TOD: TENSOR-BASED OUTLIER DETECTION, A GENERAL GPU-ACCELERATED FRAMEWORK

Yue Zhao [1] **George H. Chen** [1] **Zhihao Jia** [1]

ABSTRACT

To scale outlier detection (OD) to large, high-dimensional datasets, we propose TOD, a novel system that abstracts OD algorithms into basic tensor operations for efficient GPU acceleration. To make TOD highly efficient in both time and space, we leverage recent advances in deep learning infrastructure in both hardware and software. To deploy large OD applications on GPUs with limited memory, we introduce two key techniques. First, *provable quantization* accelerates OD computation and reduces the required amount of memory by performing specific OD computations in lower precision while provably guaranteeing no accuracy loss. Second, to exploit the aggregated compute resources and memory capacity of multiple GPUs, we introduce *automatic batching*, which decomposes OD computations into small batches that can be executed on multiple GPUs in parallel.

TOD supports a comprehensive set of OD algorithms and utility functions. Extensive evaluation on both real and synthetic OD datasets shows that TOD is on average $11.9\times$ faster than the state-of-the-art comprehensive OD system PyOD (with a maximum speedup of $38.9\times$), and takes less than an hour to detect outliers within a million samples. TOD enables straightforward integration for additional OD algorithms and provides a unified framework for combining classical OD algorithms with deep learning methods. These combinations result in an infinite number of OD methods, many of which are novel and can be easily prototyped in TOD.

1 INTRODUCTION

Outlier detection (OD) is a key machine learning task for identifying data points deviating from a general distribution. OD has numerous real-world applications, including anti-money laundering (Lee et al., 2020), rare disease detection (Li et al., 2018), and network intrusion detection (Lazarevic et al., 2003). Although many new algorithms have been proposed over the years, general OD algorithms face challenges in scaling to large datasets, both in terms of execution time and memory usage. To address these challenges, recent work focuses on both developing distributed OD algorithms on CPUs (Lozano & Acuña, 2005; Angiulli et al., 2010; Oku et al., 2014; Zhao et al., 2021) and using GPUs to accelerate certain OD algorithms (Angiulli et al., 2016; Leal & Gruenwald, 2018). However, these acceleration approaches only target specific (families of) OD algorithms with limited generalization ability. For instance, (Angiulli et al., 2016) shows the usage of GPUs only in distance-based algorithms.

Meanwhile, in the last decade, deep neural networks (DNNs) have revolutionized computer vision, natural language processing, and various other fields (Goodfellow et al., 2016), in

no small part due to the use of modern hardware accelerators (e.g., GPUs) for high-performance tensor operations (Le-Cun, 2019). In this paper, we ask: *can we leverage advances in DNN systems to optimize the performance and scalability of OD applications?* Notably, achieving this goal requires addressing two key challenges.

First, DNN systems provide a *tensor-based* programming interface and represent a DNN model as a combination of tensor operations (e.g., matrix multiplication and convolution). Deploying OD applications on existing DNN systems requires converting a diverse collection of OD algorithms to tensor operations.

Second, DNN systems do not easily support OD computations at scale. Existing DNN systems were designed to handle a small *batch* of training samples in an iteration, even though the full training dataset can be large. However, many OD applications involve performing operations on *all* samples, such as computing distances between all sample pairs. These operations typically do not fit on a single GPU.

In this paper, we present TOD, a **t**ensor-based **o**utlier **d**etection system that abstracts OD applications into tensor operations for efficient GPU acceleration. A key insight behind TOD is leveraging both software and hardware optimizations in DNN systems to enable performant and scalable OD computations. TOD is the first comprehensive

[1]Carnegie Mellon University, Pittsburgh, PA, USA. Correspondence to: Yue Zhao <zhaoy@cmu.edu>, Zhihao Jia <zhihao@cmu.edu>.
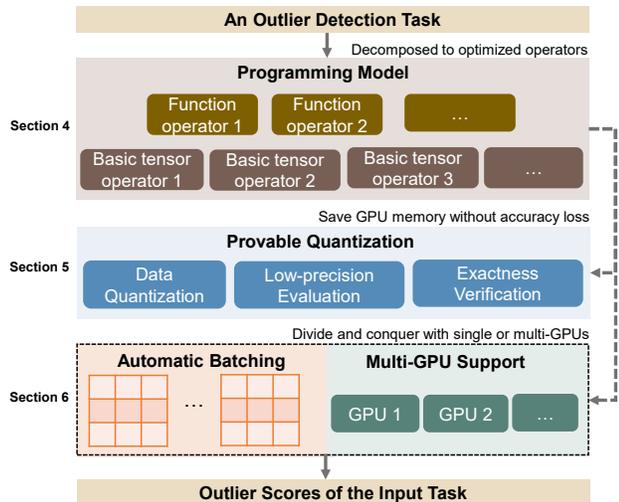
Figure 1. System overview. TOD abstracts OD algorithms into small building blocks for deep optimization, with a set of technique including *provable quantization*, *automatic batching*, and *multi-GPU support*.

GPU-based system for OD. We show an overall architecture of TOD in Fig. 1, which contains three key components:

1. We propose a novel OD programming model that decomposes a complex OD algorithm into a combination of predefined tensor operators, which can directly benefit from GPU acceleration (§4).

2. We introduce *provable quantization* that accelerates OD computation and reduces the memory requirement by performing specific computations in lower precision while provably guaranteeing no accuracy loss. (§5).

3. To overcome the resource limits of a single GPU, *automatic batching* and *multi-GPU support* decompose OD computations into small parallelizable batches, which are executed on multiple GPUs in parallel in a pipeline fashion (§6).

In extensive experiments, we compare TOD against a state-of-the-art comprehensive OD system PyOD (Zhao et al., 2019b), and also show the benefits of our proposed acceleration components (provable quantization, automatic batching, multi-GPU support). Our main experimental finding is that for the datasets we considered, TOD is on average 11.9 times faster than PyOD and can handle a million samples within an hour.

Extensibility and integration. TOD is open source and enables easy development of new OD algorithms by leveraging heavily optimized tensor operators. In addition, TOD provides a unified framework for combining classical OD algorithms and deep learning methods. This combination yields a large number of yet-to-be-discovered OD methods. Thus, our system is also a platform that enables rapid research and development of novel OD methods.

Table 1. Key OD algorithms and their time and space complexity with a brute-force implementation, where n is the number of samples, and d is the number of dimensions. Note that ensemble-based methods' complexities depend on the underlying base estimators. Algorithms that can be accelerated in TOD are marked with ✓.

Category	Algorithm	Time Compl.	Space Compl.	Optimized in TOD
Proximity	*k*NNOD	$O(n^2)$	$O(n^2)$	✓
Proximity	COF	$O(n^3)$	$O(n^2)$	✓
Proximity	LOF	$O(n^2)$	$O(n^2)$	✓
Proximity	LOCI	$O(n^2)$	$O(n^2)$	✓
Statistical	KDE	$O(n^3)$	$O(n^2)$	✗
Statistical	HBOS	$O(nd)$	$O(n)$	✓
Statistical	COPOD	$O(nd)$	$O(n)$	✓
Ensemble	LODA	N/A	N/A	✓
Ensemble	FB	N/A	N/A	✓
Ensemble	LSCP	N/A	N/A	✓
Linear	PCA	$O(nd)$	$O(n)$	✓
Linear	OCSVM	$O(n^3)$	$O(n^2)$	✗

2 BACKGROUND AND RELATED WORK

We now provide some background on existing OD algorithms and which ones are accelerated by TOD (§2.1), on the DNN infrastructure that TOD builds off of to do acceleration (§2.2), and on existing comprehensive OD systems (§2.3), including the current state-of-the-art PyOD (Zhao et al., 2019b). In our experiments later, we benchmark TOD against PyOD.

2.1 Existing OD Algorithms and Scalability

Outlier detection (also known as anomaly detection) is a key machine learning task that aims to find data points that deviate from a general distribution (Aggarwal, 2015). As shown in Table 1, key OD algorithms can be grouped into four categories: (*i*) proximity-based algorithms that rely on measuring sample similarity including *k*NNOD (Angiulli & Pizzuti, 2002), ABOD (Kriegel et al., 2008), COF (Tang et al., 2002), LOF (Breunig et al., 2000b), and LOCI (Papadimitriou et al., 2003); (*ii*) statistical approaches including KDE (Schubert et al., 2014), HBOS (Goldstein & Dengel, 2012), and COPOD (Li et al., 2020); (*iii*) ensemble-based methods that build a collection of detectors for aggregation like LODA (Pevný, 2016), and LCSP (Zhao et al., 2019a); and (*iv*) linear models such as PCA (Shyu et al., 2003).

Importantly, many OD algorithms suffer from scalability issues (Zhao et al., 2021). For example, Table 1 shows that various proximity-based OD algorithms have time and space complexities that are at least $O(n^2)$—they all require estimating and storing pairwise distances among all n samples. The high time complexities of many OD algorithms limit their applicability in real-world applications that require either real-time responses (e.g., financial fraud detection) or the concurrent processing of millions of samples. As

shown in Table 1, TOD optimizes nearly all the existing OD algorithms mentioned.

2.2 DNN Infrastructure and Acceleration

Deep neural networks have dramatically improved the accuracy of artificial intelligence systems across numerous fields (Goodfellow et al., 2016). Its success is fueled by recent advances in both hardware and software (LeCun, 2019). Specifically, DNN systems depend on tensor operations that can often be parallelized and executed in small batches. These operations are well-suited for GPUs, especially as a single GPU nowadays often has many more cores than a single CPU; while GPU cores are not as general purpose as CPU cores, they suffice in executing the tensor operations of deep learning. Moreover, the maturity of DNN programming interfaces such as PyTorch (Paszke et al., 2019) and TensorFlow (Abadi et al., 2016) makes developing machine learning models easy with a wide range of GPUs.

With the above considerations, we build TOD using the DNN ecosystem, taking advantage of its established hardware acceleration and software accessibility. This design choice also opens the opportunity for unifying classical OD algorithms (see §2.1) and DNN-based OD algorithms on the same platform—this emerging direction has gained increasing attention in OD research (Ruff et al., 2019).

2.3 Outlier Detection Systems

Over the years, comprehensive OD systems on CPUs that cover a diverse group of algorithms have been developed in different programming languages, including ELKI Data Mining (Achtert et al., 2010) and RapidMiner (Hofmann & Klinkenberg, 2013) in Java, and PyOD (Zhao et al., 2019b) in Python. Among these, PyOD is the state-of-the-art (SOTA) with deep optimization including just-in-time compilation and parallelization. It is widely used by both academia and industry works, with hundreds of citations (Zhao, 2021) and millions of downloads per year (Sincraian, 2021). Recently, the PyOD team proposed an acceleration system called SUOD to further speed up the training and prediction in PyOD with a large collection of heterogeneous OD models (Zhao et al., 2021). Specifically, SUOD uses algorithmic approximation and efficient parallelization to reduce the computational cost and therefore runtime. There are also other parallel systems designed for specific (family of) OD algorithms—Parallel Bay and Parallel LOF can scale up proximity-based OD algorithms in multi-core computer systems (Lozano & Acuña, 2005; Oku et al., 2014).

Other than improving system efficiency on CPUs, there are efforts to use GPUs for fast OD calculations (Angiulli et al., 2016; Leal & Gruenwald, 2018). These approaches rely on exploring the characteristics of a specific OD algorithm for GPU acceleration. This limits their generalization to a wide collection of OD algorithms. Furthermore, none of them has multi-GPU support, leading to limited scalability. To our best knowledge, there is no existing comprehensive GPU-based OD system. Therefore, we consider the SOTA comprehensive system, PyOD, as the primary baseline.

3 OVERVIEW

3.1 Definition and Problem Formulation

A comprehensive OD system implements a collection of OD models $\mathcal{M} = \{M_1; ...; M_m\}$ such that given a user-specified OD model $M \in \mathcal{M}$ and input data $\mathbf{X} \in \mathbb{R}^{n \ d}$ without ground truth labels (rows of $\mathbf{X}$ index data points, while columns index features), the system outputs outlier scores $\mathbf{O} := M(\mathbf{X}) \in \mathbb{R}^n$, which should roughly be deterministic irrespective of the underlying system (higher values in $\mathbf{O}$ correspond to data points that are more likely to be considered outliers; thresholding on outlier scores can be used to determine which points are outliers). Given the hardware configuration $\mathcal{C}$, e.g., CPU, RAM, and GPU (if available), the system performance can be measured in *efficiency* (both time and space complexities).

3.2 System Overview

As a reminder from Section 1, TOD is a comprehensive OD system as outlined in Fig. 1. For an outlier detection task, TOD abstracts it into a combination of predefined tensor operators via the proposed *programming model* for direct GPU acceleration (§4). Notably, the predefined tensor operators are optimized with *provable quantization* that achieves faster computation and smaller memory usage while provably maintaining operator accuracy (§5). To overcome the limit of a single GPU, we further introduce *automatic batching* and *multi-GPU* support in TOD (§6).

4 PROGRAMMING MODEL

Motivation. As a comprehensive system, TOD aims to include a diverse collection of OD algorithms, including proximity-based methods, statistical methods, and more (see §2.1). However, not all algorithms can be directly converted into tensor operations for GPU acceleration. A key design goal of our programming model is to allow for many OD algorithms to be implemented by piecing together some basic commonly recurring building blocks. In particular, rather than manually implementing many OD algorithms separately, which is a labor-intensive process, we instead define OD algorithms in terms of basic building blocks that each just needs to be implemented once. Moreover, the building blocks can be optimized independently.
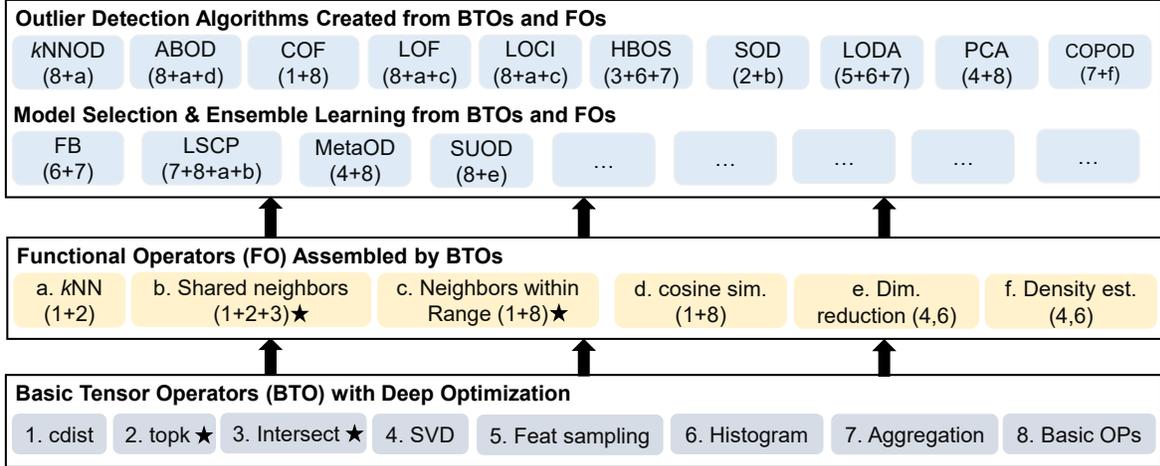
Outlier Detection Algorithms Created from BTOs and FOs

| kNNOD (8+a) | ABOD (8+a+d) | COF (1+8) | LOF (8+a+c) | LOCI (8+a+c) | HBOS (3+6+7) | SOD (2+b) | LODA (5+6+7) | PCA (4+8) | COPOD (7+f) |

Model Selection & Ensemble Learning from BTOs and FOs

| FB (6+7) | LSCP (7+8+a+b) | MetaOD (4+8) | SUOD (8+e) | ... | ... | ... | ... | ... |

Functional Operators (FO) Assembled by BTOs

| a. kNN (1+2) | b. Shared neighbors (1+2+3)★ | c. Neighbors within Range (1+8)★ | d. cosine sim. (1+8) | e. Dim. reduction (4,6) | f. Density est. (4,6) |

Basic Tensor Operators (BTO) with Deep Optimization

| 1. cdist | 2. topk ★ | 3. Intersect ★ | 4. SVD | 5. Feat sampling | 6. Histogram | 7. Aggregation | 8. Basic OPs |

Figure 2. With algorithmic abstraction, more than 20 OD algorithms (denoted by ▢) are abstracted into 8 basic tensor operators (in ▢) and 6 functional operators (in ▢). This drastically reduces the implementation and optimization effort, and opens the possibility of including new algorithms easily. Additionally, the operators marked with ⊦ are applicable to provable quantization (see §5).

4.1 Algorithmic Abstraction

The key idea of our programming model is to decompose existing OD algorithms into a set of low-level *basic tensor operators* (BTOs), which can directly benefit from GPU acceleration. On top of these BTOs, we introduce higher-level OD operators called *functional operators* (FOs) with richer semantics. Consequently, OD algorithms can be constructed as combinations of BTOs and FOs.

Fig. 2 shows the hierarchy of TOD's programming abstraction in a bottom-up way, with increasing dependency: 8 BTOs are first constructed as the foundation of TOD (shown at the bottom in gray), while 6 FOs are then created on top of them (shown in the middle). Finally, OD algorithms and key functions on the top of the figure can be assembled by BTOs and FOs. In other words, TOD uses a tree-structured dependency graph, where the BTOs (as leaves of the tree) are fully independent for deep optimization, and all the algorithms depending on these BTOs can be collectively optimized. Clearly, this abstraction process reduces the repetitive implementation and optimization effort, and improves system efficiency and generalization. Additionally, it also facilities fast prototyping and experimenting with new algorithms.

4.2 Building Complete Algorithms

It is easy to build an end-to-end OD application by constructing its *computation graph* using FOs and BTOs. For instance, angle-based outlier detection (ABOD) is a classical OD algorithm (Kriegel et al., 2008), where each sample's outlier score is calculated as the average cosine similarity of its neighbors. Fig. 3(a) highlights an implementation of ABOD that uses an FO called kNN to obtain a list of neighbors for each sample and then applies another FO called cosine sim. for calculating cosine similarity. Note that
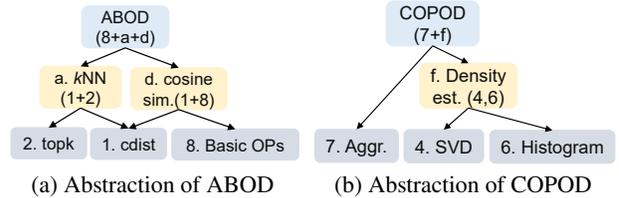

(a) Abstraction of ABOD   (b) Abstraction of COPOD

Figure 3. Examples of building complex OD algorithms with FO and BTO conveniently.

the FOs are also built as combinations of BTOs. For example, kNN is implemented as calculating pairwise sample distance using cdist and then identifying the k "neighbors" with smallest distances using topk. Additionally, Fig. 3(b) shows the computation graph of copula-based outlier detection (COPOD) (Li et al., 2020).

5 PROVABLE QUANTIZATION

Motivation. OD operators mainly depend on floating-point operations, namely $\{+, -, \times, =\}$. For these operations, the main source of the imprecision is rounding (Lee et al., 2018). Obviously, rounding errors increase when storing numbers using fewer bits, e.g., 16-bit precision leads to more inaccuracy than 64-bit precision. Many machine learning algorithms therefore use high-precision floating-points when possible to minimize the impact of the rounding errors. However, working with high-precision floating-point numbers can increase computation time and storage cost. This is especially critical for GPU systems with limited memory.

To reduce memory usage and runtime, *quantization* has been applied to many machine learning algorithms (Jain et al., 2020; Daghaghi et al., 2021). Simply put, it refers to execute an operator (function) with lower-precision floating

representations. If we denote the original function by $r(x)$ and its quantization by $r_q(x)$, the rounding error $\text{Err}(\cdot)$ of quantization is defined as the output difference between $r(x)$ and $r_q(x)$, namely $\text{Err}(r_q(x)) = r(x) - r_q(x)$. Intuitively, quantization can save memory at the cost of accuracy. How to balance the tradeoff between the *memory cost* and *algorithm accuracy* is a key challenge for quantizing machine learning (Dai et al., 2021).

The key idea we use is that depending on the operator used, it is possible to apply quantization with *no loss in accuracy*. As a motivating example, consider the sign function $r(x)$ that returns "+" if $x > 0$ and returns "−" otherwise. Clearly, even if we quantize x to have a single bit (that precisely indicates sign), we can achieve an exact answer for $r(x)$ that is the same as if instead x had more bits. Building on this intuition, we introduce *provable quantization* for a collection of OD operators, where the output of the operators remain provably unchanged before and after quantization. Provable quantization relies on a standard analysis technique for floating-point numbers called the "$(1 +)$-property" (e.g., Lee et al. 2018), which we define next.

5.1 $(1 +)$-property for Rounding Error Calculation

Let $\mathbb{F}$ denote the set of 64-bit floating-point numbers. For $x, y \in \mathbb{F}$, we define the floating-point operation "$\sim$" as $x \sim y \, , \quad \text{fl}(x * y)$, where $* \in \{+, −, \times, =\}$ and $\text{fl}(\cdot)$ refers to the IEEE 754 standard for rounding a real number to a 64-bit floating-point number (Kahan, 1996). For example, $\oplus$ is floating-point addition and $\otimes$ is floating-point multiplication. The standard technique for calculating the rounding errors in floating-point operations is the $(1 +)$-property (Lee et al., 2018), which is formally defined as follows.

Theorem 1 (Theorem 3.2 of Lee et al. 2018) *Let* $x, y \in \mathbb{F}$, *and* $* \in \{+, −, \times, =\}$. *Suppose that* $|x * y| \leq \max \mathbb{F}$. *Then when we compute* $x * y$ *in floating-point, there exist multiplicative and additive error terms* $\in \mathbb{R}$ *and* $^0 \in \mathbb{R}$ *respectively such that*

$$x \sim y = (x * y)(1 +) + {}^0, \quad \text{where } | | \leq \; ; | {}^0| \leq {}^0 \quad (1)$$

In the above equation, *and* ^{0} *are constants that do not depend on* x *or* y. *For instance, when working with 64-bit floating-point numbers,* $= 2^{-53}$ *and* $^0 = 2^{-1075}$.

As discussed by Lee et al. (2018, Section 5), this property can be further simplified when the exact result of the floating operation is not in the so-called "subnormal" range: the addictive error term ^{0} can be soundly removed, leading to a simplified $(1 +)$-property:

$$x \sim y = (x * y)(1 +); \quad \text{where } | | \leq \; : \quad (2)$$

5.2 Provable Quantization in TOD

TOD applies provable quantization for an applicable opera-

tor $r(\cdot)$ with input x in three steps: (*i*) input quantization (*ii*) low-precision evaluation and (*iii*) exactness verification. In a nutshell, the input is first quantized into lower precision, and then the operator is evaluated in the lower precision, where $r(\cdot)$ is evaluated as $r_q(x)$. To verify the exactness of the quantization, we calculate the rounding error $\text{Err}(r_q(x))$ by the simplified $(1 +)$-property in eq. (2) and then check whether the result of $r(\cdot)$ may change with the rounding error. If it passes the verification, then we use quantization; otherwise, we use the original precision of x.

Remark. Not all the operators are applicable for provable quantization—they need meeting the following criteria: (1) the output values are not in a floating-point representation and (2) the performance gain in low-precision evaluation is larger than the overhead of verification. Based on these criterion, we mark the operators that are generally applicable for provable quantization by $\mathbb{F}$ in Fig. 2.

5.3 Case Study: Neighbors Within Range

We show the usage of provable quantization in TOD on neighbors within range (NWR, one of the FOs of our programming model), a common step in many OD algorithms, e.g., LOF (Breunig et al., 2000a) and LOCI (Papadimitriou et al., 2003). NWR identifies nearest neighbors within a preset distance threshold (usually a small number), which may be considered as a variant of k nearest neighbors. More formally, given an input tensor $\mathbf{X} := [X_1, X_2, ..., X_n] \in \mathbb{R}^{n \times d}$ (n samples and d dimensions) and the distance threshold , NWR first calculates the pairwise distance among each sample via the $\texttt{cdist}$ operator, yielding a distance matrix $\mathbf{D} \in \mathbb{R}^{n \times n}$, where $\mathbf{D}_{i,j}$ is the pairwise distance between X_i and X_j. Then, each pairwise distance in $\mathbf{D}$ is compared with , and NWR outputs the indices of samples where $\mathbf{D}_{i,j} \leq$. As the pairwise distance calculation in NWR requires $O(n^2)$ space, provable quantization can provide significant GPU memory savings.

NWR meets the criterion we outline in §5.2, where eq. (2) can be applied to estimate the rounding errors. Recall that the pairwise Euclidean distance between two samples is

$$\mathbf{D}_{ij} = \|X_i - X_j\|_2^2 = \|X_i\|_2^2 + \|X_j\|_2^2 - 2X_i^T X_j : \quad (3)$$

We shall compute this in floating-point. Importantly, in our analysis to follow, the ordering of floating-point operations matters in determining rounding errors. To this end, we calculate distance using the right-most expression in eq. (3) (note that we do *not* first compute the difference $X_i - X_j$ and then compute its squared Euclidean norm). When calculating the first term in the RHS of eq. (3) via floating-

operations, we get

$$\mathrm{fl}(\|X_i\|_2^2)$$
$$= (X_{i;1} \otimes X_{i;1}) \oplus (X_{i;2} \otimes X_{i;2}) \oplus \cdots \oplus (X_{i;d} \otimes X_{i;d})$$
$$= (X_{i;1}^2(1+\epsilon_1)) \oplus (X_{i;2}^2(1+\epsilon_2)) \oplus \cdots \oplus (X_{i;d}^2(1+\epsilon_d));$$

where the second equality uses eq. (2) and we note that the errors $\epsilon_1;\ldots;\epsilon_d$ across the floating-point multiplications (for squaring) need not be the same (in fact, these need not be the same across samples $i = 1;2;\ldots;n$ but we omit this indexing to keep the equation from getting cluttered).

Next, by defining $x_{\max} \coloneqq \max_{i \in \{1;\ldots;n\};k \in \{1;\ldots;d\}} |X_{i;k}|$ and recalling from Theorem 1 that each of $\epsilon_1;\epsilon_2;\ldots;\epsilon_d$ above is at most ϵ, we get

$$\mathrm{fl}(\|X_i\|_2^2)$$
$$= (X_{i;1}^2(1+\epsilon_1)) \oplus (X_{i;2}^2(1+\epsilon_2)) \oplus \cdots \oplus (X_{i;d}^2(1+\epsilon_d))$$
$$\leq \underbrace{[x_{\max}^2(1+\epsilon)] \oplus [x_{\max}^2(1+\epsilon)] \oplus \cdots \oplus [x_{\max}^2(1+\epsilon)]}_{d \text{ terms added via floating-point addition}}$$
$$\leq d \cdot x_{\max}^2 (1+\epsilon)^{1+\lceil \log_2 d \rceil};$$

where for the last step, $\log_2 d$ shows up since summation of d elements in lower-level programming languages is implemented in a divide-and-conquer manner that reduces to $\lceil \log_2 d \rceil$ operations (there is still a "1+" term in the exponent for the floating-point multiplication/squaring). The rounding error can be bounded as follows:

$$\mathrm{Err}(\|X_i\|_2^2) = \|X_i\|_2^2 - \mathrm{fl}(\|X_i\|_2^2)$$
$$\leq d \cdot x_{\max}^2 - \mathrm{fl}(\|X_i\|_2^2)$$
$$\leq |d \cdot x_{\max}^2 - \mathrm{fl}(\|X_i\|_2^2)|$$
$$\leq d \cdot x_{\max}^2 [(1+\epsilon)^{1+\lceil \log_2 d \rceil} - 1]:$$

This same analysis can be used to bound the floating-point errors of the other terms in eq. (3). Overall, we get

$$\mathrm{Err}(D_{ij}) \leq 4d \cdot x_{\max}^2 [(1+\epsilon)^{1+\lceil \log_2 d \rceil + 2} - 1]; \quad (4)$$

where the "+2" shows up in the exponent due to the addition and subtraction in the RHS of (3) that we compute in floating-point.

Inequality (4) provides a numerical way for checking whether a single entry D_{ij} is within the range of ϵ as $|D_{ij} - \epsilon| > \mathrm{Err}(D_{ij})$. More conveniently, we could scale the input X into the range of $[0;1]$ before the distance calculation (Raschka, 2015), so that $x_{\max} \leq 1$ and the implementation complexity can be further reduced. With this treatment, a large amount of GPU memory can be saved in NWR operations (see §7.5 for results).
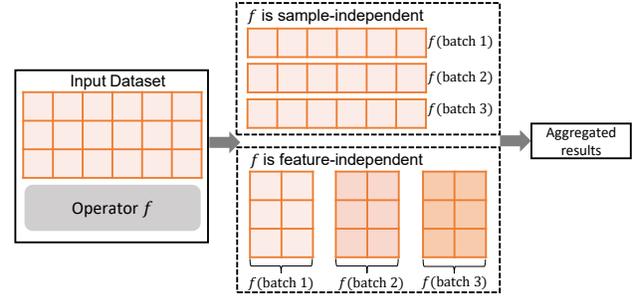

Figure 4. Direct batching with independence assumption create batches along the sample or feature index.

6 AUTOMATIC BATCHING AND MULTI-GPU SUPPORT

Motivation. Unlike CPU servers that can easily be extended to have terabytes of memory, GPU systems currently face much more stringent memory limitations—modern GPUs are mostly equipped with between 4 and 40 GB of memory (Meng et al., 2017; Svedin et al., 2021). Out-of-memory (OOM) errors have thus become common in GPU-based systems for machine learning tasks (Gao et al., 2020). To overcome this, we design an automatic batching mechanism to execute memory-intensive operators in small batches (other than on the full input dataset) as an optimization method for BTOs. With the batching mechanism, we also make TOD support single- and multi-GPU environments. Importantly, the complexity of batching an operator is subject to its dependency structure (Neubig et al., 2017; Looks et al., 2017). Intuitively, operators whose computations require all training samples and features are said to have "global dependency" and are harder to batch.

Operators without global dependency. An operator can be directly executed in small batches when it is either: (*i*) *sample-independent*: the estimation of each sample is independent, or (*ii*) *feature-independent*: the contribution of each feature is independent. Fig. 4 shows that when either of these conditions holds, the computations of an operator can be directly partitioned into multiple batches by splitting along the sample or feature dimension, and the operator is then executed on the batches for final aggregation. For instance, $\mathtt{topk}$ is a sample-independent operator. For an input tensor $X \in \mathbb{R}^{n \times d}$, $\mathtt{topk}$ outputs the indices of the largest k values for each sample (i.e., row), resulting in an output tensor $I_X \coloneqq \mathtt{topk}(X;k) \in \mathbb{R}^{n \times k}$. Therefore, we could split X into batches with full features, each with b samples (i.e., $\{X_1;X_2;\ldots\} \in \mathbb{R}^{b \times d}$). As another example, $\mathtt{Histogram}$ outputs the frequencies of each feature's values, which is feature-independent. Thus, we could partition X into blocks of b features, e.g., $\{X_1;X_2;\ldots\} \in \mathbb{R}^{n \times b}$.

Operators with global dependency. Not all operators satisfy the independence structures discussed above. For ex-
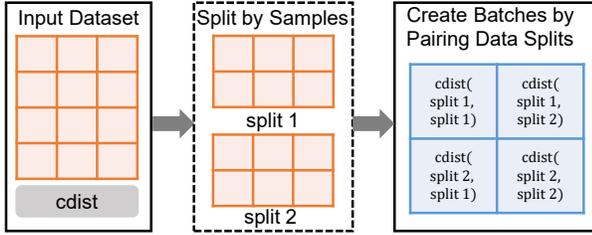
*Figure 5.* Customized batching solution for cdist in TOD.

ample, SVD performs singular value decomposition. It is neither sample- nor feature-independent and cannot be directly batched.

Using our algorithmic abstraction (see §4), TOD provides customized batching solutions for a small number of BTOs with global dependency. For instance, as shown in Fig. 5, TOD uses the approach introduced in Neeb & Kurrus (2016) to batch cdist, where an input dataset is first split by samples, and then TOD calculates the pairwise distance of *each pairs of split* in batches.

6.1 Sequential Batching and Operator Fusion

Simple concatenation. Since BTOs are independent from each other, executing a sequence of BTOs in batches is straightforward, i.e., simply feed the output of a batch operator as an input to another one. For example, kNN (see Fig. 2) finds the k nearest neighbors by first calculating pairwise distances of input samples via the cdist BTO and then returns the index of k items with the smallest distance via the topk BTO. Therefore, batching for kNN is achieved by running cdist and topk sequentially, where each uses automatic batching and the output of the former is the input of the latter.

Operator fusion. Although the simple concatenation discussed above is straightforward, a closer look unlocks deeper optimization opportunities in automatic batching with a sequence of operators. Notably, the output of kNN is the indices of the k nearest neighbors of an input dataset, where the pairwise distance generated by cdist is only used in an intermediate step but not returned. If we could prevent moving this large distance matrix between operators (i.e., cdist and topk), space efficiency can be improved. In deep learning systems, *operator fusion* is a common optimization technique to fuse multiple operators into a single one in a computational graph (Neubig et al., 2017; Looks et al., 2017; Wang et al., 2021). Inspired by this, TOD opportunistically fuses a sequence of operators in automatic batching whenever possible. Fig. 6 compares simple concatenation (subfigure a) and operator fusion (subfigure b) on kNN. Specifically, the latter executes the topk BTO on the cdist BTO's individual batches separately rather than running topk on the full distance matrix outputted by cdist. Note that the global k nearest neighbors (of the full
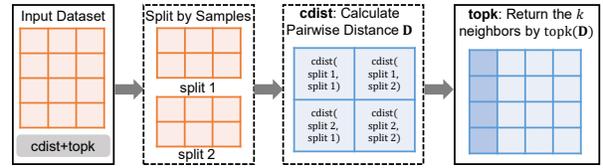
(a) Simple concatenation: topk is invoked on the full result of cdist—we need to move the distance matrix **D**.
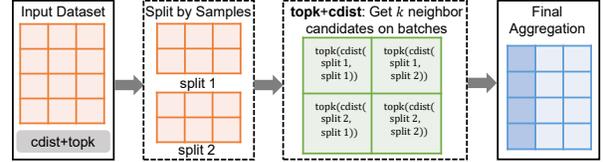
(b) Operator fusion: topk is directly invoked on the batch result of cdist, preventing the move of distance matrix **D**.

Figure 6. The comparison of automatic batching for kNN between simple concatenation and operator fusion. The latter has better scalability by not creating and moving large distance matrix **D**

dataset) can be identified from the k local neighbors candidates from batches in the final aggregation, so the result is still exact. This prevents moving the entire $n \times n$ distance matrix between operators, which often causes OOM.

6.2 Multi-GPU Support

With automatic batching, we address the memory limitation of a single GPU by making TOD compatible with multiple GPUs, an idea that has shown significant efficiency improvements in applications like graph neural networks (Jia et al., 2020). Intuitively, if there is only one GPU, TOD iterates across multiple batches sequentially and aggregates the results. When multiple GPUs are available, we could achieve better performance by executing OD computations concurrently on multiple GPUs. Specifically, TOD first applies automatic batching to an underlying task—multiple subtasks are created and assigned to available GPUs. TOD creates a *subprocess* for each available GPU to execute the assigned subtasks and a *shared global container* to store the results returned from each GPU. Since we equally distribute subtasks across GPUs, we deem the runtime of each GPU is close. Once all the subtasks are complete, the final output is generated by aggregating the results in the global container. For example, automatically batching cdist in Fig. 5 leads to 4 subtasks, each of which calculates the pairwise distances for a pair of splits (denoted as blue blocks in the figure). Each of the four available GPUs executes an assigned subtask and sends the cdist results to the global container. Finally, the cdist result on the entire dataset is obtained by aggregating the intermediate results in the global container.

Note that the multi-GPU execution is at the operator level (e.g., cdist). To prevent the overhead caused by creating subprocesses repetitively, TOD leverages the operator

(a) fashion-mnist (60,000 samples with 3072 features)

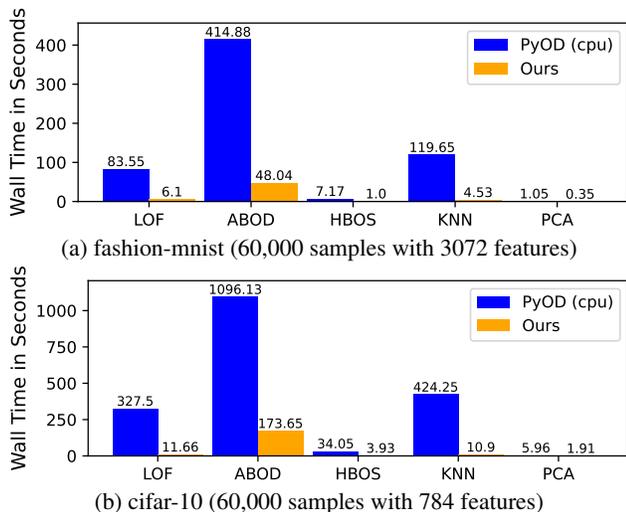(b) cifar-10 (60,000 samples with 784 features)

Figure 7. Runtime comparison between PyOD and TOD in seconds. See Appendix Fig. B.1 on additional results on synthetic datasets. Notably, TOD outperforms PyOD in all experiments.

fusion descried in §6.1 whenever possible.

7 EXPERIMENTAL EVALUATION

Our experiments answer the following questions:

- Is TOD more efficient (in time and space) than the SOTA OD system (i.e., PyOD)? (§7.3 & 7.4)
- How effective is each technique (e.g., provable quantization and automatic batching)? (§7.5 & 7.6)
- How much performance gain can TOD achieve on the multiple GPUs? (§7.7)

7.1 Implementation and Experiment Environment

TOD is implemented on top of PyTorch (Paszke et al., 2019). We extended PyTorch in the following aspects to support efficient OD. First, we implemented a set of BTOs and FOs (see Fig. 2) for fast tensor operations in OD. Second, for operators that support batching (as described in §6), we created batched versions of them to improve scalability. Additionally, we enabled specialized multi-GPU support in TOD by leveraging PyTorch's `multiprocessing`.

All the experiments were performed on an Amazon EC2 cluster with an Intel Xeon E5-2686 v4 CPU, 61GB DRAM, and an NVIDIA Tesla V100 GPU. For the multi-GPU support evaluation, we extend it to multiple NVIDIA Tesla V100 GPUs with the same CPU node.

7.2 Datasets, Baselines, and Evaluation Metrics

Datasets. Appendix Table A.1 shows the six real-world benchmark datasets used in this study, which are widely evaluated in OD research (Zhao et al., 2019a; Ruff et al.,

2019; Li et al., 2020; Zhao et al., 2021). Given the limited size of real-world OD datasets, we use the data generation function in PyOD to create larger synthetic datasets to evaluate the scalability of TOD.

Baselines. As discussed in §2, there is no existing GPU system that covers a diverse group of OD algorithms. Therefore, we use the SOTA CPU-based OD system PyOD (Zhao et al., 2019b) as a baseline, which is deeply optimized with just-in-time compilation and parallelization. Additionally, we compare to a direct implementation in PyTorch when appropriate (as a GPU baseline).

OD algorithms and operators. Throughout the experiments, we compare the performance of five representative OD algorithms across different systems (see §2.1): proximity-based algorithms including LOF, ABOD, and *k*NNOD; statistical method HBOS, and linear model PCA. We also provide an operator level analysis to demonstrate the effectiveness of certain techniques.

Evaluation metrics. Since TOD and the baselines do not involve any approximation, the output results are exact and consistent across systems. Therefore, we omit the accuracy evaluation, and compare the wall-clock time and GPU memory consumption as measures of time and space efficiency.

7.3 End-to-end Speed Evaluation: TOD vs PyOD

We first present the runtime comparison between TOD and PyOD using two real datasets (fashion-mnist and cifar-10; see Fig. 7) and two larger synthetic datasets (see Appendix Fig. B.1). The results show that TOD is on average 11.90× faster than PyOD on the five benchmarks (14.80×, 21.05×, 7.43×, 11.21×, and 5.04× speed-up on LOF, *k*NNOD, ABOD, HBOS, and PCA, respectively). For proximity-based algorithms, a larger speed-up is observed for datasets with a higher number of dimensions: LOF and *k*NNOD are 28.09× and 38.93× faster on cifar-10 with 3,072 features. This is expected as GPUs are well-suited for dense tensor multiplication, which is essential in proximity-based methods. Separately, a larger improvement can be achieved for HBOS and PCA on datasets with larger sample sizes. For instance, HBOS is 11.83× faster on Synthetic 1 (100,000 samples), while the speedup is 17.16× on Synthetic 2 (200,000 samples). This is expected as HBOS treats each feature independently for density estimation on GPUs, so a large number of samples with a small number of features should yield a large speed-up. In summary, all the OD algorithms tested are significantly faster in TOD than in the SOTA PyOD system, with the precise amount of speed improvement varying across algorithms.
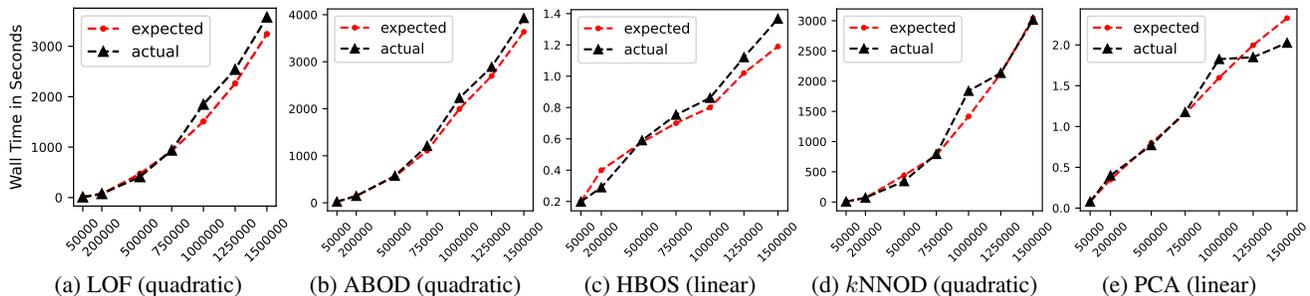
Figure 8. Scalability plot of the selected algorithms in TOD, where the actual runtime (in black) is close to the expected runtime (in red).
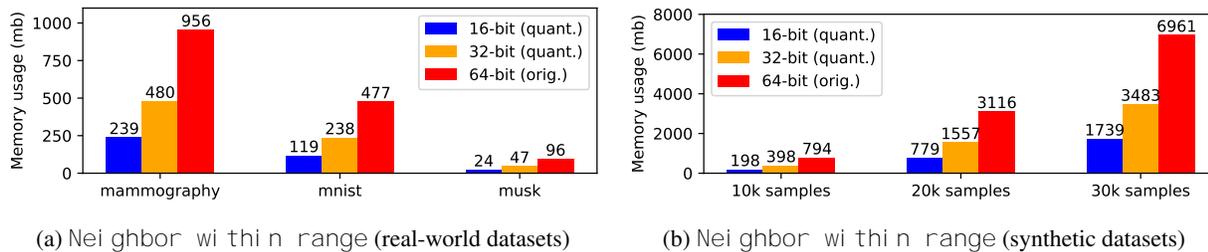

Figure 9. GPU memory consumption comparison between using provable quantization (16-bit and 32-bit) and the full precision (64-bit). Clearly, provable quantization leads to significant memory consumption saving on `nwr` and `topk` (see Appendix Fig .).

7.4 Scalability of TOD

We now gauge the scalability of TOD on datasets of varying sizes, including ones larger than fashion-mnist and cifar-10. In Fig. 8, we plot TOD's runtime with five OD algorithms on the synthetic datasets with sample sizes ranging from 50,000 to 1,500,000 (all with 200 features). To the best of our knowledge, none of the existing OD systems can handle datasets with more than a million samples within a reasonable amount of time (Aggarwal, 2015; Zhao et al., 2021), as most of the OD algorithms are associated with quadratic time complexity. Fig. 8 shows that the five OD algorithms in TOD tested can process million-sample datasets within an hour, and the actual runtime is close to the expected runtime. Thus, TOD provides a scalable approach for enabling OD in many real-world applications.

7.5 Provable Quantization

As discussed in §5, provable quantization in TOD can optimize the operator memory usage while provably preserving correctness (i.e., no accuracy degradation). To demonstrate its effectiveness, we compare the GPU memory consumption of two applicable operators, `neighbor within range (nwr)` and `topk`, with and without provable quantization. Multiple real-world and synthetic datasets are used in the comparison (see Appendix Table A.1), where synthetic datasets' names, such as "10k" and "1M", denote their sample sizes. We deem the 64-bit floating-point precision as the ground truth, and evaluate the provable

quantization results in 32- and 16-bit floating-point.

The results demonstrate that provable quantization always leads to memory savings. Specifically, Fig. 9 shows `nwr` with provable quantization on average saves 71.27% (with 16-bit precision) and 47.59% (with 32-bit precision) of the full 64-bit precision GPU memory. Similarly, Fig. B.2 in Appendix shows that `topk` with provable quantization saves 73.49% (with 16-bit precision) and 49.58% (with 32-bit precision) of the full precision memory. Regarding the runtime comparison, operating in lower precision may also lead to an edge. Table 2 shows the operator runtime comparison between using provable quantization (in 16-bit and 32-bit precision) and using the full 64-bit precision. It shows that provable quantization in 16-bit precision is faster than the computations in full precision in most cases. This empirical finding can be attributed to lower-precision operations typically being faster, and this speed improvement outweighing the overhead of post verification. Note that 32-bit provable quantization often takes the longest time—it is slower than 16-bit provable quantization in operation, and the post verification overhead makes it slower than 64-bit operation. These results demonstrate the effectiveness of using provable quantization in lower precision, especially in 16-bit, leading to both memory and computation time savings.

7.6 Automatic Batching

To evaluate the effectiveness of automatic batching, we compare the runtime of multiple BOs and FOs under (*i*) an

Table 2. Comparison of operator runtime (in seconds) with provable quantization (i.e., 16-bit and 32-bit) and without quantization (i.e., 64-bit). The best model is highlighted in bold (per column), where provable quantization in 16-bit outperform in most cases.

	nwr						topk					
Prec.	mammog.	mnist	musk	10k	20k	30k	cifar-10	f-mnist	speech	1M	2M	5M
16-bit	2.66	**0.54**	**0.06**	**0.87**	**3.02**	**6.56**	0.31	0.09	0.0054	**0.58**	**1.16**	**2.90**
32-bit	2.92	1.51	0.09	2.57	10.09	22.51	0.32	0.1	0.0048	0.70	1.40	3.52
64-bit	**2.59**	1.22	0.09	2.09	8.21	18.52	**0.26**	**0.07**	**0.0024**	**0.58**	1.17	2.93

Table 3. Operator runtime comparison among implementations in NumPy (no batching), PyTorch (no batching) and TOD (with automatic batching); the most efficient result is highlighted in bold per row. Automatic batching in TOD prevents out of memory (OOM) errors yet shows great efficiency.

Operator	Size	NumPy	PyTorch	TOD
topk	10,000,000	7.88	**1.08**	1.09
topk	20,000,000	15.77	OOM	**2.44**
topk	100,000,000	OOM	OOM	**10.83**
intersect	20,000,000	1.99	**0.12**	0.14
intersect	100,000,000	11	**0.63**	**0.63**
intersect	200,000,000	21.65	OOM	**2.11**
kNN	50,000	20.15	**0.27**	0.28
kNN	200,000	194.43	OOM	**19.65**
kNN	400,000	818.32	OOM	**71.22**

Numpy implementation on a CPU (Harris et al., 2020), (*ii*) a direct PyTorch implementation without batching (Paszke et al., 2019), and (*iii*) TOD's implementation with batching.

Table 3 compares the three implementations of key operators in OD systems. Clearly, TOD with automatic batching achieves the best balance of efficiency and scalability, leading to 7.22×, 17.46×, and 11.49× speedups compared to a highly optimized NumPy implementation on CPUs. TOD can also handle more than 10× larger datasets where the direct PyTorch implementation faces out of memory (OOM) errors. TOD is only marginally slower than PyTorch when the input dataset is small (see the first row of each operator). In this case, batching is not needed and TOD is equivalent to PyTorch; TOD is slightly slower due to the overhead of TOD deciding whether or not to enable automatic batching.

7.7 Multi-GPU Results

We now evaluate the scalability of TOD on multiple GPUs on a single compute node. Specifically, we compare the run time of three compute-intensive OD algorithms (i.e., LOF, ABOD, and kNNOD) with 1,2,4, and 8 NVIDIA Tesla V100 GPUs.

Fig. 10 shows that for three OD algorithms tested, TOD can achieve nearly linear speed-up with more GPUs—the GPU efficiency is mostly above 90%. For instance, the kNNOD result shows that using 2, 4, and 8 GPUs are 1.91×, 3.73×, and 7.34× faster than the single-GPU performance. As a
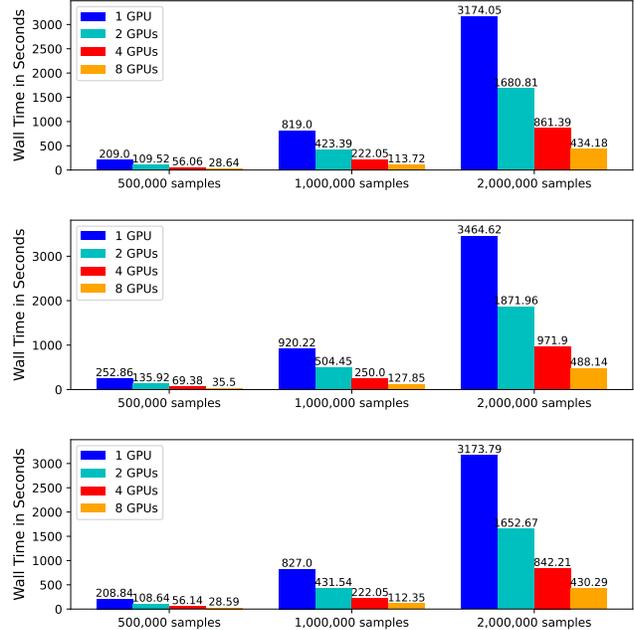

Figure 10. Runtime comparison of using different numbers of GPUs (**top**: LOF; **middle**: ABOD; **bottom**: kNNOD). TOD can efficiently leverage multiple-GPUs for faster OD.

comparison, using 2, 4, and 8 GPUs with ABOD and LOF are 1.85×, 3.63×, 7.14× faster and 1.91×, 3.70×, 7.27× faster, respectively. First, it is inevitable that there is a small overhead in multi-processing, causing the speed-up to not exactly be linear. Second, the minor efficiency difference between kNNOD and ABOD is due to most OD operations in the former being executed on multiple GPUs, while the latter involves several sequential steps that have to be run on CPUs. To sum up, TOD can leverage multiple GPUs efficiently to process large datasets.

8 CONCLUSION

In this paper, we propose the first comprehensive GPU-based outlier detection system called TOD[1] , which is on average 11.9 times faster than the leading system PyOD and takes less than an hour to detect outliers within a million samples. Our system enables many large-scale real-world OD applications that could have stringent time constraints.

---
[1]Code: https://github.com/yzhao062/pytod

In addition to the already implemented algorithms, TOD can facilitate the inclusion of additional algorithms conveniently. For instance, it is possible to include tree-based algorithms in TOD, e.g., iForest (Liu et al., 2008), by converting them to tensor operations (as discussed by Nakandala et al. (2020)). Additionally, TOD enables classical OD algorithms to fully run on GPUs so that they can be combined with deep-learning algorithms easily. One may use TOD to apply classical OD algorithms on the representations extracted by deep learning models as described by Ruff et al. (2019). Although TOD is designed for OD, its deeply optimized operators can also be used in other machine learning tasks such as clustering. Moreover, while we have focused on using exact implementations of operators, by allowing for approximation, additional acceleration is possible. For instance, exact nearest neighbor search can be switched to approximate nearest neighbor search.

REFERENCES

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pp. 265–283, 2016.

Achtert, E., Kriegel, H.-P., Reichert, L., Schubert, E., Wojdanowski, R., and Zimek, A. Visual evaluation of outlier detection models. In *International Conference on Database Systems for Advanced Applications (DASFAA)*, pp. 396–399. Springer, 2010.

Aggarwal, C. C. Outlier analysis. In *Data mining*, pp. 237–263. Springer, 2015.

Angiulli, F. and Pizzuti, C. Fast outlier detection in high dimensional spaces. In *The European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML-PKDD)*, pp. 15–27. Springer, 2002.

Angiulli, F., Basta, S., Lodi, S., and Sartori, C. A distributed approach to detect outliers in very large data sets. In D'Ambra, P., Guarracino, M., and Talia, D. (eds.), *Euro-Par 2010 - Parallel Processing*, pp. 329–340, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-15277-1.

Angiulli, F., Basta, S., Lodi, S., and Sartori, C. GPU strategies for distance-based outlier detection. *IEEE Trans. Parallel Distributed Syst.*, 27(11):3256–3268, 2016. doi: 10.1109/TPDS.2016.2528984. URL https://doi.org/10.1109/TPDS.2016.2528984.

Breunig, M. M., Kriegel, H., Ng, R. T., and Sander, J. LOF: identifying density-based local outliers. In Chen,

W., Naughton, J. F., and Bernstein, P. A. (eds.), *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA*, pp. 93–104. ACM, 2000a. doi: 10.1145/342009.335388. URL https://doi.org/10.1145/342009.335388.

Breunig, M. M., Kriegel, H.-P., Ng, R. T., and Sander, J. Lof: Identifying density-based local outliers. In Chen, W., Naughton, J. F., and Bernstein, P. A. (eds.), *SIGMOD Conference*, pp. 93–104. ACM, 2000b. ISBN 1-58113-217-4. URL http://dblp.uni-trier.de/db/conf/sigmod/sigmod2000.html#BreunigKNS00. SIGMOD Record 29(2), June 2000.

Daghaghi, S., Meisburger, N., Zhao, M., Wu, Y., Gobriel, S., Tai, C., and Shrivastava, A. Accelerating SLIDE deep learning on modern cpus: Vectorization, quantizations, memory optimizations, and more. *CoRR*, abs/2103.10891, 2021. URL https://arxiv.org/abs/2103.10891.

Dai, S., Venkatesan, R., Ren, H., Zimmer, B., Dally, W. J., and Khailany, B. Vs-quant: Per-vector scaled quantization for accurate low-precision neural network inference. *CoRR*, abs/2102.04503, 2021. URL https://arxiv.org/abs/2102.04503.

Gao, Y., Liu, Y., Zhang, H., Li, Z., Zhu, Y., Lin, H., and Yang, M. Estimating GPU memory consumption of deep learning models. In Devanbu, P., Cohen, M. B., and Zimmermann, T. (eds.), *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, pp. 1342–1352. ACM, 2020. doi: 10.1145/3368089.3417050. URL https://doi.org/10.1145/3368089.3417050.

Goldstein, M. and Dengel, A. Histogram-based outlier score (hbos): A fast unsupervised anomaly detection algorithm. *KI-2012: Poster and Demo Track*, pp. 59–63, 2012.

Goodfellow, I., Bengio, Y., and Courville, A. *Deep learning*. MIT press, 2016.

Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., et al. Array programming with numpy. *Nature*, 585(7825):357–362, 2020.

Hofmann, M. and Klinkenberg, R. *RapidMiner: Data mining use cases and business analytics applications*. CRC Press, 2013.

Jain, S. R., Gural, A., Wu, M., and Dick, C. Trained quantization thresholds for accurate and efficient fixed-point inference of deep neural networks. In Dhillon,

I. S., Papailiopoulos, D. S., and Sze, V. (eds.), *Proceedings of Machine Learning and Systems 2020, MLSys 2020, Austin, TX, USA, March 2-4, 2020*. mlsys.org, 2020. URL https://proceedings.mlsys.org/book/295.pdf.

Jia, Z., Lin, S., Gao, M., Zaharia, M., and Aiken, A. Improving the accuracy, scalability, and performance of graph neural networks with roc. In Dhillon, I. S., Papailiopoulos, D. S., and Sze, V. (eds.), *Proceedings of Machine Learning and Systems 2020, MLSys 2020, Austin, TX, USA, March 2-4, 2020*. mlsys.org, 2020. URL https://proceedings.mlsys.org/book/300.pdf.

Kahan, W. Ieee standard 754 for binary floating-point arithmetic. *Lecture Notes on the Status of IEEE*, 754(94720-1776):11, 1996.

Kriegel, H.-P., Schubert, M., and Zimek, A. Angle-based outlier detection in high-dimensional data. In Li, Y., Liu, B., and Sarawagi, S. (eds.), *KDD*, pp. 444–452. ACM, 2008. ISBN 978-1-60558-193-4. URL http://dblp.uni-trier.de/db/conf/kdd/kdd2008.html#KriegelSZ08.

Lazarevic, A., Ertöz, L., Kumar, V., Ozgur, A., and Srivastava, J. A comparative study of anomaly detection schemes in network intrusion detection. In Barbará, D. and Kamath, C. (eds.), *Proceedings of the Third SIAM International Conference on Data Mining, San Francisco, CA, USA, May 1-3, 2003*, pp. 25–36. SIAM, 2003. doi: 10.1137/1.9781611972733.3. URL https://doi.org/10.1137/1.9781611972733.3.

Leal, E. and Gruenwald, L. Research issues of outlier detection in trajectory streams using gpus. *SIGKDD Explor.*, 20(2):13–20, 2018. doi: 10.1145/3299986.3299989. URL https://doi.org/10.1145/3299986.3299989.

LeCun, Y. 1.1 deep learning hardware: past, present, and future. In *2019 IEEE International Solid-State Circuits Conference-(ISSCC)*, pp. 12–19. IEEE, 2019.

Lee, M., Zhao, Y., Wang, A., Liang, P. J., Akoglu, L., Tseng, V. S., and Faloutsos, C. Autoaudit: Mining accounting and time-evolving graphs. In Wu, X., Jermaine, C., Xiong, L., Hu, X., Kotevska, O., Lu, S., Xu, W., Aluru, S., Zhai, C., Al-Masri, E., Chen, Z., and Saltz, J. (eds.), *IEEE International Conference on Big Data, Big Data 2020, Atlanta, GA, USA, December 10-13, 2020*, pp. 950–956. IEEE, 2020. doi: 10.1109/BigData50022.2020.9378346. URL https://doi.org/10.1109/BigData50022.2020.9378346.

Lee, W., Sharma, R., and Aiken, A. On automatically proving the correctness of math.h implementations.

Proc. ACM Program. Lang., 2(POPL):47:1–47:32, 2018. doi: 10.1145/3158135. URL https://doi.org/10.1145/3158135.

Li, W., Wang, Y., Cai, Y., Arnold, C. W., Zhao, E., and Yuan, Y. Semi-supervised rare disease detection using generative adversarial network. *CoRR*, abs/1812.00547, 2018. URL http://arxiv.org/abs/1812.00547.

Li, Z., Zhao, Y., Botta, N., Ionescu, C., and Hu, X. COPOD: copula-based outlier detection. In Plant, C., Wang, H., Cuzzocrea, A., Zaniolo, C., and Wu, X. (eds.), *20th IEEE International Conference on Data Mining, ICDM 2020, Sorrento, Italy, November 17-20, 2020*, pp. 1118–1123. IEEE, 2020. doi: 10.1109/ICDM50108.2020.00135. URL https://doi.org/10.1109/ICDM50108.2020.00135.

Liu, F. T., Ting, K. M., and Zhou, Z. Isolation forest. In *Proceedings of the 8th IEEE International Conference on Data Mining (ICDM 2008), December 15-19, 2008, Pisa, Italy*, pp. 413–422. IEEE Computer Society, 2008. doi: 10.1109/ICDM.2008.17. URL https://doi.org/10.1109/ICDM.2008.17.

Looks, M., Herreshoff, M., Hutchins, D., and Norvig, P. Deep learning with dynamic computation graphs. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017. URL https://openreview.net/forum?id=ryrGawqex.

Lozano, E. and Acuña, E. Parallel algorithms for distance-based and density-based outliers. In *Proceedings of the 5th IEEE International Conference on Data Mining (ICDM 2005), 27-30 November 2005, Houston, Texas, USA*, pp. 729–732. IEEE Computer Society, 2005. doi: 10.1109/ICDM.2005.116. URL https://doi.org/10.1109/ICDM.2005.116.

Meng, C., Sun, M., Yang, J., Qiu, M., and Gu, Y. Training deeper models by gpu memory optimization on tensorflow. In *Proc. of ML Systems Workshop in NIPS*, 2017.

Nakandala, S., Saur, K., Yu, G., Karanasos, K., Curino, C., Weimer, M., and Interlandi, M. A tensor compiler for unified machine learning prediction serving. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, pp. 899–917. USENIX Association, 2020. URL https://www.usenix.org/conference/osdi20/presentation/nakandala.

Neeb, H. and Kurrus, C. Distributed k-nearest neighbors, 2016.

Neubig, G., Goldberg, Y., and Dyer, C. On-the-fly operation batching in dynamic computation graphs. In Guyon, I., von Luxburg, U., Bengio, S., Wallach, H. M., Fergus, R., Vishwanathan, S. V. N., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pp. 3971–3981, 2017. URL https://proceedings.neurips.cc/paper/2017/hash/c902b497eb972281fb5b4e206db38ee6-Abstract.html.

Oku, J., Tamura, K., and Kitakami, H. Parallel processing for distance-based outlier detection on a multi-core cpu. In *IEEE International Workshop on Computational Intelligence and Applications (IWCIA)*, pp. 65–70. IEEE, 2014.

Papadimitriou, S., Kitagawa, H., Gibbons, P. B., and Faloutsos, C. LOCI: fast outlier detection using the local correlation integral. In Dayal, U., Ramamritham, K., and Vijayaraman, T. M. (eds.), *Proceedings of the 19th International Conference on Data Engineering, March 5-8, 2003, Bangalore, India*, pp. 315–326. IEEE Computer Society, 2003. doi: 10.1109/ICDE.2003.1260802. URL https://doi.org/10.1109/ICDE.2003.1260802.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32:8026–8037, 2019.

Pevný, T. Loda: Lightweight on-line detector of anomalies. *Mach. Learn.*, 102(2):275–304, 2016. URL http://dblp.uni-trier.de/db/journals/ml/ml102.html#Pevny16.

Raschka, S. *Python machine learning*. Packt publishing ltd, 2015.

Ruff, L., Vandermeulen, R. A., Görnitz, N., Binder, A., Müller, E., Müller, K.-R., and Kloft, M. Deep semisupervised anomaly detection. In *International Conference on Learning Representations*, 2019.

Schubert, E., Zimek, A., and Kriegel, H. Generalized outlier detection with flexible kernel density estimates. In Zaki, M. J., Obradovic, Z., Tan, P., Banerjee, A., Kamath, C., and Parthasarathy, S. (eds.), *Proceedings of the 2014 SIAM International Conference on Data Mining, Philadelphia, Pennsylvania, USA, April 24-26, 2014*, pp. 542–550. SIAM, 2014. doi: 10.1137/1.9781611973440.63. URL https://doi.org/10.1137/1.9781611973440.63.

Shyu, M.-L., Chen, S.-C., Sarinnapakorn, K., and Chang, L. A novel anomaly detection scheme based on principal component classifier. Technical report, MIAMI UNIV CORAL GABLES FL DEPT OF ELECTRICAL AND COMPUTER ENGINEERING, 2003.

Sincraian, P. PyOD download statistics. https://pepy.tech/project/pyod, 2021. Accessed: 2021-09-09.

Svedin, M., Chien, S. W. D., Chikafa, G., Jansson, N., and Podobas, A. Benchmarking the nvidia GPU lineage: From early K80 to modern A100 with asynchronous memory transfers. In Plessl, C., Chow, P., and Platzner, M. (eds.), *HEART '21: 11th International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies, Virtual Event, Germany, 21-23 June, 2021*, pp. 9:1–9:6. ACM, 2021. doi: 10.1145/3468044.3468053. URL https://doi.org/10.1145/3468044.3468053.

Tang, J., Chen, Z., Fu, A. W., and Cheung, D. W. Enhancing effectiveness of outlier detections for low density patterns. In Cheng, M., Yu, P. S., and Liu, B. (eds.), *Advances in Knowledge Discovery and Data Mining, 6th Pacific-Asia Conference, PAKDD 2002, Taipei, Taiwan, May 6-8, 2002, Proceedings*, volume 2336 of *Lecture Notes in Computer Science*, pp. 535–548. Springer, 2002. doi: 10.1007/3-540-47887-6_53. URL https://doi.org/10.1007/3-540-47887-6_53.

Wang, H., Zhai, J., Gao, M., Ma, Z., Tang, S., Zheng, L., Li, Y., Rong, K., Chen, Y., and Jia, Z. PET: optimizing tensor programs with partially equivalent transformations and automated corrections. In Brown, A. D. and Lorch, J. R. (eds.), *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*, pp. 37–54. USENIX Association, 2021. URL https://www.usenix.org/conference/osdi21/presentation/wang.

Zhao, Y. PyOD citation statistics. https://scholar.google.ca/scholar?cites=3726241381117726876&as_sdt=5,39&scioidt=0,39&hl=en, 2021. Accessed: 2021-09-09.

Zhao, Y., Nasrullah, Z., Hryniewicki, M. K., and Li, Z. LSCP: locally selective combination in parallel outlier ensembles. In *SDM*, pp. 585–593. SIAM, May 2019a.

Zhao, Y., Nasrullah, Z., and Li, Z. PyOD: A python toolbox for scalable outlier detection. *JMLR*, 20:96:1–96:7, 2019b. URL http://jmlr.org/papers/v20/19-011.html.

Zhao, Y., Hu, X., Cheng, C., Wang, C., Wan, C., Wang, W., Yang, J., Bai, H., Li, Z., Xiao, C., Wang, Y., Qiao, Z., Sun,

J., and Akoglu, L. SUOD: Accelerating large-scale unsupervised heterogeneous outlier detection. *Proceedings of Machine Learning and Systems*, 2021.

SUPPLEMENTARY MATERIAL FOR TOD: TENSOR-BASED OUTLIER DETECTION

Datasets and Additional Experimental Results.

A DATASETS

Table A.1 shows the six real-world benchmark datasets used in this study, which are widely evaluated in OD research (Zhao et al., 2019a; Ruff et al., 2019; Li et al., 2020; Zhao et al., 2021). Given the limited size of the real-world datasets, we use the data generation function in PyOD to create larger synthetic datasets as well.

Table A.1. Real-world benchmark datasets used in the experiments

| Dataset	Pts (n)	Dim (d)	% Outlier
mnist	7603	100	9.21
musk	3062	166	3.17
mammography	11183	6	2.32
speech	3686	400	1.65
f-mnist	60000	784	10
cifar-10	60000	3072	10

B ADDITIONAL EXPERIMENTAL RESULTS

### B.1 End-to-end Evaluation on Synthetic Datasets

As discussed in §7.3, we compare the runtime of TOD and PyOD on two real-world datasets (fashion-mnist and cifar-10, see Fig. 7) and two larger synthetic datasets (see Appendix Fig. B.1). The results show that TOD is faster than PyOD on all five algorithms tested for both real-world and synthetic datasets.

B.2 Provable Quantization with topk

As discussed in §7.5, the results demonstrate that provable quantization always leads to memory savings. Specifically, Figure. B.2 shows that topk with provable quantization reduces memory usage by 73.49% (with 16-bit precision) and 49.58% (with 32-bit precision) compared to using full precision.
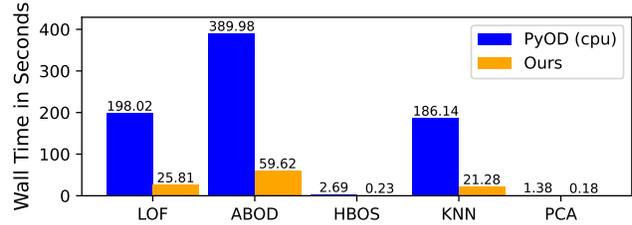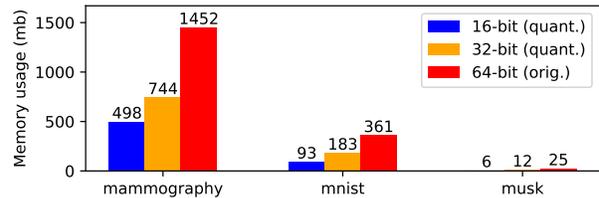

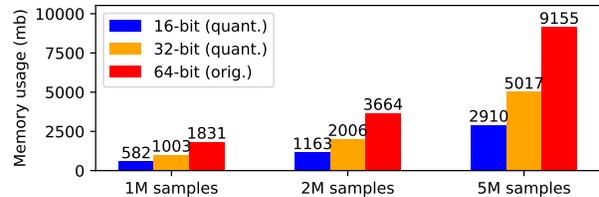(a) Synthetic dataset 1 (100,000 samples with 200 features)
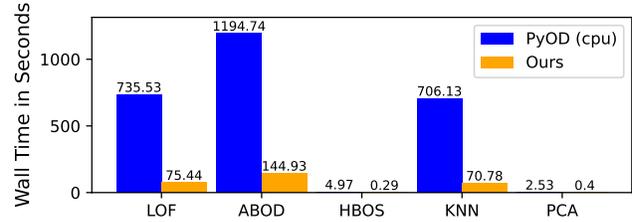
(b) Synthetic dataset 2 (200,000 samples with 200 features)

Figure B.1. Runtime comparison between PyOD and TOD in seconds for two synthetic datasets. See Fig. 7 for results on real-world datasets. TOD outperforms PyOD in all experiments.

(a) topk (real-world datasets)

(b) topk (synthetic datasets)

Figure B.2. GPU memory consumption comparison between using provable quantization (16-bit and 32-bit) and full precision (64-bit). Clearly, provable quantization leads to significant memory consumption savings for both operators.