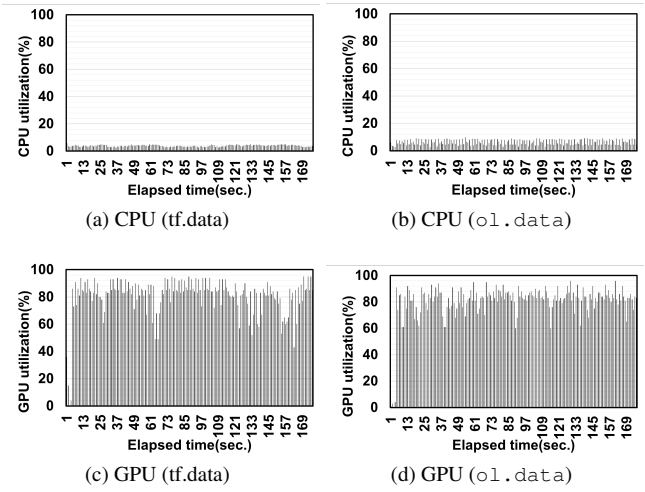
- How effective is `ol.data`'s parallelism in CPU and GPU utilization?
- Does high utilization derived from benefits of `ol.data` help to improve performance under the various workloads (*i.e.*, datasets) of ML?
- How does `ol.data` perform when the dataset size is large?
- Does `ol.data`'s parallelism negatively affect the loss or accuracy of ML model?

A. EXPERIMENTAL SETUP

For evaluation, we implemented `ol.data` based on tf.data which is one of the state-of-the-art approaches realizing an inference model. We compared it with two traditional approaches, such as Numpy and tf.data; Numpy is well-known as a default implementation (*i.e.*, baseline) and tf.data allows to overlap some steps with multiple threads. Our experiments were performed by implementing a well-known CNN inference model. For comparison, we repeated the same experiment on three different datasets: MNIST (127MB) [25], Fashion MNIST (155MB) [26], and Extended MNIST (1.6GB) [27]. All experiments were run 64-bit Linux on a

TABLE I: Hardware and software environment.

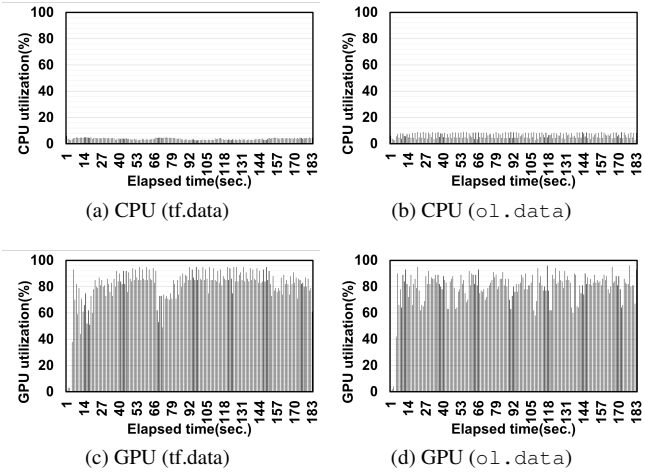
	Description
CPU	Intel® Core™ i9-12900KF
GPU	NVIDIA GeForce™ GTX 1650
Memory	32GB
Storage	Intel® 660P SSD (512GB)
OS	Ubuntu 20.04.1 LTS(64-bit)
TensorFlow	version 2.3.0
Keras	version 2.8.0



(a) CPU (tf.data) (b) CPU (`ol.data`)

(c) GPU (tf.data) (d) GPU (`ol.data`)

FIGURE 5: Comparison of CPU utilization on MNIST dataset.



(a) CPU (tf.data) (b) CPU (`ol.data`)

(c) GPU (tf.data) (d) GPU (`ol.data`)

FIGURE 6: Comparison of GPU utilization on Fashion MINST dataset.

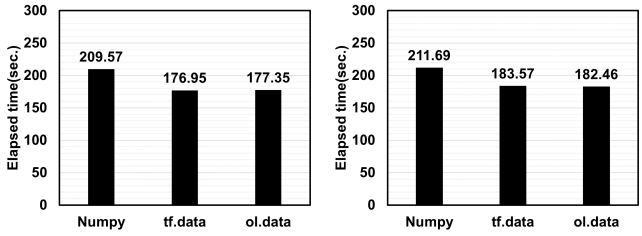
machine with Intel i9 CPU and NVIDIA GTX 1650 GPU. Table I lists the details of a machine and software environment.

For a comprehensive analysis, inference was run for 100 epochs where one epoch is a full pass over the dataset. As mentioned before, tf.data and `ol.data` can optimize some training and validation steps by configuring the number of threads. Thus, We set the number of threads as 24; it is the same as the number of physical CPUs. Finally, we used `sysstat` [28] and `gpustat` [29] to measure resource utilization on CPU and GPU while running training and validation steps.

B. DEFAULT MNIST AND FASHION MNIST DATASETS

We first conducted the same experiment mentioned in Section II with `ol.data` and tf.data to confirm the effect of multiple threads. In this experiment, we employ widely used default datasets (*i.e.*, MNIST and Fashion MNIST) to perform CNN model.

Figure 5 and Figure 6 illustrate how much `ol.data`



(a) MNIST (b) Fashion MNIST

FIGURE 7: Overall performance comparison on two different datasets.

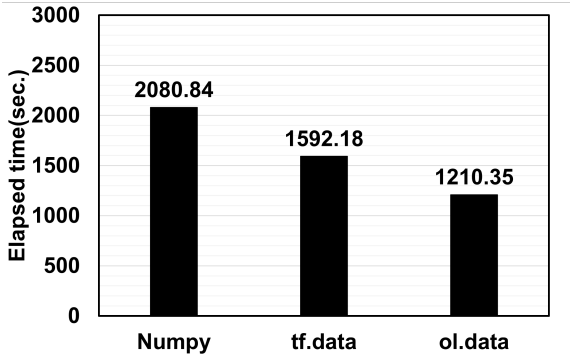
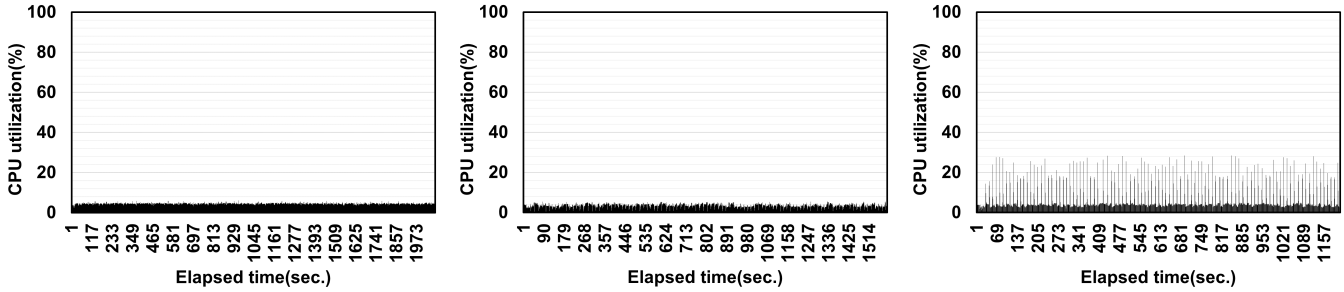


FIGURE 8: Overall performance comparison on Extended MNIST dataset.

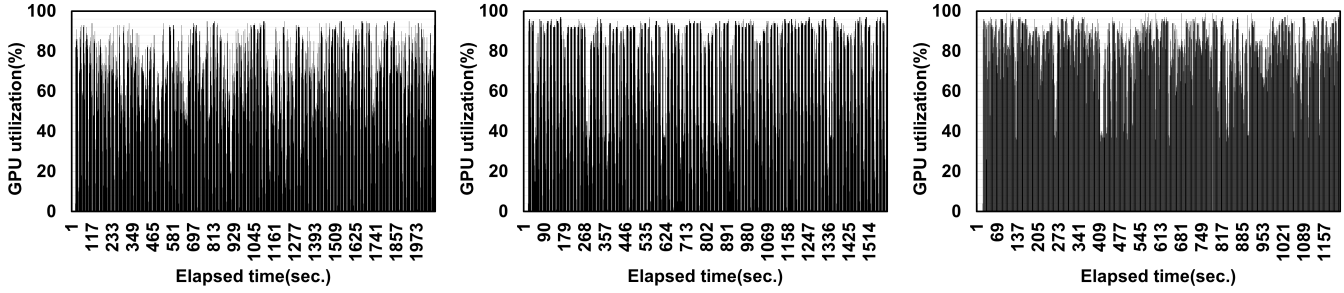
and tf.data efficiently can utilize CPU and GPU resources on MNIST and Fashion MINST datasets, respectively. As we expected, they show higher CPU and GPU utilization compared with Numpy where most steps are serialized with a single thread (see Figure 2). Especially, `ol.data` clearly presents the highest utilization in all kinds of datasets. To clearly confirm the effectiveness of `ol.data`, we measured average CPU and GPU utilization during the training and validation steps. As a result, compared with Numpy on MNIST, `ol.data` improves the overall CPU and GPU utilization by 71% and 16.2%, respectively. The reason behind such improvement is that `ol.data` enables the *enhanced validation step* where tensors can be simultaneously inserted and produced when performing the operations associated with each layer in inference model. In addition, `ol.data` depicts higher utilization of CPU than that of tf.data; it improves by up to 64.9% on MNIST and 68.9% on Fashion MNIST, respectively. The good CPU utilization of `ol.data` mainly comes from the *data-level isolation* where training and validation step no longer waits for each other to finish. Unlike CPU, average GPU utilization of `ol.data` is similar to that of tf.data. To understand the reason, we performed breakdown analysis and found that the overlapped region on GPU is very short because of small size of the dataset.

Next, we validate the overall performance to show whether high CPU and GPU utilization of `ol.data` can lead to performance gain. Figure 7 shows the elapsed time of Numpy, tf.data, and `ol.data`. Figure 7 clearly indicates the poor performance of Numpy compared with `ol.data`. Surpris-



(a) Numpy (b) tf.data (c) `ol.data`

FIGURE 9: Comparison of CPU utilization on Extended MNIST dataset.



(a) Numpy (b) tf.data (c) `ol.data`

FIGURE 10: Comparison of GPU utilization on Extended MNIST dataset.

ingly, `ol.data` shows comparable performance compared to tf.data even though it took the highest CPU utilization on both datasets. The main reason of the results is that `ol.data` unfortunately reveals the potential inefficiency of the extra operation for supporting parallelism (*i.e.*, model copy and waiting time to calculate of loss and accuracy). However, we believe that these overheads would be negligible as the size of the dataset increases; it increases the region for overlapping of `ol.data` because the validation steps spend more time to complete.

C. EXTENDED MNIST DATASET

To confirm the efficiency of `ol.data`, we additionally conducted the same experiment and analysis on Extended MNIST dataset [27]; the dataset size of the Extended MNIST is 12.9 times larger than that of the default MNIST.

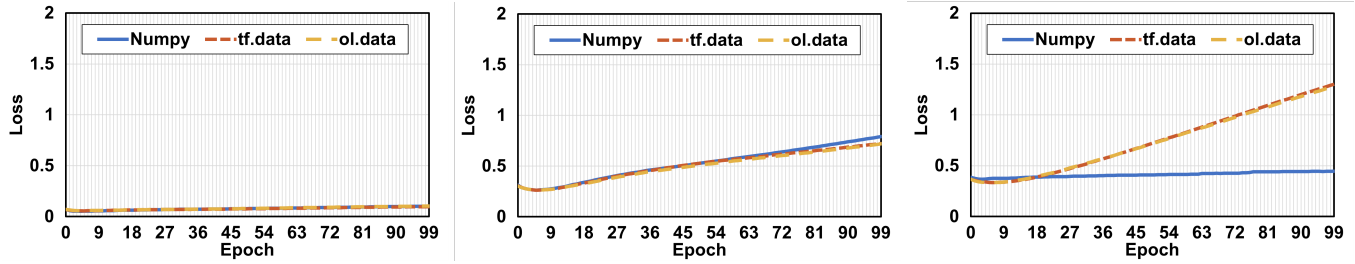
Figure 8 presents the performance results of CNN model under the three different approaches, including Numpy, tf.data, and `ol.data`. As we expected, `ol.data` shows the speedup of CNN model compared with traditional approaches; it achieves performance improvement by up to 41.8% and 24% compared to Numpy and tf.data, respectively. As mentioned before, these results mean that `ol.data` can naturally isolate between training and validation steps with the large overlapped region and the overhead derived from the extra operations is negligible. To understand such a performance gain in detail, we conducted a sensitivity analysis by measuring CPU and GPU utilization on the fly.

Figure 9 and Figure 10 show our empirical results. One interesting result is that `ol.data` utilizes highly CPU resources by up to 28.46% (see Figure 9(c)). To clearly quantify the impact of high CPU utilization, we calculated average CPU utilization. As a result, we found that CPU utilization of `ol.data` outperforms that of tf.data by 75.7% on average. To analyze such a significant gap, we focused on the gain of `ol.data` in a large dataset and we found the following strengths:

- The large size of the dataset increases the amount of data read and transformed via preprocessing steps of each epoch, thus, the steps should spend more time to be completed on CPU.
- The long processing time leads to more opportunities for computation in parallel because `ol.data` allows to overlap a preprocessing step for validation with a step for inference (see ❶ in Figure 3).
- In addition, `ol.data` can utilize more resources in that its preprocessing step for validation runs in parallel on CPU (see ❸ in Figure 3).

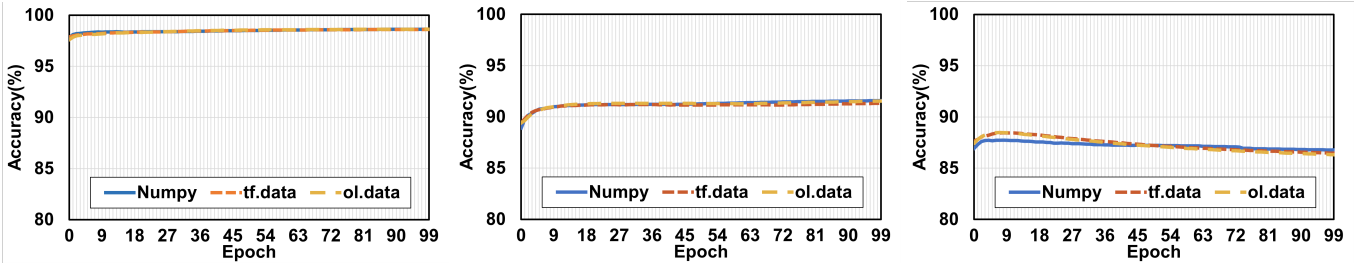
On the other hand, the traditional approaches show a similar trend of the results in Section IV-B even though the size of the dataset increases. This is because Numpy and tf.data perform most steps of their counterparts in Figure 9(c) in a serialized way.

Now, we explain how `ol.data` works to maximize GPU utilization. Figure 10 presents GPU utilization on Extended MNIST. As shown in Figure 10(c), `ol.data` often plots a region of high utilization on GPU and it is much more thick and intense compared to that of other figures; average



(a) MNIST (b) Fashion MNIST (c) Extended MNIST

FIGURE 11: Loss patterns for the inference model (100 epochs) on three different datasets. Lower value is better, in model.



(a) MNIST (b) Fashion MNIST (c) Extended MNIST

FIGURE 12: Accuracy patterns for the inference model (100 epochs) on three different datasets. Higher value is better, in model.

utilization of `ol.data` is 38.7% and 30.2% higher than Numpy and tf.data, respectively. Especially, this result is significantly meaningful in that the gap between `ol.data` and the traditional approaches is larger than that of the result in Section IV-B. We believe that the noteworthy improvement comes from the same strengths as mentioned before (see ❷ and ❹ in Figure 3).

In summary, `ol.data` can have several benefits: (1) maximizing CPU and GPU utilization by processing crucial steps of inference model in parallel , (2) reducing the amount of time waiting for completion of the prior step.

D. LOSS AND ACCURACY

Unlike the traditional approaches, `ol.data` performs validation steps with multiple threads. Thus, It needs to merge model loss and accuracy values of each validation task at the end of an epoch; we generate the model loss and accuracy by calculating the average loss and accuracy of all tasks. Note that the merge operation does not cause the huge overhead as described before. But, this behavior in `ol.data` might negatively affect the loss or accuracy of the inference model. To confirm this, we additionally measured the loss and accuracy of Numpy and tf.data.

Figure 11 and Figure 12 present model loss and accuracy pattern during 100 epochs. As shown in Figure 11 and Figure 12, `ol.data` is comparable to the traditional approaches. These results mean that there is no negative impact caused by the split operation of the *enhanced validation step* in terms of loss and accuracy. Meanwhile, `ol.data` shows higher loss values than Numpy over time (*i.e.*, epoch) in Figure 11(c). To understand the reason, we observed the processing flow in detail and found that the root cause comes from the behavior of tf.data; it is out of scope of this paper. But, we believe that the gap is negligible because `ol.data` completes the inference model 41.8% faster by using multiple threads.

V. RELATED WORK

There were diverse prior efforts in building elegant and efficient machine learning (ML) services. In this section, we describe two categories of related work: (1) optimizaing resource utilization and (2) offloading techniques for ML models.

Optimizing resource utilization. Recent work focused on the internal behaviors and data flow of inference models that can be optimized to better employ the given hardware resources. tf.data [13] proposes a method to pipeline input data processing in the existing ML frameworks; it is the closest to our work.

TABLE II: Comparison of the characteristics.

	Numpy	tf.data	`ol.data`
Multi-threads support	✗	✓	✓
Input-data parallelism	✗	✓	✓
Validation step parallelism	✗	✗	✓
Computation overlapping	✗	✗	✓

To clearly compare the characteristics of `ol.data`, Table II summarizes the difference between the existing approaches and `ol.data`. Like `ol.data`, tf.data can support multiple threads but it employs them to handle the input data. However, after finishing the processes of input data, the parallelism on tf.data is disabled because the remaining steps are handled in a serial way, like Numpy. On the other

hand, `ol.data` fully utilizes multiple threads in the validation steps. In addition, `ol.data` can bring better resource utilization because it can overlap training steps with validation steps via the *data-level isolation*. Meanwhile, CheckFreq [14] provides a fine-grained checkpointing scheme that reduces the checkpointing cost in compute-intensive and time-consuming ML models using the pipelined snapshot() and persist() phase. DBS [15] allows to dynamically adjust the batch size and dataset according to the performance of the worker in distributed training of deep neural networks (DNNs). Synergy [16] provides the optimistic profiling and schedules a set of job in multi-tenant clusters; it is a workload and data-aware scheduler.

Offloading techniques for ML models. In recent, some researchers have explored optimization opportunities in ML by offloading its techniques to hardware. FlashNeuron [21] offers a peer-to-peer direct path for data transfer between a GPU and NVMe SSDs to reduce the interference of host resources, such as CPU and DRAM. Kwon *et al.* [20] also focus on reducing the amount of data movement between GPU and CPU and they proposed three parallelism levels along with an acceleration system based on Intel high-end FPGAs. FlexLearn [30] proposes a flexible on-chip learning architecture that includes the specialized data paths and dependencies to optimize parallelism for brain simulation. Finally, we believe even if the previous studies can significantly accelerate the performance of inference models, they are undesirable in diverse environments because of requiring hardware support.

VI. CONCLUSION

Designing a high-performance machine learning (ML) model based on rich hardware resources needs a clear understanding of the internal data flow. In this paper, we have described how the existing inference model works and utilizes CPU and GPU resources. We also proposed `ol.data` for fully utilizing the wasted hardware resources. The key concepts for `ol.data` is to (1) reduce the time waits for completing the previous step of the data flow inside the inference model and (2) leverage the strengths of multiple threads in terms of resource utilization. To realize our insight, we enhanced the steps of the data flow with the *data-level isolation* and *enhanced validation step*. By implementing a CNN model, we compared the effectiveness of `ol.data` with two traditional approaches: Numpy and tf.data. Our experimental results based on three widely used datasets confirmed that the inference model with `ol.data` is faster up to 41.8% on the Extended MNIST dataset while guaranteeing high CPU and GPU utilization. Especially, compared to the state-of-the-art approach, `ol.data` can increase the CPU and GPU utilization up to 75.7% and 30.2%, respectively.

Finally, we expect that our study will lead to easy development of diverse ML models and benefit future developers when designing and implementing models. In addition, the methodology of `ol.data` can be used to solve the problems of the most representative computational intelligence algorithms, such as monarch butterfly optimization (MBO) [31] and moth search (MS) [32] algorithm.

REFERENCES

[1] Q. Zhou, S. Guo, Z. Qu, J. Guo, Z. Xu, J. Zhang, T. Guo, B. Luo, and J. Zhou, "Octo: INT8 Training with Loss-aware Compensation and Backward Quantization for Tiny On-device Learning," in *Proceedings of the 2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 2021, pp. 177–191.

[2] M. Abadi, B. Paul, C. Jianmin, C. Zhifeng, D. Andy, D. Jeffrey, D. Matthieu, y. G. Sanja, I. Geoffrey, I. Michael, K. Manjunath, L. Josh, M. Rajat, M. Sherry, G. M. Derek, B. Steiner, T. Paul, V. Vijay, W. Pete, W. Martin, Y. Yuan, and Z. Xiaoqiang, "TensorFlow: A System for Large-Scale Machine Learning," in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (USENIX OSDI 16)*, 2016, pp. 265–283.

[3] J. H. Park, G. Yun, C. M. Yi, N. T. Nguyen, S. Lee, J. Choi, S. H. Noh, and Y. ri Choi, "HetPipe: Enabling Large DNN Training on (Whimpy) Heterogeneous GPU Clusters through Integration of Pipelined Model Parallelism and Data Parallelism," in *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020, pp. 307–321.

[4] "NVIDIA A100 Tensor Core GPU Architecture," https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf, 2020, [Online; accessed 12-August-2021].

[5] "Apple M1," https://www.apple.com/newsroom/2020/11/apple-unleashes-m1/, 2020, [Online; accessed 20-August-2021].

[6] J. et al., "In-Datacenter Performance Analysis of a Tensor Processing Unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 2017, pp. 1—-12.

[7] "Tensorflow," https://www.tensorflow.org/?hl=en, 2020, [Online; accessed 20-August-2021].

[8] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," *Advances in Neural Information Processing Systems*, vol. 32, pp. 8026–8037, 2019.

[9] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional Architecture for Fast Feature Embedding," in *Proceedings of the 22nd ACM International Conference on Multimedia*, 2014, pp. 675—-678.

[10] Ketkar and Nikhil, *Introduction to Keras*. Berkely, CA: Apress, 2017, pp. 97–111, accessed: 30-March-2022. [Online]. Available: https://doi.org/10.1007/978-1-4842-2766-4_7

[11] U. Gupta, C.-J. Wu, X. Wang, M. Naumov, B. Reagen, D. Brooks, B. Cottel, K. Hazelwood, M. Hempstead, B. Jia, H.-H. S. Lee, A. Malevich, D. Mudigere, M. Smelyanskiy, L. Xiong, and X. Zhang, "The Architectural Implications of Facebook's DNN-Based Personalized Recommendation," in *Proceedings of the 2020 IEEE International Symposium on High Performance Computer Architecture*, 2020, pp. 488–501.

[12] C. A. Gomez-Uribe and N. Hunt, "The Netflix Recommender System: Algorithms, Business Value, and Innovation," *ACM Transactions on Management Information Systems (TMIS)*, pp. 1–19, 2015.

[13] D. G. Murray, J. Simsa, A. Klimovic, and I. Indyk, "tf.data: A Machine Learning Data Processing Framework," *Very Large Data Bases (VLDB)*, vol. 14, no. 12, p. 2945–2958, 2021.

[14] J. Mohan, A. Phanishayee, and V. Chidambaram, "CheckFreq: Frequent, Fine-Grained DNN Checkpointing," in *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST 21)*, 2021, pp. 203–216.

[15] Q. Ye, Y. Zhou, M. Shi, and J. Lv, "FLSGD: Free Local SGD with Parallel Synchronization," *The Journal of Supercomputing*, pp. 1–24, 2022.

[16] J. Mohan, A. Phanishayee, J. Kulkarni, and V. Chidambaram, "Synergy: Resource Sensitive DNN Scheduling in Multi-Tenant Clusters," *arXiv preprint*, vol. abs/2110.06073, 2021. [Online]. Available: https://arxiv.org/abs/2110.06073

[17] J. Ren, S. Rajbhandari, R. Y. Aminabadi, O. Ruwase, S. Yang, M. Zhang, D. Li, and Y. He, "ZeRO-Offload: Democratizing Billion-Scale Model Training," in *Proceedings of the 2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 2021, pp. 551–564.

[18] H. Jang, J. Kim, J.-E. Jo, J. Lee, and J. Kim, "MnnFast: A Fast and Scalable System Architecture for Memory-Augmented Neural Networks," in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019, p. 250–263.

[19] J. Liu, H. Zhao, M. A. Ogleari, D. Li, and J. Zhao, "Processing-in-Memory for Energy-Efficient Neural Network Training: A Heterogeneous Approach," in *Proceedings of the 2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 655–668.
[20] D. Kwon, S. Hur, H. Jang, E. Nurvitadhi, and J. Kim, "Scalable Multi-FPGA Acceleration for Large RNNs with Full Parallelism Levels," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, 2020, pp. 1–6.
[21] J. Bae, J. Lee, Y. Jin, S. Son, S. Kim, H. Jang, T. J. Ham, and J. W. Lee, "FlashNeuron: SSD-Enabled Large-Batch Training of Very Deep Neural Networks," in *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST 21)*, 2021, pp. 387–401.
[22] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "GradientBased Learning Applied to Document Recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.
[23] J. J. Hopfield, "Neural Networks and Physical Systems with Emergent Collective Computational Abilities," *National Academy of Sciences*, vol. 79, no. 8, pp. 2554–2558, 1982.
[24] K. Fukushima and S. Miyake, "Neocognitron: A Self-Organizing Neural Network Model for a Mechanism of Visual Pattern Recognition," *Biological Cybernetics*, pp. 267–285, 1982.
[25] Y. LeCun, C. Cortes, and C. J. Burges. (2013) The MNIST Database of Handwritten Digits. [Online]. Available: http://yann.lecun.com/exdb/mnist/
[26] H. Xiao, K. Rasul, and R. Vollgraf. (2017) Fashion-MNIST. [Online]. Available: https://github.com/zalandoresearch/fashion-mnist
[27] G. Cohen, S. Afshar, J. Tapson, and A. van Schaik. (2017) The EMNIST Dataset. [Online]. Available: https://www.nist.gov/itl/products-and-services/emnist-dataset
[28] S. Godard, *sysstat - System Performance Tools For the Linux Operating System*, Paris, France, 1999. [Online]. Available: http://sebastien.godard.pagesperso-orange.fr/download.html
[29] J. Choi, *gpustat*, Ann Arbor, MI, USA, 2016. [Online]. Available: https://github.com/wookayin/gpustat
[30] E. Baek, H. Lee, Y. Kim, and J. Kim, "FlexLearn: Fast and Highly Efficient Brain Simulations Using Flexible On-Chip Learning," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 304–318.
[31] G.-G. Wang, S. Deb, and Z. Cui, "Monarch Butterfly Optimization," *Neural Computing & Applications*, vol. 31, no. 7, pp. 1995–2014, 2019.
[32] G.-G. Wang, "Moth Search Algorithm: A Bio-inspired Metaheuristic Algorithm for Global Optimization Problems," *Memetic Computing*, vol. 10, no. 2, pp. 151–164, 2018.

JINSEO CHOI received the B.S. degree in the Department of Computer Engineering at Changwon National University in South Korea. He is currently pursuing the M.S. degree in the Department of Computer Engineering with Changwon National University in South Korea. His research interests include operating systems, system software, and system for AI.

DONGHYUN KANG is an assistant professor with the Department of Computer Engineering at Changwon National University in South Korea. Before joining Changwon National University, he was an assistant professor with Dongguk University (2019-2020) and a software engineer at Samsung Electronics in South Korea (2018-2019). He received his Ph.D. degree in College of Information and Communication Engineering from Sungkyunkwan University in 2018. His research interests include file and storage systems, operating systems, system for AI, and emerging storage technologies.