

XPySom: Accelerating SOM-based Services on Multi-Cores and GP-GPUs

Riccardo Mancini¹ and Tommaso Cucinotta

Scuola Superiore Sant'Anna

Pisa, Italy

e-mail: {firstname.lastname}@santannapisa.it

Abstract

Self Organizing Maps (SOMs) are unsupervised, shallow, artificial neural networks, built on top of the competitive learning principle, and typically employed for clustering, dimensionality reduction and high-dimensional data visualization. This paper presents *XPySom*, a high-performance and parallel implementation of SOMs for Python, with a focus on acceleration of the computations through the use of multicore processors and GP-GPUs. After describing the major design and implementation choices, we present results from an extensive experimental evaluation of XPySom on the Extended MNIST openly available dataset of handwritten digits. On this dataset, the training time is shown to be up to two orders of magnitude faster than existing open-source SOM implementations, taking as little as 2 seconds to train the model on the 240000 images in the dataset for 100 epochs. The implementation is released under an open-source GPLv3 license, and made publicly available as the `xpysom` package for Python.

Keywords: self-organizing maps (SOMs), performance comparison, experimental evaluation, GP-GPU acceleration

1. Introduction

In recent years, the adoption of data-driven techniques for decision-support services has grown steadily across a variety of application domains. This growth has been driven by the rapid evolution of Information and Communication Technologies (ICTs) over the last two decades. First, the widespread availability of affordable, high-speed Internet connectivity has enabled the collection and sharing of increasingly massive quantities of online data. Second, the Internet-of-Things (IoT) paradigm [1] has interconnected a vast number of data sources and sinks, causing an unprecedented surge in daily data traffic. Third, the rise of cloud and big data technologies [2] has introduced

ubiquitous, on-demand scaling to process and extract critical value from these ever-expanding data streams [3]. Finally, heterogeneous hardware architectures have evolved to deliver greater processing power with significantly lower energy footprints. Key milestones in this hardware transition include the development of multi-core and many-core processors, and more recently, General-Purpose Graphics Processing Units (GPUs). Together, these advancements have enabled complex processing scenarios that were practically unimaginable just a decade ago.

These technological shifts have enabled the deployment of highly sophisticated data processing pipelines. This drove the transition towards the modern era of data science, where Machine Learning (ML), Artificial Neural Networks (ANNs), and Deep Neural Networks (DNNs) are trained on vast datasets, learning their behavior directly from data rather than relying on manually written code.

¹Riccardo Mancini was at Scuola Superiore Sant'Anna while working on this paper. Now, he is a software engineer in AWS Lambda.

This rapid ICT revolution has driven the emergence of highly complex data processing services that demand substantial computational power [4]. To keep computations within reasonable time limits and at sustainable costs, smart optimizations and hardware acceleration techniques, including many-core processors and heterogeneous computing with GPU and FPGA accelerators, have become fundamental building blocks for supporting modern high-performance embedded systems [5, 6, 7, 8, 9]. In this context, the use of GP-GPUs has become the de-facto standard to accelerate ML services [10, 11, 12], coupled with the use of the increasingly popular Python programming language, thanks to the availability of good libraries with efficient native implementations.

As an instance of this growing trend, researchers have been investigating the acceleration of Self-Organizing Maps [13, 14, 15], given their suitability to analyze huge quantities of data available in a multitude of diverse domains, including: medical images analysis [16]; defect classification in the steel industry [17, 18, 19]; analysis of web network traffic [20]; and behavioral analysis of virtualized network functions within Telecommunication infrastructures [21, 22].

1.1. Paper Contributions

This paper presents *XPySom*, a high-performance, parallel and efficient implementation of SOMs in Python focusing on multicore and GPU acceleration, leveraging on the use of the *numpy* [23] and *cupy* [24] libraries. It is released as the open-source *xpysom* package for Python². *XPySom* achieves high efficiency and performance by exploiting the linear algebra optimizations provided by native BLAS libraries³, such as the *Intel Math Kernel Library (MKL)* [25] and *cuBLAS*⁴. We rearrange the data processing operations to avoid slow interpreted loops in Python, replacing them with matrix-based operations that leverage highly optimized native library functions. This design makes optimal use of the available parallelism and GPU acceleration. With a streamlined

codebase, *XPySom* invokes the underlying *NumPy* or *CuPy* libraries only a few times per training batch, resulting in a high-performance implementation that outperforms alternative open-source SOM libraries.

An experimental comparison of *XPySom* is performed on the Extended MNIST [26] data set, which contains 240000 28x28-pixel pictures of handwritten digits, against other popular open-source implementations: *MiniSom* [27], *TensorFlow SOM* [28], *Somoclu* [29, 30], *SOMvec* [31], and *SuSi* [32, 33]. The experiments show how *XPySom* is an order of magnitude faster than the other evaluated SOM implementations, both for multicore and GPU-accelerated platforms, and it is up to 2 orders of magnitude faster in some benchmarks.

1.2. Differences from prior conference version

This paper extends our prior conference publication [13], introducing: (1) algorithmic optimizations via caching of constant terms for pairwise Euclidean distances and generalized Minkowski p -norm formulations (Sections 4.1 and 6.5); (2) an expanded comparative evaluation in Section 6.3 including two additional SOM implementations (*SuSi* and *SOMvec*); (3) new experimental results performed on an upgraded hardware (Intel Xeon E5-2698 v4 CPU and NVIDIA Tesla V100 GPU), complementing the ones on the older setup in [13] (i7-4790K CPU and GTX 1080Ti GPU); and (4) updated scaling and quantization error convergence benchmarks comparing all seven implementations and different distance functions (Section 6.4). These additions provide a comprehensive evaluation of the performance-accuracy trade-offs achievable through *XPySom*.

1.3. Paper Organization

This paper is organized as follows. Section 2 provides an overview of SOMs and their state-of-the-art implementations. Section 3 briefly discusses relevant works found in the research literature. Section 4 presents *XPySom*, discussing the most important choices in its design regarding optimized execution and parallelization, while Section 5 provides additional implementation details. Section 6 shows an experimental evaluation of *XPySom*, comparing against other widely used open-source SOM

²More information at: <https://github.com/Manciukic/xpysom>.

³More information at: <http://www.netlib.org/blas/>.

⁴More information at <https://docs.nvidia.com/cuda/cublas/index.html>

implementations. Section 7 provides some final remarks, along with a brief discussion of possible directions for future work on the topic.

2. Background

This section presents background concepts about Self Organizing Maps, explaining the general working of the algorithm (Section 2.1), and then describing the two most common variations: the *online learning* algorithm (Section 2.2) and the *batch learning* algorithm (Section 2.3).

2.1. Self Organizing Maps

A Self Organizing Map (SOM) [34] is a particular type of unsupervised and shallow Artificial Neural Network, based on the competitive learning principle, used mainly in clustering and dimensionality reduction. The core idea is that a SOM maps a high-dimensional space into a lower-dimensional one (most commonly 2D), while preserving *topology* and *distribution* of the original data. This lower-dimensionality representation can be used for visualization, decision-making, and anomaly detection.

The dimensionality reduction is performed in a SOM through an unsupervised vector quantization technique that maps a set of N -dimensional input data samples $V \subseteq \mathbb{R}^N$ into a finite set $W = w_1, \dots, w_M, w_i \in \mathbb{R}^N$ of vectors, called *codebooks* or *weights*. The *codebooks* correspond to two-dimensional nodes $\mathcal{L}$ in a finite lattice graph, called *neurons* or *units* $\{r_i\}$. The lattice graph is embedded in a (usually bi-dimensional) euclidean space, forming a regular tiling. For example, a finite grid lattice $\mathcal{L} \subset \mathbb{R}^2$ of dimensions $L_W \times L_H$, with $L_W, L_H \in \mathbb{N}$, can be obtained as follows:

$$\mathcal{L} = \{hA + kB \mid 0 \leq h < L_W, 0 \leq k < L_H, h, k \in \mathbb{N}\},$$

where $A, B \in \mathbb{R}^2$ determine its shape. For example, a *rectangular* grid is obtained by using $A = (1, 0), B = (0, 1)$, while a *triangular* one with $A = (\frac{1}{2}, \frac{\sqrt{3}}{2}), B = (1, 0)$. Elements r_i of the lattice are usually indexed from 1 to $M = L_W \times L_H$.

Formally, a SOM can be defined as a pair (w, b) , where:

- $w : \mathcal{L} \rightarrow \mathbb{R}^N, w_i = w(r_i)$ is a discrete map from the two-dimensional lattice $\mathcal{L}$ into the N -dimensional vector space of samples in V , called *features space*;

- $b : \mathbb{R}^N \rightarrow \mathcal{L}$ returns the *best-matching unit* (BMU) for each input data depending on a distance d within the *feature space*:

$$b(x) = \underset{r_i \in \mathcal{L}}{\operatorname{argmin}} d(x, w(r_i)).$$

Usually, the Euclidean distance is used, but other choices are possible, like the cosine distance [35], shown in Equation (1a), or the Minkowski p -norm [36], shown in Equation (1b), where for example the Manhattan distance can be obtained with $p = 1$, or the Euclidean one with $p = 2$:

$$d_{\cosine}(x, w_i) = 1 - \frac{x \cdot w_i}{\|x\| \cdot \|w_i\|}; \quad (1a)$$

$$d_{minkowski}(x, w_i, p) = \sqrt[p]{\sum_j |x_j - w_{i,j}|^p}. \quad (1b)$$

When an input sample x is mapped to a trained SOM, its associated BMU $b(x)$ identifies the neuron whose weight vector $w(b(x))$ is as close as possible to x according to the chosen distance d , i.e., the best “approximation” of x is identified, among the weight vectors. Therefore, a SOM is trained by incrementally adjusting the weights of the individual neurons to decrease the average distance between the input samples and their looked-up BMU weight vectors. This can be done either through an *online* or *batch* learning procedure, as detailed below.

2.2. Online learning

With the SOM *online learning* algorithm, for each training epoch T , the *weights* are updated at each iteration t from a given input vector $x \in V$. An epoch ends when all input vectors in V have been processed exactly once. At each iteration t , the BMU $b(x)$ is found and the *weights of all units* are updated according to a given *neighborhood function* $h : (\mathcal{L}, \mathcal{L}, \mathbb{N}) \rightarrow \mathbb{R}$, a learning rate function $\eta : \mathbb{N} \rightarrow \mathbb{R}$, and a radius function $\delta : \mathbb{N} \rightarrow \mathbb{R}$, as shown in Equation (2a), where $w_i^{[t]}$ represents the weights of neuron i at the previous iteration t and $w_i^{[t+1]}$ are its new weights.

$$w_i^{[t+1]} = w_i^{[t]} + h(b(x), r_i, t)(x - w_i^{[t]}) \quad \forall r_i \in \mathcal{L}. \quad (2a)$$

$$h(r, s, T) = \eta(T) e^{-\frac{\|r-s\|^2}{\delta(T)}}, \quad r, s \in \mathcal{L}. \quad (2b)$$

A popular choice for h is for example the Gaussian neighborhood function [37] shown in Equation (2b). The learning rate $\eta(T)$ and radius $\delta(T)$ decrease with the current epoch T , to make the algorithm converge. Usually, exponentially or linearly decreasing functions are used for both. The neighborhood function h acts as a magnet to attract *units* close to the BMU in order to preserve the lattice topology. Thus, not only the winning *neuron* is updated, but all of them, depending on their distance from the BMU. Optionally, the h function may be forced to zero beyond a maximum distance between the BMU $b(x)$ and r_i , to keep neuron updates compact, local to neurons close to the BMU.

Before starting the training, the weights are initialized, either randomly sampling the V dataset, or using Principal Components Analysis [38].

2.3. Batch learning

Although conceptually simple, the online SOM learning algorithm is not suited for parallel implementation because each iteration depends on the previous one. Consequently, multiple input samples cannot be evaluated concurrently, as each one of them modifies the *weights*. The *batch learning* algorithm [39] resolves this serialization bottleneck by enabling concurrent evaluation of multiple input samples. Instead of updating the *weights* at each data sample, the algorithm processes a batch $B \subseteq V$ of data samples and computes the weight updates using their weighted average, as shown in Equation (3). This formulation allows parallel computations of all terms in both numerator and denominator sums.

$$w_i^{[t+1]} = \frac{\sum_{x \in X} h(b(x), r_i, T)x}{\sum_{x \in X} h(b(x), r_i, T)}, \forall r_i \in \mathcal{L}. \quad (3)$$

2.4. Quantization error

The quality of a trained SOM is usually measured in terms of the average distance of the samples in V to their respective BMUs in the final weights after training. This is done recurring to the so called *quantization error*, defined as:

$$\epsilon = \frac{1}{|V|} \sum_{x \in V} d(x, w(b(x))). \quad (4)$$

This metric decreases through the training epochs.

Using batch learning to train the SOM usually results in the need for a higher number of training epochs, compared to using online learning, to achieve a target accuracy in terms of quantization error. However, the performance advantages obtained through parallelization make this worthwhile. Therefore, when comparing different SOM implementations in terms of their performance and execution time, it is important to keep an eye on the quantization error, to make sure that the comparison is carried out at equal or comparable final accuracy. This will be highlighted in Section 6.4.

3. Related Work

Practically, most data-intensive algorithms, like ML ones, can take advantage of the parallelism in modern hardware architectures. This includes clustering techniques [40], like SOMs, visualization methods for many-dimensional spaces [41], and classic statistical algorithms [42]. In particular, SOMs, in their batch formulation, can easily take advantage of parallel computations in multiple processors and, also SIMD architectures, like GP-GPUs and the SIMD extensions available in modern high-end processors, like Advanced Vector eXtensions (AVX) [43], and Streaming SIMD Extensions (SSE). However, in these architectures, most of the time memory accesses become the bottleneck for high-performance, and implementations that make smart use of caches and memory locality exhibit superior performance. For this reason, highly optimized libraries, called BLAS, have been developed for performing the most common linear algebra operations, like Intel MKL [25] and cuBLAS [44]. In the research literature, multiple works addressed further the performance and optimization of SOMs, many of which focusing on accelerating computations via multicores and GPUs.

3.1. Concurrent computation of SOMs

In [45], a proposal for the parallelization of SOMs using *online learning* is proposed. In order to concurrently compute the BMU for each input sample, the lattice is subdivided into P partitions, where P is the number of processes. Each process is then responsible for computing the local minimization problem and finding the BMU in its partition.

Then, the partial results are aggregated to find the final BMU, and the result is propagated back to the other processes, to update the weights of the neurons they are responsible for. The authors show experimentally that the synchronization overhead is negligible compared to the cost of the floating point operations involved in the calculations, and report that the results could be improved by exploiting better the L2 cache.

In [46], the authors implemented a SOM combining both data and model parallelism, pushing the parallelization capability of the batch training algorithm to the extreme. Not only the training set is split into independently-processed batches, but also the map is partitioned at unit weights-level, as in [45]. This enables a single (CPU or GPU) thread to handle updates for only a neuron. The method is shown experimentally to perform best when dealing with many input samples, but its effectiveness is reduced as the features per data point increase.

Some researchers tackled the problem of distributed architectures for training SOMs on vast amounts of input data. In [47], the *Spark* framework was used to efficiently scale the computations onto massive datasets of up to 100 million points. In [29], the researchers developed an implementation based on the MapReduce-MPI [48] framework, which is used to distribute the computation over multiple processes, and also on GPUs. The approach is refined in a following work [30], which presents an open-source parallel SOM library, called *Somochu*. In this renewed implementation, the MapReduce framework is replaced with MPI (for inter-node communications), and OpenMP (for single-node parallelism). For GPU execution, the Thrust [49] and cuBLAS [44] libraries are used. This results in a flexible solution that can be used both on single machines and on clusters, capable of exploiting both multicore CPUs and multiple GPUs. Furthermore, the library exposes interfaces compatible with widely-used data analysis ecosystems, like *Python*, *R* and *MATLAB*. However, despite the claimed efficiency of the implementation, in Section 6 we will see how the much simpler design of our proposed *XPySom* results in a processing efficiency that is an order of magnitude better on both multicores and GPU-equipped platforms.

Finally, Lachmair et al. [14] performed an in-depth analysis of the energy efficiency and per-

formance of multiple architectures, and concluded that none of them performs consistently better than the other in terms of energy efficiency for the complete design space. In particular, ARM-based SoCs achieved the highest efficiency for small network sizes, while FPGAs and GPUs performed best for large data sets. Large simulations especially benefit from the FPGA implementation, outperforming the optimized CPU implementation by a factor of 220, and providing a 28-times higher energy efficiency. In the comparison presented in this paper, we did not perform energy consumption measurements, as our focus was on reducing the execution time of SOM training tasks, but this might be a possible dimension to consider in the future.

3.2. GPU acceleration of SOMs

Regarding GPU implementations, an OpenCL- and CUDA-based approach combined with MPI is evaluated compared to a multi-processor only baseline in [50]. The approach is suitable for both single and multiple GPUs, but is shown not to provide significant gains in the latter, due to the synchronization requirements during the training procedure. Furthermore, the CUDA implementation is faster than the OpenCL one, given the specificity to the NVIDIA GPUs being used.

In [51], Xiao et al. remark that GPU implementations of SOMs are inefficient when big neighborhoods are used, due to the large number of weights update operations that need to be serialized. To address it, the researchers developed a novel approach that implements the distance computation between the training samples and the weights exploiting the parallelism of the compute shader in the rendering capabilities of GPUs.

Liu et al. [52] proposed a heterogeneous parallel implementation based on MPI and CUDA, that is particularly efficient in distributing the load onto multiple GPUs thanks to the process- and thread-level data parallelism that is employed. Particularly, the *batch learning* algorithm with a single weight update per epoch (batch size equal to training data size) is used, where the most computationally-intensive parts can be turned into matrix operations that can be performed with cuBLAS, fully exploiting the computational capacity of GPUs.

Only a few of the approaches described above are generally available as up-to-date open-source soft-

ware that can be run on modern platforms. In what follows, our proposed open-source *XPySom* is first described in Section 4, then evaluated experimentally in Section 6, where we show evidence of the effectiveness of its streamlined design and efficient implementation, compared to the available open-source alternatives.

4. A Novel “Matrix” Approach for SOM

This section introduces the “matrix approach” to the *batch learning* algorithm that is implemented by *XPySom*. Writing efficient, custom vectorized code for CPUs and GPUs is a complex task. Performance depends not only on the raw number of executed operations, but also heavily on memory access patterns that exploit data locality and hardware caches, which vary significantly across architectures. To address this complexity, ready-made Basic Linear Algebra Subprograms (BLAS) libraries are widely available. These libraries leverage SIMD extensions on CPUs and highly optimized memory access patterns to maximize cache utilization and data locality.

Rearranging the batch learning algorithm formulas (Section 2.3) into matrix operations allows us to leverage these highly optimized BLAS routines, outperforming custom implementations. Furthermore, this matrix-based abstraction simplifies porting the algorithm between CPU and GPU platforms. For more implementation details, see Section 5.

In the following, we will assume that the batch X of input data samples is arranged in a $B \times N$ matrix (i.e. $X \in \mathbb{R}^{B \times N}$), where B is the number of data samples in the batch, and N the number of dimensions. The weights of the neurons are in turn arranged into a matrix W of size $M \times N$ (i.e. $W \in \mathbb{R}^{M \times N}$), where $M = L_W \times L_H$ is the number of neurons.

The remaining sections are organized in the order in which the operations are executed in the algorithm. First, the pair-wise distances between batch input samples and the weights are computed (Section 4.1). The result is a matrix $D = \{d_{i,j}\} \in \mathbb{R}^{B \times M}$, where each item $d_{i,j}$ is the distance between the i^{th} input sample in X and the j^{th} weight vector in W . Next, the best matching units (BMUs) for each input data sample in the batch are found (Section 4.2). This means finding the index of the small-

est value in each row of D as a vector $BMU \in \mathbb{N}^B$. Then, the neighborhood function is computed for each element of the BMU vector, yielding a matrix $H \in \mathbb{R}^{B \times M}$ (Section 4.3). Finally, the numerator and denominator sums are aggregated and the weights of the neurons are updated, after all batches in input, by dividing element-wise the numerator matrix $\mathcal{N} \in \mathbb{R}^{M \times N}$ by the denominator vector $\mathcal{D} \in \mathbb{R}^M$ (Section 4.4).

4.1. Calculating pair-wise distances

In the SOM *batch learning* algorithm, the first step of each iteration is computing the pair-wise distances between the B data samples in the batch X and the M weights in W , with operations in $\mathbb{R}^N$. This is the most expensive computation to perform, resulting in $O(B \times M \times N)$ number of computations.

This section first presents the optimized formulation for the euclidean distance (Section 4.1.1), then generalizes it for all Minkowski distances with an even norm p (Section 4.1.2).

4.1.1. Euclidean distance

The pair-wise distance $D = \{d_{i,j}\}$ between X and W is defined as:

$$d_{i,j} = \|x_i - w_j\| = \sqrt{\sum_{k=1}^N (x_{i,k} - w_{j,k})^2} \quad (5)$$

Note that, since to find BMUs we are interested in comparing distances in a row of D , we can drop the square-root operation, without changing the ordering between the elements. Next, we can note that this requires $2 \times B \times M \times N$ Multiply-Accumulate operations⁵ (MACs):

- $B \times M \times N$ subtractions to compute all differences
- $B \times M \times N$ MACs to sum all powers

This can be improved if we rewrite the formulation of $D = d_{i,j}$ in Equation (5), similarly to what

⁵Multiplying by a value and adding the result to an accumulator can be efficiently calculated with common dedicated hardware.

done in [53, 54], obtaining the formulation in Equation (6), shown below:

$$d_{i,j}^2 = \|x_i\|^2 - 2\langle x_i, w_j \rangle + \|w_j\|^2$$

$$d_{i,j}^2 = \sum_{k=1}^N x_{i,k}^2 - 2 \sum_{k=1}^N x_{i,k} w_{j,k} + \sum_{k=1}^N w_{j,k}^2 \quad (6)$$

With this approach, the number of operations is:

- $B \times N$ MACs to sum all squared items of X ;
- $B \times M \times N$ MACs to sum the dot product $\langle x_i, w_j \rangle$;
- $M \times N$ MACs to sum all squared items of W ;
- $2 \times M \times B$ ops for the final sum of the three parts.

The sum can be rewritten as $B \times M \times N \times (1 + \frac{1}{M} + \frac{1}{B} + \frac{2}{N}) \simeq B \times M \times N$, as the coefficient in parentheses is in practice very close to 1 for big values of M , B , and N . Furthermore, this formulation can leverage on the highly-optimized matrix multiplication functions from BLAS libraries, exploiting better memory locality to further improve the efficiency.

Finally, we can also note 2 further optimizations:

- Since $\|x_i\|^2$ is constant on every row of D , and we are only interested in the minimum across the row, we can remove it.
- Since the weights do not change during the epoch, we can compute the vector of squares $\|w_j\|^2$ only once per epoch, and reuse it for each batch.

The final version of the algorithm used by *XPySOM* is shown in Equation (7) below:

$$\tilde{d}_{i,j}^2 = \|w_j\|_{\text{cached}}^2 - 2\langle x_i, w_j \rangle. \quad (7)$$

4.1.2. "Norm-even" distance

We can extend the just described optimization to every even Minkowski norm p as follows:

$$d_{i,j}^p = \sum_{k=1}^N |x_{i,k} - w_{j,k}|^p = \sum_{k=1}^N (x_{i,k} - w_{j,k})^p =$$

$$= \sum_{k=1}^N \sum_{t=0}^p (-1)^{p-t} \binom{p}{t} x_{i,k}^t w_{j,k}^{p-t} =$$

$$= \sum_{t=0}^p (-1)^{p-t} \binom{p}{t} \sum_{k=1}^N x_{i,k}^t w_{j,k}^{p-t}. \quad (8)$$

$$D^p = \sum_{t=0}^p (-1)^{p-t} \binom{p}{t} X^{\circ t} \cdot (W^T)^{\circ(p-t)}. \quad (9)$$

Note that at line 1 in Equation (8), we need p to be even. In this formulation, all $p+1$ parts of the sum are accumulated by adding at each step the dot product of the element-wise t -power of X (denoted as $X^{\circ t}$) with the element-wise $(p-t)$ -power of W . This is efficiently implemented by BLAS libraries.

4.2. Best Matching Unit

For each input sample x_i , its best matching unit (BMU) is the unit whose weight w_j is closer to it, minimizing $d_{i,j}$. In this matrix formulation, we will call $BMU \in \mathbb{N}^B$ the vector that associates every input vector x_i to the index j of its BMU. This can be calculated element-wise as in Equation (10), where each computation is independent, and can therefore be embarrassingly parallelized.

$$BMU_i = \operatorname{argmin}_j d_{i,j}. \quad (10)$$

4.3. Neighborhood function

The neighborhood function is computed as a matrix $H \in \mathbb{R}^{B \times M}$, where $h_{i,k}$ represents the weight of the update to be applied to the unit w_k by the input vector x_i . While we describe a Gaussian neighborhood function on a standard rectangular grid layout, the same approach easily generalizes to alternative functions and topologies.

Let $G \in \mathbb{N}^{M \times 2}$ be the matrix where each row g_j is the (x, y) coordinates of neuron j in the grid. Similarly, let $U \in \mathbb{N}^{B \times 2}$ be the matrix where each row u_i is the (x, y) coordinates of BMU_i in the grid. In a rectangular grid, we can simply unravel the BMU_i indices found earlier to obtain directly the coordinates of the winning neuron in the $L_W \times L_H$ grid as follows:

$$u_{i,1} = BMU_i \bmod L_W \in [0, L_W)$$

$$u_{i,2} = BMU_i \operatorname{div} L_W \in [0, L_H). \quad (11)$$

With G and U , we can finally compute the H neighborhood matrix from Equation (2b) as follows:

$$h_{i,j} = \eta(T) e^{-\frac{\|u_i - g_j\|^2}{\delta(T)}}. \quad (12)$$

where $\delta(T)$ and $\eta(T)$ are, the neighborhood radius and the learning rate at epoch T , respectively. Note that each row can be independently computed, enabling straightforward parallelization across samples.

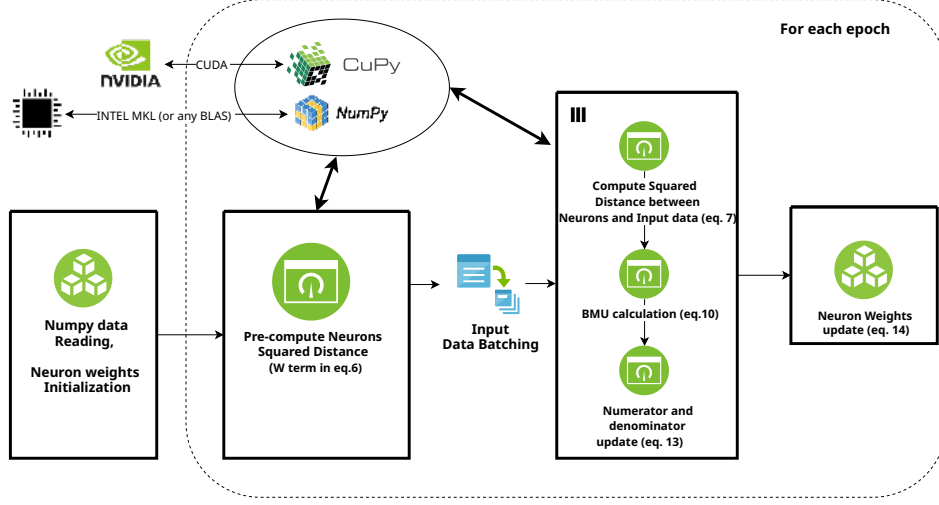


Figure 1: Visualization of the data flow for SOM training in *XPySom* from our published conference paper [13].

4.4. Weights update

To update the weights at the end of the epoch, we need to keep track of the numerator matrix $\mathcal{N} \in \mathbb{R}^{M \times N}$ and the denominator vector $\mathcal{D} \in \mathbb{R}^M$ as per Equation (3). These are updated after each batch by computing the updates $\mathcal{N}'$ and $\mathcal{D}'$ as in eq. (13), and accumulating them on $\mathcal{N} \leftarrow \mathcal{N} + \mathcal{N}'$ and $\mathcal{D} \leftarrow \mathcal{D} + \mathcal{D}'$.

$$\begin{aligned} \mathcal{N}' &= H^T \cdot X, \text{ where } n'_{j,k} = \sum_{i=1}^B h_{i,j} x_{i,k} \\ \mathcal{D}' &= \sum_{row} H, \text{ where } d'_k = \sum_{i=1}^B h_{i,k} \end{aligned} \quad (13)$$

Finally, at the end of the epoch, after all batches have been processed, the new weights $W^{[t+1]}$ are obtained by division:

$$W^{[t+1]} = \text{diag}(\mathcal{D})^{-1} \mathcal{N}, \text{ where } w_{j,k}^{[t+1]} = n_{j,k} / d_j. \quad (14)$$

5. XPySOM Implementation Details

XPySom implements the “Matrix approach” described above. The code is a fork of the popular *MiniSom* [27] library, which is a very simple and versatile library that implements the *online learning* algorithm for Self Organizing Maps in Python.

The source code for *XPySom* is openly available on GitHub² under a GPLv3 license.

XPySom has been designed to run on top of any Python library implementing the *NumPy* API. Therefore, it is able to run without changes on GPUs using *CuPy*, a CUDA-powered *NumPy*-like library for Python. Internally, *CuPy* uses a combination of custom kernels, and calls to the *cuBLAS* library to efficiently implement all operations.

The resulting data flow is exemplified in Figure 1, where the various steps *XPySom* goes through in order to perform a SOM training operation are visualized, in connection with the corresponding equations defined in the previous subsections. Core features of the *XPySom* architecture are its capability to perform SOM training operations in batches of input samples, arranged as matrix/vector operations that are executed very efficiently through relatively few calls to the underlying NumPy or CuPy libraries. *XPySom* is able to use either of them interchangeably, thanks to their compatible APIs.

6. Experimental Results

In this section, our proposed *XPySom* is evaluated, presenting extensive experimental results comparing its training and inference performance and accuracy to other open-source solutions.

First, in Section 6.1 the experimental setup for the experiments is presented. Next, in Section 6.2

the distance calculation is benchmarked across the different algorithms shown in Section 4. Then, in Section 6.3 the execution time of the different SOM training implementations is compared against *XPySom*. In Section 6.4, the quantization error of the algorithms is discussed. Finally, Section 6.5 evaluates the performance of the SOM training with different distances in *XPySom* and *Somoclu*.

6.1. Experimental setup

The experiments have been performed on an NVIDIA DGX Station⁶ server featuring 4x NVIDIA Tesla V100 GPUs, an Intel Xeon E5-2698 v4 CPU, and 256 GiB of system memory. For the experiments, only one of the GPUs was used. The NVIDIA Tesla V100 uses the Volta architecture (compute capability 7.0), and has 32 GiB of memory and 5120 CUDA cores. The Intel Xeon E5-2698 v4 CPU is a 20-core CPU with SMT, it has a base clock of 2.20 GHz, and a boost clock of 3.60 GHz.

Software Configuration. Experiments were conducted on Ubuntu 18.04 with Linux kernel 4.15.0 in a Docker container⁷ with CUDA 11.7, CuDNN 8, Ubuntu 20.04, and Python 3.10. The NVIDIA driver version used in the experiments is 515.65.01. Key packages: NumPy 1.23.x (Intel MKL build from Anaconda channel), CuPy 11.3.0, TensorFlow 2.10.0, Somoclu (commit 63895f4), SOMvec 0.1.2, SuSi 1.2.2, MiniSom (commit 0540834), and scikit-learn 1.1.x. Thread parallelism used MKL/OpenMP defaults to use all available CPU cores. GPU execution used a single V100 for benchmarks.

6.2. Distance benchmarking

6.2.1. Compared algorithms

In this section, the performance of the following distance algorithms are compared, to show the impact of the different optimizations in Section 6.5: *Euclidean distance matrix*: the algorithm for calculating the pair-wise euclidean distance between two matrices from Equation (6).

Euclidean distance matrix optimized: the algorithm for calculating the pair-wise euclidean distance between two matrices with the optimizations from Equation (7).

Norm-p non-optimized: the algorithm for calculating the pair-wise norm-p distance using the straightforward calculation from Equation (8). This is equivalent to Equation (5) for $p = 2$.

Norm-p optimized (even): the optimized algorithm for calculating the pair-wise norm-p distance in Equation (9).

6.2.2. Euclidean distance calculation time by input sample size

Figure 2 shows the computing time of the euclidean distance for the different algorithms, at varying input sample sizes. On the left, the performance on GPU is shown using *cupy*. We can observe that every algorithm, except the non-optimized norm-2, has practically the same execution time until 10000 samples. This is due to the overhead of executing on GPU dominating the actual on-GPU execution time. The difference between the different algorithms is determined by the number of computations that need to be performed, with the “optimized matrix” algorithm dominating the others, followed by the “matrix” (non-optimized), and the “norm-2 optimized”. The “norm-2 non-optimized” algorithm is actually as fast as the “matrix optimized” algorithm up to 1000 samples, due to the simpler formulation yielding less on-GPU operations. However, its inefficiency is evident at 10k samples, where it is up to 10x slower than the others. At 100k samples, the “matrix optimized” algorithm is 4x faster than the “matrix” (non-optimized) at 300 microseconds, and 5x faster than “norm-2 optimized”. “Norm-2 non-optimized” is 67x slower at 20ms.

On the right, the CPU performance is shown using *numpy*. In this case, the execution time is almost linearly dependent on the input sample size after 1000 samples, where the effect of fixed overheads gets offset. Note how the “norm-2 optimized” algorithm is faster than the “matrix” (non-optimized) algorithm on CPU. The simple calculations of “norm-2 non-optimized” are always slower than the rest. At 100k samples, the “matrix optimized” algorithms lead the pack with 10ms (33x slower than on GPU), followed by the “norm-2 optimized” one (20x slower), and by the “matrix” (non-

⁶<https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/dgx-station/nvidia-dgx-station-datasheet-uk.pdf>

⁷From the following NVIDIA CUDA base image: `nvcr.io/nvidia/cuda:11.7.0-cudnn8-devel-ubuntu20.04`

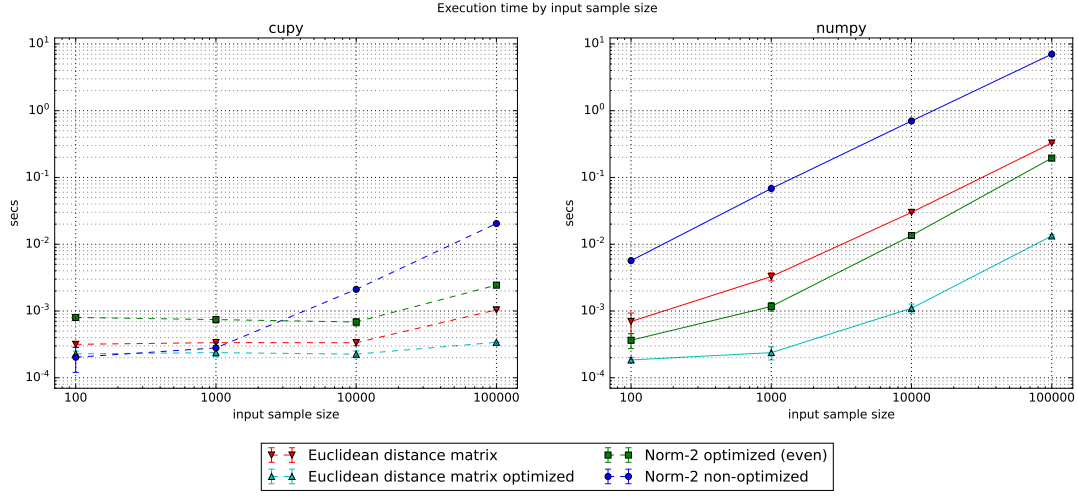


Figure 2: Distance calculation time for a grid of 16x16 neurons (256) and 300 dimensions for different values of the input sample array size. The average of 10 runs is shown with 95% CI.

optimized) algorithm (30x slower, but only 50% slower than the norm-2 optimized), and finally the “norm-2 non-optimized” (700x slower at 7 seconds, 350x slower than GPU).

6.2.3. Euclidean distance calculation time by grid size

Figure 3 shows the time to compute the euclidean distance for the different algorithms, at varying numbers of neurons (M) in the grid. The same rankings among algorithms can be observed as in the previous section. The only difference is that on CPU the matrix algorithm is faster than the norm-2 optimized one for big grid sizes, showing almost no difference in execution time across different grid sizes. The relative speeds between algorithms remain similar as before.

Figure 4 compares the euclidean distance computation times for the different algorithms, at varying numbers of dimensions, where similar observations as above can be made.

6.2.4. Norm- p distance calculation time by power

Finally, Figure 5 shows the time to compute the norm- p distance for the different algorithms, at varying values of the power p . We can note how the execution time of the non-optimized algorithm is approximately constant at 20ms for GPU and 30s on CPU, with the only exception of $p = 2$ on CPU at 7s. On GPU, the optimized algorithm is faster

than the non optimized up to 16 power p . It is 5x, 3x, and 2x faster, for 4, 8, and 16, respectively. Also on CPU, the optimized algorithm is faster than the non optimized up to 16 power p . It is 10x, 3x, and 1.5x faster, for 4, 8, and 16, respectively.

This is coherent with the expectations: the simple algorithm does the same computations irrespectively of p , while the optimized algorithm has to do linearly more computations to compute all the terms of the binomial expansion. At $p = 2$ the squaring operation is optimized by a simple multiplication instruction on CPU, resulting in a faster computation.

6.2.5. Final remarks

In summary, the optimized algorithm for calculating the euclidean distance using matrices is faster than the simpler matrix algorithm, and much faster than the textbook equation. The norm- p optimization presented in Section 4.1.2 is also significantly faster than the straightforward formula, but only for small values of p (up to 16), after which it becomes slower than the non-optimized formulation, since the latter does not depend on p for the execution time.

6.3. SOM benchmarks

6.3.1. Compared SOM implementations

Among the multiple approaches available in the literature, only a few of them have been made

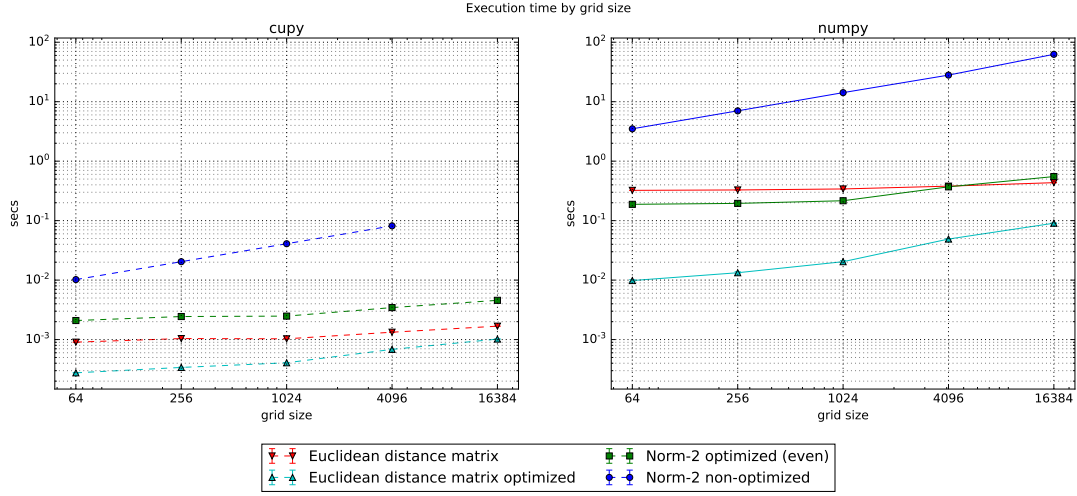


Figure 3: Distance calculation time for an input data array of size 100,000, and 300 dimensions for different grid sizes (number of neurons). The average of 10 runs is shown with 95% CI. The last data point of “Norm-2 non-optimized” on GPU is missing due to an out-of-memory error.

widely available as open-source projects exhibiting a Python API, actively maintained and well documented with online examples. For these reasons, the following projects were chosen to compare against *XPySom*:

MiniSom [27], a widely-used Python library based on the *online learning* algorithm. Due to its simplicity, and wide usage, its codebase has been used as a starting point for *XPySom*, which implements the same API, and can be used as a drop-in replacement for it. Despite its simplicity, *MiniSom* does not make use of parallel computation, relying on *Numpy* to optimize vector-to-vector and vector-to-matrix operations. The processing of only one input data vector at a time cripples its performance due to the limited parallelism and frequent slow Python code calls.

Somoclu [29, 30], a widely-used C++ implementation of the *batch learning* algorithm using MPI, OpenMP and CUDA, that provides interfaces into several other programming languages, including Python. It is very flexible, and able to execute on single to multicore systems, with GPUs, and across clusters.

TensorFlow SOM [28], implements *batch* SOMs in Python using the well-known *TensorFlow* framework, and can run on multiple processors, and both single and multiple GPUs.

SuSi [32, 33], which implements both the *online*

and *batch algorithms* on CPU.

SOMvec [31], providing *online learning* on CPU.

Both *online* and *batch* algorithm implementations have been included for completeness. Implementations of the *online* algorithm don’t have a GPU implementation, nor a parallel CPU implementation, resulting slower in time for each epoch, but converge with fewer epochs. This trade-off is explored in section 6.4.

6.3.2. Experimental setup

Different tests have been performed comparing different training situations. Each test run has been repeated 3 times with a 30-minute timeout per run (the timeout is shown as a red horizontal line on the plots). All tests below use the *Extended MNIST* (EMNIST) dataset [26] that contains 240000 data samples, each composed of 784 features (28x28 gray-scale images). In all the experiments, the input data values are scaled in the $[0, 1]$ range by dividing by 255, and are represented using 32-bit floating point numbers. In the experiments that use different numbers of features, all input images have been rescaled to the desired size.

In the *TFSom* implementation, the batch size is set to 1000, which shows a nice compromise in terms of memory usage and computation time. The learning rate starts every time with a value of 0.5 and decays exponentially over epochs in all the

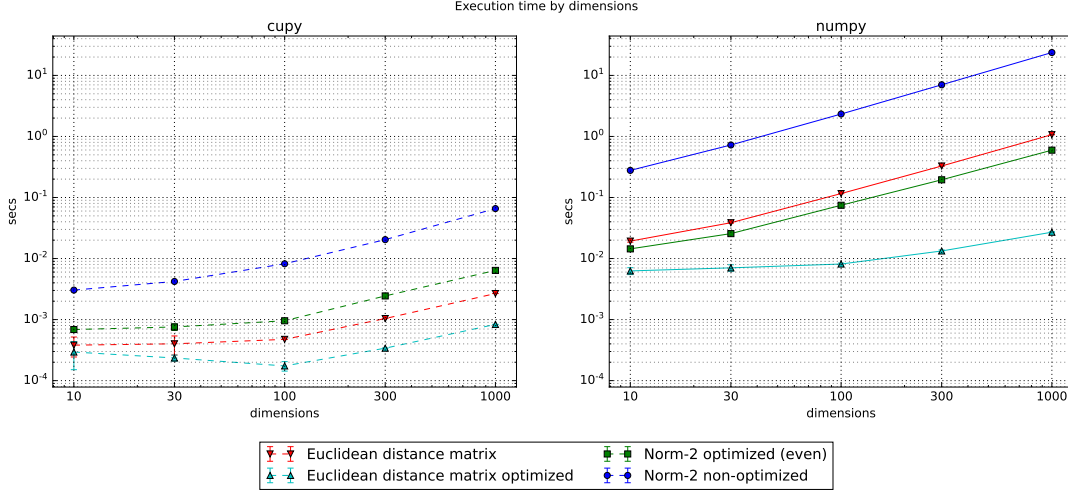


Figure 4: Distance calculation time for an input data array of size 100,000, and a grid of 16x16 neurons (256) for number of dimensions. The average of 10 runs is shown with 95% CI.

tested implementations.

6.3.3. Training time at varying training samples

In this experiment, the number of epochs, features, and neurons are kept constant at 10, 784, and 100 respectively, while the number of input data samples is varied by randomly sampling the EMNIST dataset. The results are shown in Figure 6. Note that the training time increases linearly with the number of samples (the graph uses a logarithmic y-axis).

On GPU, the implementations of *Somoclu* and *TFSom* take approximately the same time, from 20s at 30k samples to 150s at 240k samples. On the other hand, *XPySom* is 500x faster, going from 40ms at 30k samples to 300ms at 240k samples. It seems that both *Somoclu* and *TFSom* do not fully utilize the GPU, and instead show significant load on the CPU, which indicates a non-optimal use of the accelerator resources.

On CPU, *XPySom* is still the fastest, going from 1.2s to 10s (30x slower than on GPU). *Somoclu* follows but is only 4x slower, while *TFSom* is 20x slower, and *MiniSom* and *SOMvec* are both 100x slower than *XPySom*. *SuSi online* times out after half of the dataset (120k), where it is 200x slower than *XPySom*. Its *batch* counterpart performs worse, taking more than double the time and timing out at 60k samples (a fourth of the dataset).

Comparing these findings with our prior conference results [13] shows how upgrading to the server-class heterogeneous platform yields a 10× speedup for *XPySom* on the GPU (Tesla V100 vs. GTX 1080 Ti), but results in a 2× slowdown on the CPU (Xeon E5-2698 v4 vs. Core i7-4790K) [13] due to platform differences, including lower clock frequency. In terms of relative performance against the baseline configurations: (1) On the GPU, *XPySom*’s advantage over the best alternative has expanded from 100× faster on the older platform [13] to 500× faster on the newer one, where alternative GPU implementations fail to fully utilize the upgraded hardware resources. (2) On the CPU, *XPySom*’s performance gap over the best multi-core competitor has narrowed from 7× faster on the desktop Core i7 [13] to 4× faster on the server Xeon.

6.3.4. Training time with different SOM grid sizes

In this experiment, the number of input data samples, features, and epochs are kept constant at 240000, 784, and 10 respectively, while the number of neurons composing the grid is changed. The results are shown in Figure 7.

As expected, the training time (y-axis) increases with the number of neurons (x-axis). Both in CPU and GPU implementations, *XPySom* is the fastest, outperforming all others.

On GPU, the second best is *TFSom* that scales

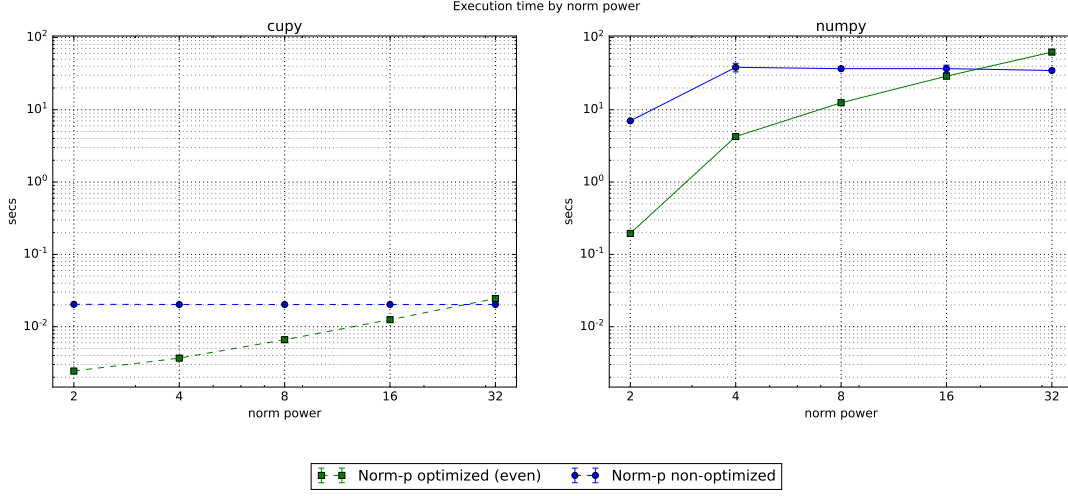


Figure 5: Norm-p distance calculation time for an input data array of size 100,000, a grid of 16x16 neurons (256), and 300 dimensions, for different values of the power p . The average of 10 runs is shown with 95% CI.

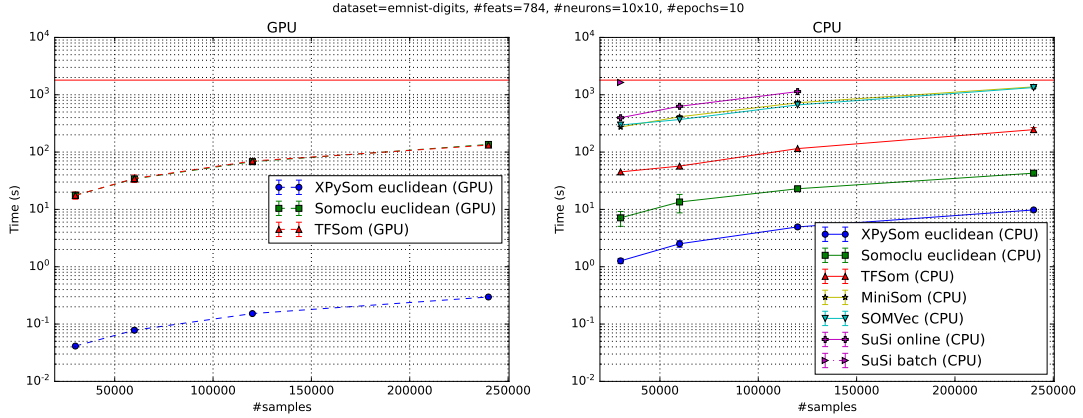


Figure 6: Training time as a function of the number of training samples (EMNIST, 240000 samples, 784 features)

much better than the others with the number of neurons, outperforming *Somoclu* after 100 neurons. Both are however, much slower than *XPySom* as in the previous experiment. *XPySom* takes only 300ms on a 10x10 grid, beating both other implementations by more than 500x.

On CPU, the other algorithms are closer to *XPySom*, with the resulting order being *XPySom* (30x slower than on GPU on a 10x10 grid), *Somoclu* (3x slower than *XPySom* on a 10x10 grid), *TFSom* (20x slower than *XPySom* on a 10x10 grid), *SOMVec* and *MiniSom*. *SuSi* times out both in the online and batch implementations.

6.3.5. Training time at varying training epochs

In this third experiment, the number of input data samples, features, and neurons are kept constant at 240000, 784, and 100 respectively, while the number of epochs is changed. The results are shown in Figure 8.

In this experiment, we can notice how the training time increases linearly with the number of epochs. Furthermore, *XPySom* takes less than 2 seconds on GPU to compute even 100 epochs, which is two orders of magnitude faster than *TFSom* and almost three orders of magnitude faster than *Somoclu*.

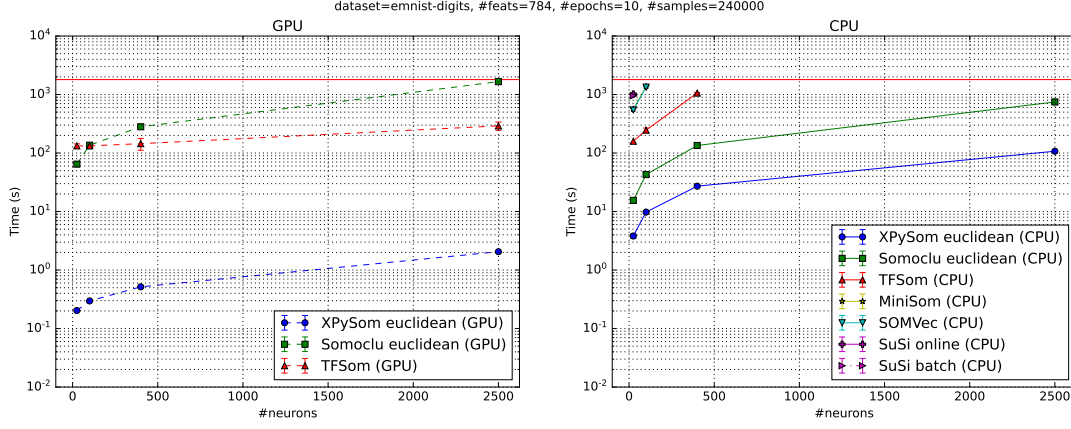


Figure 7: Training time as a function of the number of neurons (EMNIST, 240000 samples, 784 features).

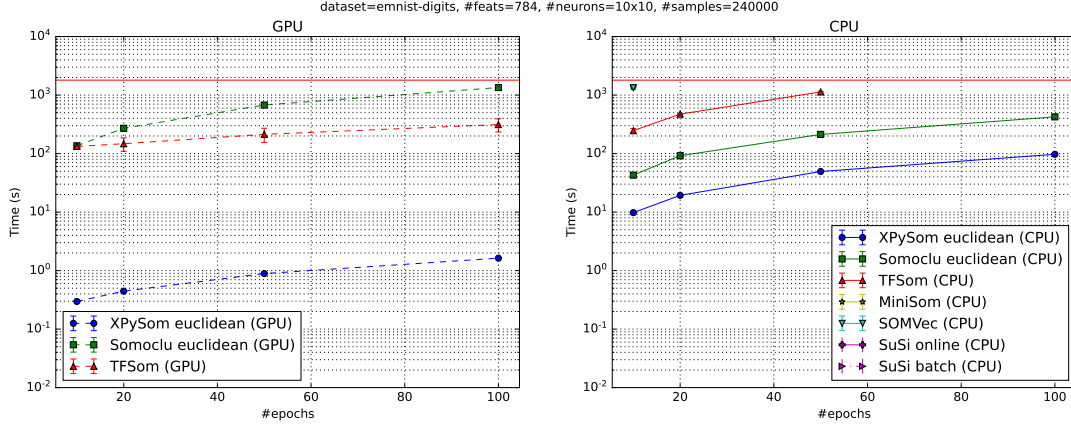


Figure 8: Training time as a function of the number of training epochs (EMNIST, 240000 samples, 784 features)

For the rest, we notice the same behavior and ranking between the algorithms as in the previous experiment, with many more timeouts caused by the elevated number of epochs.

6.3.6. Training time at varying input features

In this final experiment, the number of epochs, data samples, and neurons are kept constant at 10, 240000, and 100 respectively, while the number of features is changed, by resizing the images through interpolation using OpenCV in a range from 7x7 to 56x56. The results are shown in Figure 9.

The TensorFlow implementation, *TFSom*, cannot start the training session on GPU with 3136 features due to an out-of-memory error using a batch size of 1000, while it times out on the same

picture size on CPU. Furthermore, *SOMvec*, *MiniSom*, and *SuSi* time out on the last data point.

At 49 dimensions, for the first time *Somoclu* records an execution time close to *XPySom*, suggesting that it might perform similarly on datasets with low dimensionality.

6.3.7. Discussion

In almost all benchmarks, *XPySom* is up to two orders of magnitude faster on the GPU and one order of magnitude faster on the CPU compared to alternative implementations. The only exception occurs when the input dataset has fewer than 100 features, where *Somoclu* achieves performance comparable to *XPySom*. However, real-world SOM applications typically exhibit high-dimensional inputs.

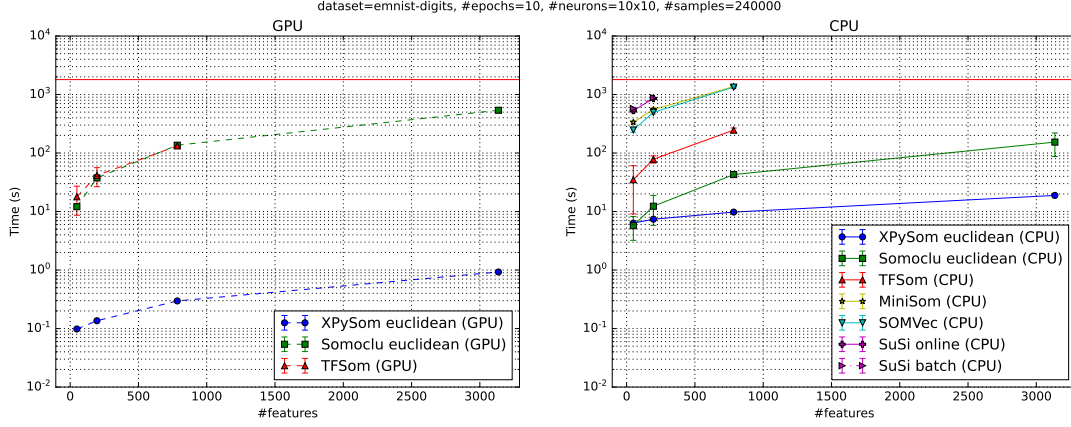


Figure 9: Execution time vs Increasing number of training features (EMNIST, 240000 samples, 784 features)

Furthermore, all evaluated *online learning* implementations remain substantially slower than the batch-based versions, despite promoting speedups via optimized per-sample updates.

6.4. Quantization error measurements

To verify that the compared SOM algorithms were actually working and converging towards a solution, we computed the average quantization error over 3 runs on the full EMNIST dataset, on a 10x10 grid, over 10 epochs. *SuSi* was excluded to this evaluation due to timeout issues in computing the quantization error using the built-in method, while *SOMvec*, *MiniSom* and *TFSom* (CPU) timed out from 20 epochs onwards.

The results are shown in Figure 10 that shows how *Somoclu* and *XPySom* are very close, since they both implement the batch algorithm. *MiniSom* achieves the fastest convergence in terms of epochs, given the use of the online learning algorithm. However, it is important to note that even at only 10 epochs its execution time is more than 10x slower than *XPySom* (CPU) at 100 epochs, which has a similar quantization error. Furthermore, *SOMvec* which also implements the online algorithm has the worst quantization error, and also times out at 20 epochs. Finally, *TFSom* (GPU) does not seem to get a better quantization error with increasing number of epochs, which raises some doubts on the implementation, but are consistent with what previously observed in [13].

6.5. Evaluation of different distances

In this experiment, we evaluate the impact of different distances on the training time of the SOM in *XPySom* and *Somoclu*. The experiment is performed by changing the number of training features, while keeping constant the number of samples (240k), epochs (10), and neurons (10x10). The results are shown in Figure 11.

From the results we can note how the non-optimized matrix euclidean distance from Equation (6) is 50% slower on GPU and 3x slower on CPU than the optimized version used by default in *XPySom*. We can note how the edge of *XPySom* over *Somoclu* is actually coming mostly from this optimization, as using the non-optimized euclidean distance in *XPySom* achieves a similar performance to *Somoclu*.

The norm-8 (Minkowski norm with $p = 8$) is 3x slower than the euclidean norm on GPU and 10x slower on CPU. The implementation in *Somoclu* (only for CPU) is slightly faster than *XPySom*, except for the last point with 3k features.

7. Conclusions

This paper has presented the design and implementation of *XPySom*, an open-source SOM implementation for Python that successfully exploits the parallelization potential of *batch learning*. By reformulating the training algorithm into matrix operations, we leverage highly optimized linear algebra

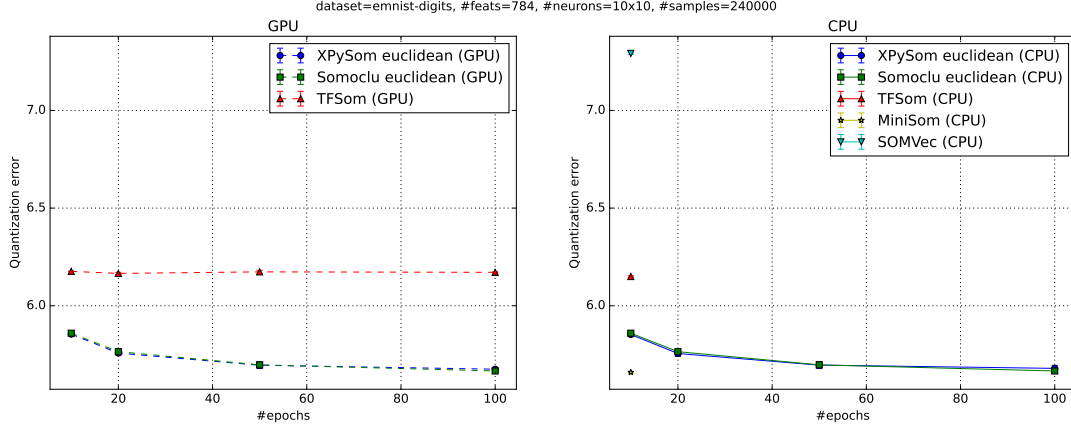


Figure 10: Evolution of the quantization error for a 10x10 SOM throughout training epochs (EMNIST, 240000 samples, 784 features).

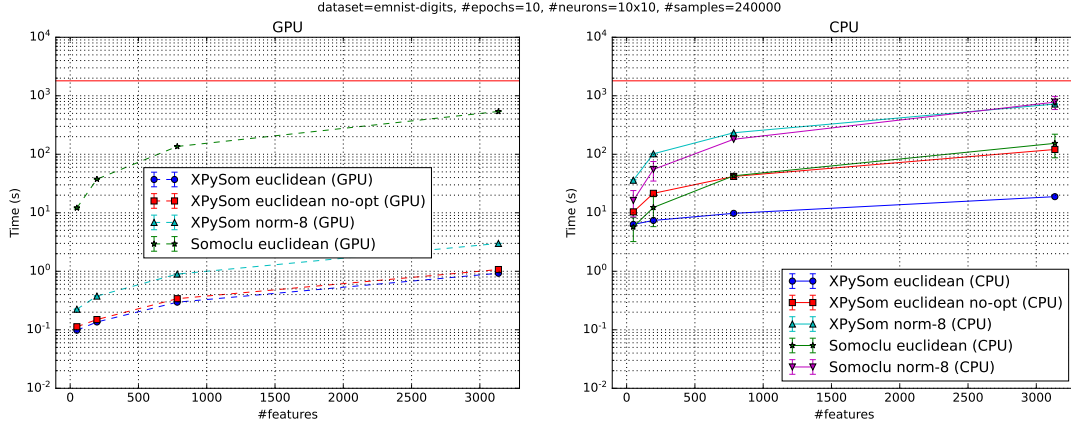


Figure 11: Execution time vs Increasing number of training features for multiple distances (EMNIST, 240000 samples, 10 epochs, 100 neurons)

routines via NumPy and CuPy. This design results in a high-performance yet minimalistic code-base that remains easy to maintain, adapt, and extend.

XPySom has been experimentally tested on multicore and GPU platforms, outperforming other available open-source implementations, including the popular *Somoclu* package. For example, on the EMNIST dataset, *XPySom* was as much as two or three orders of magnitude faster, obtaining similar quantization error results.

We also performed a similar comparison on a (non-public) time-series dataset from the Telco sector [55], where the daily resource consumption pat-

terns of thousands of VMs in a production NFV environment have been clustered using *XPySom* and the other alternatives considered in this paper. We obtained essentially similar results and conclusions about the performance of *XPySom* compared to other open-source implementations. More details about this dataset are available in [55], but the dataset is not openly available, so we preferred to include in this paper only results from the EMNIST dataset, for the sake of reproducibility.

8. Future Work

Further investigations could be performed in the future, on the topics presented in this paper, ei-

ther using *XPySom*, or by enhancing it further. For example, the current *XPySom* implementation is limited to the use of a single GPU and a single machine. However, it is common to have multi-GPU platforms, or even multi-node clusters, nowadays. By cleverly distributing the input data across different GPUs, or even different computing nodes, it could be possible to further improve the performance on multi-GPU architectures for very large datasets. Furthermore, the host CPU remains largely idle while the GPU processes the input data samples. A promising future direction is to leverage these idle CPU resources concurrently to further accelerate training. Our study focused on Euclidean and norm-p distance metrics only, probably the most popular choices when using SOMs; however, integrating further distance metrics in *XPySom*, and evaluating their impact on the obtained performance, might be an interesting future extension. Finally, additional experimental evaluations could be performed possibly on additional datasets, for a more detailed assessment of the achievable convergence, stability, and accuracy trade-offs, or even comparing with other and more recent clustering methods, including deep-learning based ones [56]. In this context, advanced representation-learning backbones, such as token-selective transformers [57] or state-space models [58], could be explored in future extensions to extract compact, localized features from high-dimensional inputs before applying topological clustering.

Despite these current limitations, *XPySom* provides a highly efficient tool that other researchers and practitioners can readily leverage, extend, and integrate to build high-performance, SOM-based services. Indeed, *XPySom* was recently adapted to support cutting-edge biomedical research published in *Nature* [59].

References

- [1] S. Greengard, The Internet of Things, revised and updated edition, The MIT Press, 2021.
- [2] M. D. Assunção, R. N. Calheiros, S. Bianchi, M. A. Netto, R. Buyya, Big Data computing and clouds: Trends and future directions, *Journal of Parallel and Distributed Computing* 79-80 (2015) 3–15, Special Issue on Scalable Systems for Big Data Management and Analytics.
- [3] X. Liu, R. Buyya, Performance-Oriented Deployment of Streaming Applications on Cloud, *IEEE Transactions on Big Data* 5 (1) (2019) 46–59.
- [4] R. Andreoli, R. Mini, P. Skarin, et al., A multi-domain survey on time-criticality in cloud computing, *IEEE Transactions on Services Computing* (2025) 1–19.
- [5] C. Silvano, D. Ielmini, F. Ferrandi, et al., A survey on deep learning hardware accelerators for heterogeneous hpc platforms, *ACM Comput. Surv.* 57 (11) (Jun. 2025).
- [6] E. Quinones, S. Royuela, C. Scordino, et al., The AMPERE Project: A Model-driven development framework for highly Parallel and EneRgy-Efficient computation supporting multi-criteria optimization, in: 23rd IEEE International Symposium on Real-Time Distributed Computing, Nashville, TN, USA, May 19-21, 2020, IEEE, 2020, pp. 201–206.
- [7] H. Cho, J. Lee, J. Lee, FARNN: FPGA-GPU Hybrid Acceleration Platform for Recurrent Neural Networks, *IEEE Transactions on Parallel and Distributed Systems* 33 (7) (2022) 1725–1738.
- [8] H. Chen, J. Zhang, Y. Du, et al., Understanding the Potential of FPGA-Based Spatial Acceleration for Large Language Model Inference, *ACM Trans. Reconfigurable Technol. Syst.* (Apr. 2024).
- [9] T. Cucinotta, A. Amory, G. Ara, F. Paladino, M. Di Natale, Multi-criteria Optimization of Real-time DAGs on Heterogeneous Platforms under P-EDF, *ACM Trans. Embed. Comput. Syst.* 23 (1) (jan 2024).
- [10] D. C. Ciresan, U. Meier, J. Masci, L. M. Gambardella, J. Schmidhuber, Flexible, High Performance Convolutional Neural Networks for Image Classification, in: T. Walsh (Ed.), *Proc. 22nd International Joint Conference on Artificial Intelligence, IJCAI/AAAI*, 2011, pp. 1237–1242.
- [11] A. Krizhevsky, I. Sutskever, G. E. Hinton, ImageNet Classification with Deep Convolutional Neural Networks, in: F. Pereira, C. J. C. Burges, L. Bottou, K. Q. Weinberger (Eds.), *Advances in Neural Information Processing Systems* 25, 2012, pp. 1097–1105.
- [12] Y. Jiang, Y. Zhu, C. Lan, et al., A Unified Architecture for Accelerating Distributed DNN Training in Heterogeneous GPU/CPU Clusters, in: 14th USENIX Symposium on Operating Systems Design and Implementation, USENIX Association, 2020, pp. 463–479.
- [13] R. Mancini, A. Ritacco, G. Lanciano, T. Cucinotta, XPySom: High-Performance Self-Organizing Maps, in: *Proc. 32nd IEEE International Symposium on Computer Architecture and High Performance Computing*, IEEE, 2020, pp. 209–216.
- [14] J. Lachmair, T. Mieth, R. Griessl, J. Hagemeyer, M. Porrmann, From CPU to FPGA — Acceleration of self-organizing maps for data mining, in: *Proc. International Joint Conference on Neural Networks (IJCNN)*, Vol. 2017-May, IEEE, 2017, pp. 4299–4308.

- [15] S. Jovanović, H. Hikawa, A survey of hardware self-organizing maps, *IEEE Transactions on Neural Networks and Learning Systems* 34 (11) (2023) 8154–8173.
- [16] L. Zhang, A medical image segmentation methods based on SOM and wavelet transforms, in: *MIPPR 2019: Automatic Target Recognition and Navigation*, Vol. 11429, SPIE, 2020, pp. 25–30.
- [17] L. A. Martins, F. L. Pádua, P. E. Almeida, Automatic detection of surface defects on rolled steel using Computer Vision and Artificial Neural Networks, in: *IECON 2010 - 36th Annual Conference on IEEE Industrial Electronics Society*, 2010, pp. 1081–1086.
- [18] H. Ouidadi, S. Guo, C. Zamiela, L. Bian, Real-time defect detection using online learning for laser metal deposition, *Journal of Manufacturing Processes* 99 (2023) 898–910.
- [19] M. Vannucci, V. Colla, A. Giacomini, AI Data Analysis and SOM for the Monitoring and Improvement of Quality in Rolled Steel Bars, in: A. Rocha, H. Adeli, G. Dzemmyda, F. Moreira, V. Colla (Eds.), *Information Systems and Technologies*, Springer Nature Switzerland, Cham, 2024, pp. 490–499.
- [20] D. C. Le, N. Zincir-Heywood, M. I. Heywood, Unsupervised Monitoring of Network and Service Behaviour Using Self Organizing Maps, *Journal of Cyber Security and Mobility* 8 (1) (2018) 15–52.
- [21] G. Lanciano, A. Ritacco, T. Cucinotta, et al., SOM-based behavioral analysis for virtualized network functions, in: *Proc. 35th Annual ACM Symposium on Applied Computing*, ACM, New York, NY, USA, 2020, pp. 1204–1206.
- [22] T. Cucinotta, G. Lanciano, A. Ritacco, et al., Behavioral Analysis for Virtualized Network Functions: A SOM-based Approach, in: *Proc. 10th International Conference on Cloud Computing and Services Science, SCITEPRESS - Science and Technology Publications*, 2020, pp. 150–160.
- [23] C. R. Harris, K. J. Millman, S. J. van der Walt, et al., Array programming with NumPy, *Nature* 585 (7825) (2020) 357–362.
- [24] R. Okuta, Y. Unno, D. Nishino, S. Hido, C. Loomis, Cupy: A numpy-compatible library for nvidia gpu calculations, in: *Proceedings of Workshop on Machine Learning Systems (LearningSys)*, 31st Annual Conference on Neural Information Processing Systems (NIPS), 2017.
- [25] Intel, Developer Guide for Intel oneAPI Math Kernel Library for Linux, <https://www.intel.com/content/www/us/en/develop/documentation/oneapi-linux-developer-guide/top.html>.
- [26] G. Cohen, S. Afshar, J. Tapson, A. van Schaik, EM-NIST: an extension of MNIST to handwritten letters, *CoRR abs/1702.05373* (2017).
- [27] G. Vettigli, MiniSom (2019). URL <https://github.com/JustGlowing/minisom>
- [28] C. Gorman, TensorFlow Self-Organizing Map (2019). URL <https://github.com/cgorman/tensorflow-som>
- [29] P. Wittek, S. Daranyi, A GPU-Accelerated Algorithm for Self-Organizing Maps in a Distributed Environment, in: *Proceedings of ESANN-12, 20th European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning*, 2012.
- [30] P. Wittek, S. C. Gao, I. S. Lim, L. Zhao, somoclu: An Efficient Parallel Library for Self-Organizing Maps, *Journal of Statistical Software* 78 (9) (jun 2017).
- [31] G. Clark, SOMvec, <https://bitbucket.org/GeoffreyClark/somvec/src/master/> (2020).
- [32] F. M. Riese, SuSi: SUPervised Self-organizing maps in Python (2019). URL <https://doi.org/10.5281/zenodo.2609130>
- [33] F. M. Riese, S. Keller, S. Hinz, Supervised and Semi-Supervised Self-Organizing Maps for Regression and Classification Focusing on Hyperspectral Data, *Remote Sensing* 12 (1) (2020).
- [34] T. Kohonen, The Self-Organizing Map, *Proc. IEEE* 78 (9) (1990) 1464–1480.
- [35] A. Singhal, Modern information retrieval: A brief overview, *IEEE Data Eng. Bull.* 24 (4) (2001) 35–43.
- [36] R. C. de Amorim, B. G. Mirkin, Minkowski metric, feature weighting and anomalous cluster initializing in K-Means clustering, *Pattern Recognit.* 45 (3) (2012) 1061–1075.
- [37] W. Natita, W. Wiboonsak, S. Dusadee, Appropriate learning rate and neighborhood function of self-organizing map (som) for specific humidity pattern classification over southern thailand, *International Journal of Modeling and Optimization* 6 (1) (2016) 61.
- [38] A. A. Akinduko, E. M. Mirkes, Initialization of Self-Organizing Maps: Principal Components Versus Random Initialization. A Case Study.
- [39] F. Mulier, V. Cherkassky, Self-organization as an iterative kernel smoothing process, *Neural computation* 7 (6) (1995) 1165–1177.
- [40] M. A. Souza, L. A. Maciel, P. H. Penna, H. C. Freitas, Energy Efficient Parallel K-Means Clustering for an Intel® Hybrid Multi-Chip Package, in: *30th International Symposium on Computer Architecture and High Performance Computing*, IEEE, 2018, pp. 372–379.
- [41] D. M. Chan, R. Rao, F. Huang, J. F. Canny, T-SNE-CUDA: GPU-Accelerated T-SNE and its Applications to Modern Data, in: *30th International Symposium on Computer Architecture and High Performance Computing*, IEEE, 2018, pp. 330–338. [arXiv:1807.11824](https://arxiv.org/abs/1807.11824).

- [42] T. V. Sang, R. Kobayashi, R. S. Yamaguchi, T. Nakata, Accelerating Solution of Generalized Linear Models by Solving Normal Equation Using GPGPU on a Large Real-World Tall-Skinny Data Set, in: 31st International Symposium on Computer Architecture and High Performance Computing, IEEE, 2019, pp. 112–119.
- [43] Intel Whitepaper: Optimizing Performance with Intel Advanced Vector Extensions (2014).
- [44] NVIDIA, CUDA Toolkit Documentation: cuBLAS API Reference, <https://docs.nvidia.com/cuda/cublas/index.html>.
- [45] A. Rauber, P. Tomsich, D. Merkl, parSOM: a parallel implementation of the self-organizing map exploiting cache effects: making the SOM fit for interactive high-performance data analysis, in: Proc. International Joint Conference on Neural Networks. Neural Computing: New Challenges and Perspectives for the New Millennium, Vol. 6, IEEE, 2000, pp. 177–182 vol.6.
- [46] T. Richardson, E. Winer, Extending parallelization of the self-organizing map by combining data and network partitioned methods, *Advances in Engineering Software* 88 (2015) 1–7.
- [47] T. Sarazin, H. Azzag, M. Lebbah, SOM Clustering Using Spark-MapReduce, in: 2014 IEEE International Parallel & Distributed Processing Symposium Workshops, IEEE, 2014, pp. 1727–1734.
- [48] S. J. Plimpton, K. D. Devine, MapReduce in MPI for Large-Scale Graph Algorithms, *Parallel Comput.* 37 (9) (2011) 610–632.
- [49] NVIDIA, CUDA Toolkit Documentation: Thrust API Reference, <https://docs.nvidia.com/cuda/thrust/index.html>.
- [50] S. McConnell, R. Sturgeon, G. Henry, A. Mayne, R. Hurley, Scalability of Self-organizing Maps on a GPU cluster using OpenCL and CUDA, *Journal of Physics: Conference Series* 341 (2012) 012018.
- [51] Y. Xiao, R.-B. Feng, Z.-F. Han, C.-S. Leung, GPU Accelerated Self-Organizing Map for High Dimensional Data, *Neural Processing Letters* 41 (3) (2015) 341–355.
- [52] Y. Liu, J. Sun, Q. Yao, et al., A Scalable Heterogeneous Parallel SOM Based on MPI/CUDA, in: J. Zhu, I. Takeuchi (Eds.), *Proceedings of The 10th Asian Conference on Machine Learning*, Vol. 95 of *Proceedings of Machine Learning Research*, PMLR, 2018, pp. 264–279.
- [53] H. O. Song, Y. Xiang, S. Jegelka, S. Savarese, Deep Metric Learning via Lifted Structured Feature Embedding, in: *IEEE Conference on Computer Vision and Pattern Recognition*, Las Vegas, NV, USA, June 27–30, 2016, IEEE Computer Society, 2016, pp. 4004–4012.
- [54] A. Samuel, Euclidean Distance Matrix Trick (2019). URL https://www.robots.ox.ac.uk/~albanie/notes/Euclidean_distance_trick.pdf
- [55] G. Lanciano, A. Ritacco, F. Brau, et al., Using self-organizing maps for the behavioral analysis of virtualized network functions, in: D. Ferguson, C. Pahl, M. Helfert (Eds.), *Cloud Computing and Services Science - 10th International Conference, CLOSER 2020, Prague, Czech Republic, May 7–9, 2020, Revised Selected Papers*, Vol. 1399 of *Communications in Computer and Information Science*, Springer, 2020, pp. 153–177.
- [56] J. Paparrizos, S. P. T. R. Bogireddy, Time-series clustering: A comprehensive study of data mining, machine learning, and deep learning methods, *Proc. VLDB Endow.* 18 (11) (2025) 4380–4395.
- [57] Y. Xiao, Q. Yuan, K. Jiang, et al., Ttst: A top-k token selective transformer for remote sensing image super-resolution, *IEEE Transactions on Image Processing* 33 (2024) 738–752. doi:10.1109/TIP.2023.3349004.
- [58] Y. Xiao, Q. Yuan, K. Jiang, et al., Frequency-assisted mamba for remote sensing image super-resolution, *IEEE Transactions on Multimedia* 27 (2024) 1783–1796. doi:10.1109/TMM.2024.3521798.
- [59] M. Kuehl, Y. Okabayashi, M. N. Wong, et al., Pathology-oriented multiplexing enables integrative disease mapping, *Nature* 644 (8076) (2025) 516–526.