Fast geometric learning with symbolic matrices

Jean Feydy[*]
Imperial College London
jfeydy@ic.ac.uk

Joan Alexis Glaunès[*]
Université de Paris
alexis.glaunes@parisdescartes.fr

Benjamin Charlier[*]
Université de Montpellier
benjamin.charlier@umontpellier.fr

Michael M. Bronstein
Imperial College London / Twitter
m.bronstein@imperial.ac.uk

Abstract

Geometric methods rely on tensors that can be encoded using a symbolic formula and data arrays, such as kernel and distance matrices. We present an extension for standard machine learning frameworks that provides comprehensive support for this abstraction on CPUs and GPUs: our toolbox combines a versatile, transparent user interface with fast runtimes and low memory usage. Unlike general purpose acceleration frameworks such as XLA, our library turns generic Python code into binaries whose performances are competitive with state-of-the-art geometric libraries – such as FAISS for nearest neighbor search – with the added benefit of flexibility. We perform an extensive evaluation on a broad class of problems: Gaussian modelling, K-nearest neighbors search, geometric deep learning, non-Euclidean embeddings and optimal transport theory. In practice, for geometric problems that involve 10^3 to 10^6 samples in dimension 1 to 100, our library speeds up baseline GPU implementations by up to two orders of magnitude.

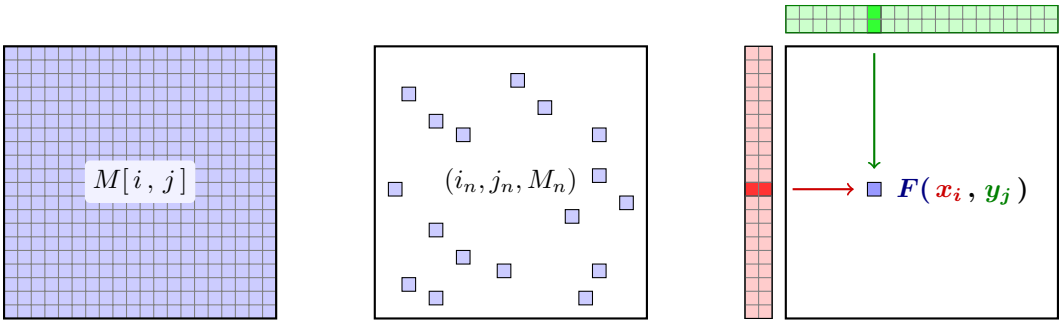
1 Introduction

Fast numerical methods are the fuel of machine learning research. Over the last decade, the sustained development of the CUDA ecosystem has driven the progress in the field: though Python is the lingua franca of data science and machine learning, most frameworks rely on efficient C++ backends to leverage the computing power of GPUs [1, 86, 101]. Recent advances in computer vision or natural language processing attest to the fitness of modern libraries: they stem from the mix of power and flexibility that is provided by PyTorch, TensorFlow and general purpose accelerators such as XLA.

Nevertheless, important work remains to be done. Geometric computations present a clear gap in performances between Python and C++: notable examples are implementations of point cloud convolutions or of the nearest neighbor search [65, 76]. To scale up geometric computations to real-world data, a common practice is therefore to replace the compute-intensive parts of a Python code by handcrafted CUDA kernels [35, 60, 92]. These are expensive to develop and maintain, which leads to an unfortunate need to compromise between ease of development and scalability.

To address this issue, we present KeOps: an extension for PyTorch, NumPy, Matlab and R that combines the speed of a handcrafted CUDA kernel with the simplicity of a high level language. Our toolbox optimizes map-reduce operations on generalized point clouds and provides transparent support for distance-like matrices, as illustrated in Figure 1. The resulting computations are fully differentiable and have a negligible memory footprint. Their runtimes are competitive with state-of-the-art CUDA libraries when they exist, and peerless in the many use cases that are not covered by existing implementations. Our library fits seamlessly within existing codebases and provides a sizeable performance boost to a wide range of methods. Among other applications, we present optimal transport solvers and geometric operators in hyperbolic spaces which are orders of magnitude faster than the state-of-the-art. We believe that our library is an important addition to the existing arsenal of tools and will have a stimulating impact on machine learning research.

34th Conference on Neural Information Processing Systems (NeurIPS 2020), Vancouver, Canada.



(a) Dense matrix (b) Sparse matrix (c) Symbolic matrix

Figure 1: Machine learning frameworks understand variables as matrices, also known as tensors. (a) These are usually **dense** and encoded as explicit numerical arrays $(M_{i,j}) = (M[i,j]) \in \mathbb{R}^{\mathrm{N}\times\mathrm{M}}$ that can have a large memory footprint. (b) Alternatively, some operators can be encoded as **sparse matrices**: we store in memory the indices (i_n, j_n) and values $M_n = M_{i_n,j_n}$ that correspond to a small number of non-zero coefficients. Reduction operations are then implemented using indexing methods and scattered memory accesses. (c) We provide support for a third class of tensors: **symbolic matrices** whose coefficients are given by a formula $M_{i,j} = F(x_i, y_j)$ that is evaluated on data arrays (x_i) and (y_j). Reduction operations are implemented using parallel schemes that compute the coefficients $M_{i,j}$ on-the-fly. We take advantage of the structure of CUDA registers to bypass costly memory transfers and achieve optimal runtimes on a wide range of applications.

2 Related work

Machine learning and data science applications often encounter the problem of computing the proximity or distance between data samples. Given $x_1, \dots, x_{\mathrm{N}}$ and $y_1, \dots, y_{\mathrm{M}} \in \mathbb{R}^{\mathrm{D}}$ two clouds of N and M points in dimension D, the bottleneck of many methods is an **interaction step** of the form:

$$a_i \leftarrow \operatorname*{\square}_{j=1}^{\mathrm{M}} F(i, j, x_i, y_j)\ , \quad \forall i \in [\![1, \mathrm{N}]\!]\ , \tag{1}$$

where F is a vector-valued formula and $\square$ is an associative reduction operator, e.g. a sum or a minimum. This paper is part of a large body of work that lowers the $\mathcal{O}(\mathrm{NM})$ computational cost of such an operation: we now recall the main approaches to this problem.

Sparse matrices. A first strategy is to prune out negligible terms: for every index i, we perform the reduction (1) on a subset of neighbors $\mathcal{N}(i) \subset [\![1, \mathrm{M}]\!]$. As illustrated in Figure 1, this method is akin to using sparse matrices: the neighborhood structure is usually understood as a connectivity matrix that comes from a triangle mesh or a K-nearest neighbors (KNN) graph [16, 69, 114]. This method can be used whenever the operation F is local but has a major limitation: at a low level, truncated reductions rely on random memory accesses that do not stream well on GPUs [25, 82]. Consequently, speed-ups are only achieved if the neighborhoods $\mathcal{N}(i)$ are orders of magnitude smaller than the full set of indices $[\![1, \mathrm{M}]\!]$ – a condition that is often too restrictive and cannot be satisfied.

Nearest neighbor finders. Going further, the implementation of KNN queries is itself a geometric problem in the mould of (1). When the datasets (x_i) and (y_j) have a small intrinsic dimension, efficient schemes can outperform brute-force approaches [11, 31, 49, 53, 65, 77, 85]. Unfortunately, these methods rely on pre-computations that are too expensive to be performed at every iteration of a training loop. Reference implementations also tend to lack flexibility and only support a handful of metrics: for instance, in spite of a strong interest for hyperbolic embeddings in the machine learning literature [72, 83], Poincaré metrics are not supported out-of-the-box by standard libraries.

Approximated convolutions. When the reduction $\square$ is a sum and $F(i, j, x_i, y_j) = k(x_i - y_j) = K_{i,j}$ is a translation-invariant kernel, the interaction (1) is understood as a discrete convolution. To speed up this operation, a first idea is to rely on low-rank decompositions of the kernel matrix $(K_{i,j})$ [90, 111, 116]. Multiscale schemes can be used to handle singular kernels [7, 8, 50, 52, 115] or compress generic operators [5, 17, 55]. Alternatively, semi-Eulerian methods rely on intermediate

grid representations to leverage fast Fourier transforms or convolution routines [32, 51, 76]. These approaches can achieve dramatic speed-ups but tend to require a significant amount of tuning for each kernel k. They work best when the latter is smooth or is defined on a space of dimension $\mathrm{D} \leqslant 3$.

Acceleration frameworks. In contrast to mathematical approaches, several compilation frameworks have been designed to speed-up machine learning architectures. Modern toolboxes accelerate a wide range of operations but are not geared towards geometric problems: most of them keep a focus on distributed learning [63, 64, 97, 108] or image processing and dense tensor manipulations [22, 58, 74, 105]. TVM [22] and CuPy [84] are the two libraries which are closer to our work: they both provide partial support for symbolic tensors. However, they have limited support for automatic differentiation and require the use of a custom low-level syntax to produce efficient binaries.

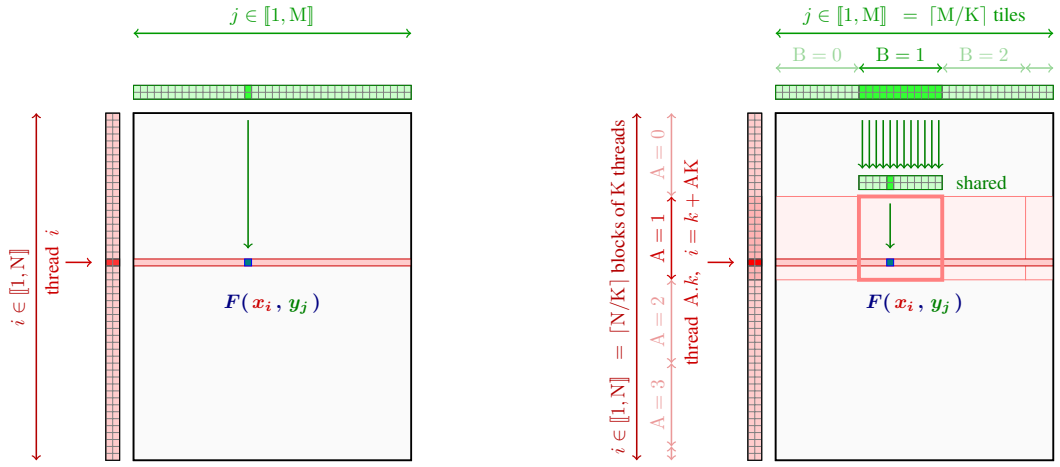
3 Motivation

Requirements for geometric data analysis and learning. None of the aforementioned methods are fully suited for modern research in geometric data analysis and machine learning. Let us briefly explain why. First of all, some acceleration schemes do not stream well on GPUs or have to rely on expensive pre-computations: hierarchical matrices [55] or advanced nearest neighbor finders [77] can hardly be used in the training loop of a neural network. Other strategies make strong assumptions on the properties of the convolution filter k or on the dimension and geometry of the ambient feature space. These restrictions make existing tools cumbersome to use in deep learning, where one wishes to have modelling freedom, e.g. w.r.t. the choice of the embedding space geometry and dimension. Finally, most acceleration frameworks for Python expect users to be knowledgeable on GPU parallelism or do not support automatic differentiation. The bottomline is that most existing tools are not ready to be used by a majority of researchers in the community.

A gap in the literature. In order to tackle these issues, the developers of deep learning libraries have recently put an emphasis on just-in-time compilation for neural networks. For instance, the recent PyTorch JIT and XLA engines enable operator fusion and unlock performance speed-ups for research code [15, 86]. These *general purpose* compilers are fully transparent to users and show promise for a wide range of applications. Nevertheless, they fall short on geometric computations along the lines of (1). This is most apparent for nearest neighbor search [36, 60, 65], matrix-vector products with kernel matrices and message passing methods on point clouds [35, 36, 102], where one still has to develop and maintain custom CUDA kernels to achieve state-of-the-art performance.

Symbolic matrices. We notice that all the aforementioned methods rely on reductions of an N-by-M matrix $(M_{i,j}) = (F(i, j, x_i, y_j))$ that is often **too large to be stored in memory as a dense tensor**. Acknowledging the fact that memory management is a bottleneck for tensor programs, we choose to focus on the fundamental concept of symbolic matrices, illustrated in Figure 1. For the first time, we provide support for this abstraction on the GPU with all the desirable features of a deep learning library: a math-friendly interface, high performance, transparent support for batch processing and automatic differentiation. The example below is representative of our user interface:

```
from torch import rand, autograd       # NumPy, R and Matlab are also supported
from pykeops.torch import LazyTensor   # Symbolic wrapper for PyTorch Tensors

# Setup data on the CPU and/or GPU with shapes (N,D), (M,D), (M,E):
N, M, D, E = 10 ** 5, 10 ** 6, 50, 100
x, y, b = rand(N, D, requires_grad=True), rand(M, D), rand(M, E)

# Perform arbitrary symbolic computations:
x_i  = LazyTensor(x.view(N, 1, D))     # (N,D) Tensor -> (N,1,D) Symbolic Tensor
y_j  = LazyTensor(y.view(1, M, D))     # (M,D) Tensor -> (1,M,D) Symbolic Tensor
D_ij = ((x_i - y_j) ** 2).sum(dim=2)   # (N,M) Symbolic matrix of squared distances.
K_ij = (- D_ij).exp()                  # (N,M) Symbolic Gaussian kernel matrix.

# Come back to genuine torch Tensors with reductions on dimensions 0 and 1:
nn = D_ij.argKmin(K=10, dim=1)  # K-NN search: (N,10) array of integer indices.
a  = K_ij @ b  # Kernel matrix-vector product: (N,M) * (M,E) = (N,E)
[g_x] = autograd.grad((a ** 2).sum(), [x])  # Seamless backpropagation.
```



(a) Simple reduction scheme on the CPU.

(b) Tiled reduction scheme on the GPU.

Figure 2: We rely on fast parallel schemes to compute reductions of symbolic matrices, as in Eq. (1). (a) On the CPU, each thread i computes a value a_i by looping over the reduction index j and consuming the values of F **on-the-fly**. (b) On the GPU, we cut (a) in K-by-K tiles (where K is the CUDA block size) to leverage the low latency of the shared memory buffer and block-wise memory accesses. This ensures an optimal management of the y_j's: we refer to our online documentation (`www.kernel-operations.io`) and to the N-body simulation chapter of [82] for details.

4 Implementation

Creating a symbolic tensor. The entry point to our library is a `LazyTensor` wrapper that turns dense arrays into symbolic matrices (lines 9-10). We use standard operations to build up arbitrary formulas F: lines 11-12 update a symbolic representation of the matrices `D_ij` and `K_ij`. Our math engine is **lazy**, and defers the evaluation of formulae until reduction time. It is also **versatile**: we support batch dimensions, operator broadcasting and a wide range of elementary operations. Formulae can use an arbitrary number of variables $x_i^1, x_i^2, \dots$ and $y_j^1, y_j^2, \dots$ with varied dimensions.

Parallel reduction schemes. Numerical computations take place at the reduction time (lines 15-16), when an associative operator $\square$ is called to perform an interaction step (1). To this end, we use a tiled map-reduce scheme that is detailed in Figure 2. Our C++ engine optimizes the use of registers to avoid costly memory transfers: the algorithm has $\mathcal{O}(\mathrm{NM})$ time complexity but does not allocate any buffer in the global device memory. For the sake of performance, we compile a specific binary for every new formula-reduction pair $(F, \square)$. These are stored in a cache directory for later use: the compilation is only done once. Unlike most acceleration frameworks, we do not expect the sizes of the data arrays $(x_i) \in \mathbb{R}^{\mathrm{N} \times \mathrm{D}}$ and $(y_j) \in \mathbb{R}^{\mathrm{M} \times \mathrm{D}}$ to be fully known at compile time: our binaries only rely on the feature dimensions D. This allows us to work with datasets of varying sizes, without having to re-compile binaries or sub-sample point clouds to an arbitrary point count. Notably, this allows our engine to process raw shape data on-the-fly.

Pruning negligible interactions. Our library provides support for alternative reduction strategies that improve numerical accuracy [67] or increase GPU usage when $\mathrm{N} < \mathrm{M}$. Going further, our symbolic tensors support the specification of block-wise sparsity masks as optional attributes. Block reduction tiles are encoded using a collection of $(i_{\text{start}}, i_{\text{end}}, j_{\text{start}}, j_{\text{end}})$ tuples of indices that allow our engine to focus on a subset of the full collection of interaction pairs $[\![1, \mathrm{N}]\!] \times [\![1, \mathrm{M}]\!]$. They can be used to prune out negligible terms from symbolic reductions, without giving up on the contiguous memory accesses that make or break the performance of CUDA kernels.

Among high-level computing frameworks, this feature is unique to our library. Block-diagonal sparsity masks allow us to provide seamless support for batch processing, even in heterogeneous situations that are of interest for point cloud and mesh processing. Assuming that the data arrays have been pre-sorted in contiguous clusters, block-sparsity masks also enable the GPU implementation of fast multiscale methods such as the Barnes-Hut algorithm [7].

Strengths and limitations. At its heart, KeOps leverages the low Kolmogorov complexity of symbolic arrays: it can be used when the computational bottleneck of a method is an interaction step of the form (1). As we show next in the benchmarks of Section 5, KeOps is likely to offer gains on runtime and memory usage when the numbers of samples N and M range from 10^3 to 10^7.

The main limitation of KeOps stems from the overflow of CUDA registers in the reduction of Figure 2: these result in decreased performances on large feature vectors with $\mathrm{D} > 100$. The problem is known as *register spilling*, with some documented work-arounds [25, 82]. Our priority for future developments is to improve performance for problems with $\mathrm{D} \sim 1\mathrm{k}$.

Another drawback is that we do not pre-ship binaries but instead rely on C++/CUDA compilers to run our kernels. To mitigate deployment issues and ease maintenance in the long run, we implement the core of the library in C++ and keep dependencies to a minimum [61]. In practice, compilation times range from 10 to 25 seconds when $\mathrm{D} \leqslant 1{,}000$ and can be prohibitive for $\mathrm{D} \geqslant 2{,}000$. Our library runs out-of-the-box on Linux or Mac configurations that provide a CUDA environment, e.g. fresh Google Colab sessions.

5 Experiments and Applications

Configuration. We now showcase our toolbox on a wide range of machine learning problems. All benchmarks were performed on a workstation equipped with 8 Intel Xeon Gold 6142 CPU @ 2.60GHz cores (16 threads), 128Gb of RAM and a Nvidia RTX 2080 Ti GPU with 11Gb of device memory. When relevant, we include comparisons with PyTorch JIT and JAX/XLA: just like our library, these two frameworks offer a transparent user interface and high performance on GPU. At the Python level, the implementations tested are rigorously equivalent to each other: we implement the same computations and only have to account for minor syntactic divergences. Whenever possible, we work with batches of samples and shapes to improve GPU usage.

Notations, datasets. In tables, "mem" stands for an out-of-memory error, "∞" for a time that was too high to be recorded and "—" denotes a lack of available implementation. L^1 and L^2 denote the Manhattan and Euclidean metrics, respectively, $\langle\,,\rangle$ denotes the cosine similarity and $\mathbb{H}^{\mathrm{D}}$ is the Poincaré metric on a hyperbolic space of dimension D [18, 21]. We perform numerical experiments with random normal samples and freely available datasets: digits from Scikit-Learn [87], Stanford dragon [26], ShapeNet [19]. MNIST [73], SIFT [62], GloVe-25 and GloVe-100 [88] were taken from the ANN-benchmarks repository [4], while HyperE-10 and HyperE-50 are hyperbolic embeddings processed from WordNet datasets [95].

5.1 Kernel Methods and Clustering

Kernel methods. Accelerating kernel methods is one of the core strengths of KeOps [20]. The code snippet of Section 3 shows how to perform a Gaussian convolution on point clouds and can be adapted to any other standard kernel. As detailed in the Supplementary Materials, symbolic `LazyTensors` can be transparently interfaced with the standard iterative solvers of the `scipy.sparse.linalg` library [66, 104]. This allows us to solve large-scale Gaussian regression problems ($\mathrm{N} > 100\mathrm{k}$-$1\mathrm{M}$) in seconds on a single GPU [57, 78]. Going forward, KeOps provides solid numerical foundations for recent methods in the field that rely on kernel matrices [45, 80].

Clustering. As detailed in the Supplementary Materials, KeOps can be used to implement K-means clustering with several metrics. Going further, we use our symbolic `LazyTensors` to implement the standard Expectation Maximisation (EM) algorithm on a Gaussian Mixture Model (GMM) [56]: we estimate the weight, mean and covariance matrix of each of the K components. To this end, the EM algorithm strives to maximize the likelihood that is computed from a data sample of N points in $\mathbb{R}^{\mathrm{D}}$. The most costly computations involve sum reductions of a K-by-N `LazyTensor` that encodes interactions between the K means of the clusters and the N sample points. Our results are summarized in Table 1. We show that KeOps performs well when computations involve variables of modest size: it can scale to large datasets in seconds, without memory overflows. This is most apparent when covariance matrices are diagonal and encoded as vectors of size D. In the second half of the table, we handle full D-by-D covariance matrices and notice a slow-down when $\mathrm{D} \geqslant 10$.

5.2 Nearest neighbor search with any metric

Syntax for a K-nearest neighbors query. Our library enables the efficient implementation of bruteforce KNN finders for any metric that is known in closed form. Users can apply the `argKmin(K=...)` reduction to extract the indices for the K-smallest values along the lines of any symbolic tensor:

```
# Turn (N,D) and (M,D) dense arrays into symbolic tensors:
x_i, y_j = LazyTensor(x.view(N,1,D)), LazyTensor(y.view(1,M,D))  # (N,1,D), (1,M,D)

# Compute the (N,M) symbolic matrix of distances:
E_ij = ((x_i - y_j) ** 2).sum(dim=2)     # Squared Euclidean metric
M_ij =  (x_i - y_j).abs().sum(dim=2)     # Manhattan distance
C_ij = 1 - (x_i | y_j)                   # Cosine similarity
H_ij = E_ij / (x_i[...,0] * y_j[...,0])  # Hyperbolic metric on the half-space

# Perform a K-NN search - in this case, for the hyperbolic metric:
indices = H_ij.argKmin(K=10, dim=1)  # Dense (N,K) array of integer indices
```

In the example above, the hyperbolic metric is defined on the half-space model of dimension D, with a positive first coordinate $x[0] > 0$ for feature vectors $x \in \mathbb{R}^{\mathrm{D}}$ [18]. The mapping $x \mapsto \operatorname{arcosh}(1+x/2)$ is increasing, which allows us to work with the pseudo-distance $H(x_i, y_j) = \|x_i - y_j\|^2 / (x_i[0] y_j[0])$: similar acceleration tricks can be applied to e.g. the Poincaré ball metric.

Exact and approximate nearest neighbor search. The complexity of a KNN query depends on the number of points N and features D that are available for a given dataset: in the literature, two types of strategies have been designed to handle different scenarios. On the one hand, **exact** bruteforce schemes compute all pairwise distances between query and data points. These methods stream well on GPUs, have little to no parameters to tune and require no pre-processing of the data. On the other hand, **approximate** schemes leverage the structure of the dataset to prune out useless computations whenever possible. These methods are usually implemented on CPUs and tend to rely on hierarchical decompositions of the dataset that must be pre-computed ahead of the first KNN query. As discussed in Sections 2 and 3, approximate methods are tailored for large-scale retrieval whereas bruteforce schemes are generally more suited to geometric scenarios.

Benchmarks. To illustrate this behaviour, we compare our bruteforce reduction scheme to a selection of common baselines: bruteforce PyTorch and JAX implementations on the GPU; the popular FAISS library [65], with an approximate HNSW algorithm on the CPU [77] and two dedicated CUDA implementation on the GPU: an exact bruteforce scheme and the approximate IVF-Flat method. To showcase the variety of settings in which KNN queries are used throughout the literature, we run these methods on a collection of *random* normal samples and *structured* datasets. We use $\mathrm{N} = 10^4$ to 10^7 points with $\mathrm{D} = 3$ to 784 features: these lower and upper bounds let us represent the range of problems that are encountered in geometric data analysis, from shape processing to the manipulation of word embeddings. We stress that for larger datasets, approximate methods would clearly outperform our bruteforce approach.

Results are displayed in Table 2. A first observation is that on these medium-sized problems, our bruteforce GPU implementation is faster than approximate CPU methods to build a full KNN graph: even when FAISS-HNSW outperforms KeOps in terms of queries per second, this comes at the cost of a significant pre-processing time. A second remark is that our **generic** reduction engine for bruteforce computations is on par with the hand-crafted CUDA routines of the FAISS library: KeOps is less efficient than the FAISS-Bruteforce routine when $\mathrm{D} \geqslant 50$ but is up to $\times 3$ faster in smaller dimensions. Crucially, we also provide the only competitive routines for non-Euclidean geometries, such as the increasingly popular hyperbolic metrics [83].

Conclusion. Overall, the performances of our C++ engine hold up to scrutiny: KeOps provides respectable runtimes on small and medium-sized problems (up to $\mathrm{N} = 10^6$ and $\mathrm{D} = 100$) with the added benefit of **flexibility**. These results attest to the effectiveness of our optimization techniques and bode well for the other computations that are supported by the KeOps engine. Going forward, we note that our library could probably be used as an efficient GPU backend for approximate KNN methods [31, 54, 118] and intend to provide the relevant tools for researchers in the field.

Table 1: Fitting a Gaussian Mixture Model: we perform 10 iterations of the standard EM algorithm with N points and K components in dimension D.

Covariances	N	K	D	Sklearn	PyTorch	**Ours**
Diagonal	50k	100	5	2.9 s	**19.0 ms**	33 ms
Diagonal	500k	1k	5	286 s	0.94 s	**0.22 s**
Diagonal	5M	10k	5	∞	mem	**11.38 s**
Diagonal	500k	1k	50	∞	mem	**2.96 s**
Diagonal	5M	10k	50	∞	mem	**245 s**
Full	50k	100	5	6.9 s	0.23 s	**0.04 s**
Full	500k	1k	5	830 s	mem	**0.362s**
Full	50k	100	20	16.0 s	mem	**0.84 s**
Full	500k	1k	20	∞	mem	**63 s**

Table 2: KNN search: average queries per second with a dataset of N points in dimension D. We work with batches of 10k queries at a time and $K = 10$ neighbors. The first three columns correspond to schemes that are provided by the FAISS library; the approximate methods HNSW and IVF-Flat are tuned to provide a minimum recall of 90% and optimize runtimes (we use the parameters of the ANN-benchmarks website as a first reference); when relevant, the pre-processing time is reported in parenthesis. We stress that choosing optimal parameters for the HNSW and IVF-Flat routines is a fairly complex problem: as non-specialists, we cannot guarantee that our experiments reflect the best level of performance that can be reached by these impressive methods. This is especially true for the IVF-PQ routine that is provided by FAISS and combines the IVF algorithm with a quantization method: it certainly fares even better than IVF-Flat on large-scale problems, but we found it to be very complex to use and opted to not include our unreliable benchmarks in this Table. High number of queries and low pre-processing is better: we highlight the column with the fastest time to process N queries and thus build a KNN graph for the dataset. (*) performed with smaller batch sizes when necessary to avoid memory overflows

Dataset, metric	N	D	HNSW (CPU)	Bruteforce	IVF-Flat	PyTorch*	JAX*	**Ours**
Random, L^2	10k	3	1.5e6 (0.15s)	3.3e6	1.9e6	8.8e5	1.6e5	**6.8e6**
Random, L^2	1M	3	1.3e6 (25s)	5.0e4	**1.6e6**	6.8e3	5.5e2	1.8e5
Random, L^2	1M	10	3.1e5 (53s)	4.6e4	**2.9e5**	4.9e3	4.9e2	1.1e5
Random, L^2	1M	100	1.5e3 (540s)	**3.1e4**	mem	9.0e2	4.0e2	1.4e4
Random, L^2	10M	100	(∞)	**3.2e3**	mem	∞	∞	1.4e3
Random, L^1	1M	10	—	—	—	1.7e3	4.6e2	**1.1e5**
Random, L^1	1M	100	—	—	—	2.1e2	3.0e2	**1.5e4**
MNIST, L^2	60k	784	5.4e4 (14s)	1.5e5	**2.2e5**	4.7e4	3.4e3	2.5e4
MNIST, L^1	60k	784	—	—	—	5.7e2	2.0e3	**2.5e4**
GloVe-25, $\langle , \rangle$	1.2M	25	7.7e4 (130s)	4.2e4	**1.6e5**	2.7e3	4.1e2	4.8e4
GloVe-100, $\langle , \rangle$	1.2M	100	8.6e3 (480s)	2.6e4	**3.9e4**	6.8e2	3.3e2	1.3e4
Random, $\langle , \rangle$	1M	10	1.8e5 (77s)	4.6e4	**2.7e5**	5.2e3	5.3e2	1.5e5
Random, $\mathbb{H}^D$	1M	10	—	—	—	3.2e3	5.1e2	**7.1e4**

Table 3: Accelerating geometric deep learning architectures: shown is average training / inference time per shape on the ShapeNet dataset, with clouds of $N = 2,048$ points in dimension $D = 3$.

	PointCNN $\rightarrow$ **Ours**	DGCNN $\rightarrow$ **Ours**
training	254 ms $\rightarrow$ **128 ms**	170 ms $\rightarrow$ **80 ms**
inference	172 ms $\rightarrow$ **43 ms**	109 ms $\rightarrow$ **20 ms**

5.3 Geometric deep learning and geometric primitives

Geometric deep learning. Fast low-dimensional KNN search has important applications in the field of geometric deep learning [16], a popular class of algorithms now ubiquitous in 3D computer vision and graphics [110]. As a first illustration, we use KeOps to speed-up two popular networks for 3D point cloud segmentation: Point CNNs [75] and Dynamic Graph CNNs (DGCNN) [110]. We follow closely the original architectures for part segmentation on the ShapeNet dataset [19] and compare two different implementations: a reference PyTorch_Geometric code [36] and a hybrid PyTorch_Geometric+KeOps implementation, where KNN graphs are built as in Section 5.2. Results are summarized in Table 3: training denotes a full "forward + backward" pass through the network whereas inference is forward only, with a faster batch normalisation [59]. Note that the Deep Graph Library [109] relies on a bruteforce PyTorch implementation for KNN search and behaves like PyTorch_Geometric on these problems. In practice, the switch to KeOps provides $\times 2$ and $\times 5$ speedups for training and inference times, respectively: the construction of KNN graphs becomes a negligible overhead, while the majority of the network runtime is spent on MLP and batch normalization layers.

Geometric descriptors at all scales. Table 4 shows the results of using KeOps to compute geometric features on generalized point clouds. In the interaction step (1), these correspond to the case where the reduction $\Box$ is a (possibly weighted) average on a neighborhood of x_i and the formula F is a function of the difference $x_i - y_j \in \mathbb{R}^{\mathrm{D}}$: the identity $x \mapsto x$, an outer product $x \mapsto xx^\top$ or a multi-layer perceptron with H hidden neurons and O output features. Neighborhoods are either defined through a 40-nearest neighbors query or by weighting each pair of points with a window of radius σ such as:

$$k(x_i, y_j) \;=\; \exp(-\|x_i - y_j\|^2/2\sigma^2). \tag{2}$$

Our library is well suited for these computations and consistently outperforms the PyTorch and JAX baselines by up to two orders of magnitude. We remark that on clouds of $\mathrm{N} = 2{,}048$ points, our bruteforce scheme (set radius) is up to an order of magnitude faster than the best sparse methods, which rely on scattered memory accesses to build neighborhoods as $(\mathrm{N}, \mathrm{K}, \mathrm{D})$ arrays. A similar behavior is observed for chamfer and Energy distance computations [14, 99]. These results highlight the dichotomy between *contiguous* ("bruteforce") and *scattered* ("sparse") memory accesses, which are optimized separately by GPUs and are best suited to different types of computations. We stress that KeOps supports the specification of block-wise sparsity schemes, which let users implement tree-based pruning strategies to best leverage the structure of their problems.

Conclusion. Overall, our library enables the quick development of differentiable layers for geometric deep learning which are an order of magnitude faster than Python baselines. It factors out C++ boilerplate code for point cloud processing and lets researchers focus on their *models*. We thus believe that KeOps will be of utmost interest to the developers of shape analysis frameworks [12, 36, 60, 92, 103] and CUDA kernels for point cloud convolution [35, 102, 119]. Future developments may also be relevant to natural language processing: transformer architectures and attention layers fit a similar design pattern, albeit in higher-dimensional spaces [100, 106, 113].

Table 4: Accelerating geometric primitives: average time per shape on the ShapeNet dataset, with clouds of $\mathrm{N} = 2,048$ points in dimension $\mathrm{D} = 3$.

	PyTorch	JAX		**Ours**	
Primitive / Neighborhood	40-NN	40-NN	Set radius	40-NN	Set radius
Local mean vectors	686 µs	1,052 µs	469 µs	121 µs	**12 µs**
Local covariance matrices	700 µs	1,093 µs	1,259 µs	138 µs	**23 µs**
MLP features ($\mathrm{H} = \mathrm{O} =$ `8`)	737 µs	1,180 µs	4,089 µs	192 µs	**75 µs**
MLP features ($\mathrm{H} = \mathrm{O} =$ `16`)	775 µs	1,253 µs	7,043 µs	**240 µs**	649 µs
Chamfer loss	374 µs	130 µs		**21 µs**	
Energy distance	486 µs	378 µs		**31 µs**	

5.4 Easing the development of complex geometric programs

Dimension reduction and geometric embeddings. The uniform manifold approximation and projection (UMAP) algorithm [79] is a standard method for dimension reduction. It can be used with various input metrics and relies on the analysis of a KNN graph of the input data. To showcase the benefits of KeOps for data analysis, we benchmark three different implementations of this method: the reference CPU-only library [79]; CuML, a fast GPU implementation that relies on FAISS for nearest neighbor queries [65, 91]; and a custom CuML+KeOps implementation. As typical examples of UMAP for visualization, we embed several datasets in the Euclidean plane: the digits, SIFT and MNIST datasets (endowed with the Euclidean and Manhattan metrics); the Glove-25 dataset (cosine similarity); and the HyperE-10 and -50 embeddings (hyperbolic metric).

Since the initial construction of the KNN graph is the most compute-intensive part of the UMAP algorithm, results are similar to those of Section 5.2 (a full table of results is provided in the Supplementary Materials). KeOps provides a $\times 2-5$ speed-up for the visualization of low-dimensional datasets, but is outperformed by CuML+FAISS when the input dimension D exceeds 50. Crucially, we provide the only GPU method that can handle non-Euclidean metrics: for hyperbolic embeddings, our implementation is $\times 200$ faster than the baseline on the HyperE-10 and -50 datasets [72, 83].

Optimal transport. Optimal transport (OT) is a generalization of sorting to spaces of dimension $\mathrm{D} > 1$ [81, 107]. It revolves around the resolution of a linear optimization problem whose value is usually known as the Wasserstein or Earth Mover's distance between two point clouds (x_i) and (y_j) [68, 93]. As detailed in [37, 89], modern OT solvers rely on iterated reductions performed on a cost matrix $C(x_i, y_j) = \|x_i - y_j\|$ or $\frac{1}{2}\|x_i - y_j\|^2$ and stand to benefit greatly from our library. To demonstrate this, we benchmark an exact linear solver [13, 42] against several variants and implementations of the Sinkhorn algorithm: a vanilla Sinkhorn loop [27, 29, 34, 41, 44, 94, 98, 112, 117]; an accelerated algorithm with simulated annealing [71]; and a fast multiscale scheme with kernel truncation [10, 96]. For the sake of numerical stability, we use symmetric updates [70] and perform all computations in the logarithmic domain [23]. To reflect the varied use cases of OT in the machine learning literature, we tackle two different problems: a high-precision matching between deformations of the Stanford dragon in dimension $\mathrm{D} = 3$, and a low-precision matching between deformations of the Glove-25 dataset in dimension $\mathrm{D} = 25$. These two regimes correspond to typical applications in computer vision [24, 38] and statistics [43, 46, 48], respectively.

A table of results is provided in the Supplementary Materials. In all settings, the switch from a PyTorch implementation to KeOps provides a $\times 5 - 20$ speed-up with no loss of precision. Most importantly, the low memory footprint of symbolic `LazyTensors` allows us to scale up to large problems ($\mathrm{N}, \mathrm{M} > 100\mathrm{k}$) without memory overflows. Our support for block-wise sparsity masks also lets us provide the first implementation of a multiscale solver for discrete OT on the GPU. In practice, this means that large OT problems with $\mathrm{N} = 100\mathrm{k}$ or $1\mathrm{M}$ samples can now be solved with high precision in fractions of a second instead of minutes or hours [37]. This implementation is straightforward to generalize to stochastic settings [2, 3, 47] and non-Euclidean cost functions [9, 28, 30]: we believe that it will open new ranges of applications to the OT community [37, 39].

6 Conclusion

KeOps combines a user-friendly interface with a high level of performance: we believe that it fills an important niche in the machine learning toolbox. We hope that our library will stimulate research in the field, with a simple but powerful structure that makes it a good tool for research off the beaten track. We look forward to feedback from users and keep the door open for contributors.

In months to come, our priority will be to improve performances on high-dimensional vectors with the newly released GPU Tensor cores and add support for quantization or approximation schemes such as the Nyström and FFM methods [5, 111]. We also work towards easing the deployment of pre-compiled binaries and target support of the ONNX standard [6]. These improvements should allow KeOps to become a standard toolbox in the field, both as an efficient backend for high-level software [12, 40, 45, 80] and as a versatile prototyping tool for theoretical research.

Broader Impact

Our work targets a wide range of machine learning applications, from kernel methods to geometric deep learning. In these fields, our library lowers the barrier of entry to state-of-the-art performances: fast nearest neighbors queries or point cloud convolutions can now be implemented by researchers who have no background in parallel computing. We hope that this will empower small research teams and organizations who don't have access to dedicated teams of software engineers. More specifically, the flexibility of our library is ideally suited to the formulation of data-driven models for shape analysis and point cloud processing. Progress in these sectors can have a major impact in computer vision and medical imaging – topics that carry both risks and promises for society as a whole. We hope that our library will promote the growth of a diverse ecosystem of academic and industrial actors, and look forward to seeing applications of our work to e.g. computational anatomy.

Acknowledgments and Disclosure of Funding

The three first authors are the project leaders: they contributed equally to the library and its documentation. Michael Bronstein is supported by the ERC Consolidator grant No. 724228 (LEMAN). KeOps was first motivated by applications to computational anatomy, in collaboration with Alain Trouvé. We also thank Ghislain Durif, who develops the R bindings of KeOps, as well as François-David Collin, who helps us to maintain our hardware and testing infrastructure.

References

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.

[2] J. Altschuler, F. Bach, A. Rudi, and J. Weed. Massively scalable Sinkhorn distances via the Nyström method. *arXiv preprint arXiv:1812.05189*, 2018.

[3] J. Altschuler, J. Niles-Weed, and P. Rigollet. Near-linear time approximation algorithms for optimal transport via Sinkhorn iteration. In *Advances in Neural Information Processing Systems*, pages 1964–1974, 2017.

[4] M. Aumüller, E. Bernhardsson, and A. Faithfull. ANN-Benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. *Information Systems*, 87:101374, 2020.

[5] M. Aussal and M. Bakry. The Fast and Free Memory method for the efficient computation of convolution kernels. *arXiv preprint arXiv:1909.05600*, 2019.

[6] J. Bai, F. Lu, K. Zhang, et al. ONNX: Open neural network exchange. `https://github.com/onnx/onnx`, 2019.

[7] J. Barnes and P. Hut. A hierarchical O(N log N) force-calculation algorithm. *Nature*, 324(6096):446, 1986.

[8] R. Beatson and L. Greengard. A short course on fast multipole methods. *Wavelets, multilevel methods and elliptic PDEs*, 1:1–37, 1997.

[9] J.-D. Benamou, W. L. Ijzerman, and G. Rukhaia. An entropic optimal transport numerical approach to the reflector problem. working paper or preprint, Apr. 2020.

[10] J.-D. Benamou and M. Martinet. Capacity constrained entropic optimal transport, Sinkhorn saturated domain out-summation and vanishing temperature. working paper or preprint, May 2020.

[11] N. Bhatia et al. Survey of nearest neighbor techniques. *arXiv preprint arXiv:1007.0085*, 2010.

[12] A. Bône, M. Louis, B. Martin, and S. Durrleman. Deformetrica 4: an open-source software for statistical shape analysis. In *International Workshop on Shape in Medical Imaging*, pages 3–13. Springer, 2018.

[13] N. Bonneel, M. Van De Panne, S. Paris, and W. Heidrich. Displacement interpolation using Lagrangian mass transport. In *Proceedings of the 2011 SIGGRAPH Asia Conference*, pages 1–12, 2011.

[14] G. Borgefors. Distance transformations in arbitrary dimensions. *Computer vision, graphics, and image processing*, 27(3):321–345, 1984.

[15] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, and S. Wanderman-Milne. JAX: composable transformations of Python+NumPy programs. *v0.1.55*, 2018.

[16] M. M. Bronstein, J. Bruna, Y. LeCun, A. Szlam, and P. Vandergheynst. Geometric deep learning: going beyond Euclidean data. *IEEE Signal Processing Magazine*, 34(4):18–42, 2017.

[17] L. Cambier and E. Darve. Fast low-rank kernel matrix factorization through skeletonized interpolation. *SIAM Journal on Scientific Computing*, 41(3):A1652–A1680, 2019.

[18] J. W. Cannon, W. J. Floyd, R. Kenyon, W. R. Parry, et al. Hyperbolic geometry. *Flavors of geometry*, 31:59–115, 1997.

[19] A. X. Chang, T. Funkhouser, L. Guibas, P. Hanrahan, Q. Huang, Z. Li, S. Savarese, M. Savva, S. Song, H. Su, et al. Shapenet: An information-rich 3d model repository. *arXiv preprint arXiv:1512.03012*, 2015.

[20] B. Charlier, J. Feydy, J. A. Glaunès, F.-D. Collin, and G. Durif. Kernel operations on the GPU, with autodiff, without memory overflows. *arXiv preprint arXiv:2004.11127*, 2020.

[21] É. Charpentier, E. Ghys, and A. Lesne. *The scientific legacy of Poincaré*, volume 36. American Mathematical Soc., 2010.

[22] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, et al. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, 2018.

[23] L. Chizat, G. Peyré, B. Schmitzer, and F.-X. Vialard. Scaling algorithms for unbalanced optimal transport problems. *Mathematics of Computation*, 87(314):2563–2609, 2018.

[24] H. Chui and A. Rangarajan. A new algorithm for non-rigid point matching. In *Computer Vision and Pattern Recognition, 2000. Proceedings. IEEE Conference on*, volume 2, pages 44–51. IEEE, 2000.

[25] S. Cook. *CUDA programming: a developer's guide to parallel computing with GPUs*. Newnes, 2012.

[26] B. Curless and M. Levoy. A volumetric method for building complex models from range images. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 303–312. ACM, 1996.

[27] M. Cuturi. Sinkhorn distances: Lightspeed computation of optimal transport. In *Advances in Neural Information Processing Systems*, pages 2292–2300, 2013.

[28] J. Delon, J. Salomon, and A. Sobolevski. Local matching indicators for transport problems with concave costs. *SIAM Journal on Discrete Mathematics*, 26(2):801–827, 2012.

[29] W. E. Deming and F. F. Stephan. On a least squares adjustment of a sampled frequency table when the expected marginal totals are known. *The Annals of Mathematical Statistics*, 11(4):427–444, 1940.

[30] S. Di Marino, A. Gerolin, and L. Nenna. Optimal transportation theory with repulsive costs. *Topological optimization and optimal transport*, 17:204–256, 2017.

[31] W. Dong, C. Moses, and K. Li. Efficient k-nearest neighbor graph construction for generic similarity measures. In *Proceedings of the 20th international conference on World wide web*, pages 577–586, 2011.

[32] A. Dutt and V. Rokhlin. Fast Fourier transforms for nonequispaced data. *SIAM Journal on Scientific computing*, 14(6):1368–1393, 1993.

[33] D. Eddelbuettel and R. François. Rcpp: Seamless R and C++ integration. *Journal of Statistical Software*, 40(8):1–18, 2011.

[34] S. Erlander. *Optimal spatial interaction and the gravity model*, volume 173. Springer Science & Business Media, 1980.

[35] M. Fey, J. Eric Lenssen, F. Weichert, and H. Müller. SplineCNN: Fast geometric deep learning with continuous b-spline kernels. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 869–877, 2018.

[36] M. Fey and J. E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[37] J. Feydy. *Geometric data analysis, beyond convolutions*. PhD thesis, Université Paris-Saclay, 2020.

[38] J. Feydy, B. Charlier, F.-X. Vialard, and G. Peyré. Optimal transport for diffeomorphic registration. In *International Conference on Medical Image Computing and Computer-Assisted Intervention*, pages 291–299. Springer, 2017.

[39] J. Feydy, P. Roussillon, A. Trouvé, and P. Gori. Fast and scalable optimal transport for brain tractograms. In *International Conference on Medical Image Computing and Computer-Assisted Intervention*, pages 636–644. Springer, 2019.

[40] J. Feydy, T. Séjourné, F.-X. Vialard, S.-i. Amari, A. Trouvé, and G. Peyré. Interpolating between optimal transport and MMD using Sinkhorn divergences. In *The 22nd International Conference on Artificial Intelligence and Statistics*, pages 2681–2690, 2019.

[41] S. E. Fienberg et al. An iterative procedure for estimation in contingency tables. *The Annals of Mathematical Statistics*, 41(3):907–917, 1970.

[42] R. Flamary and N. Courty. POT python optimal transport library, 2017.

[43] A. Forrow, J.-C. Hütter, M. Nitzan, P. Rigollet, G. Schiebinger, and J. Weed. Statistical optimal transport via factored couplings. In *The 22nd International Conference on Artificial Intelligence and Statistics*, pages 2454–2465, 2019.

[44] A. Galichon and B. Salanié. Matching with trade-offs: Revealed preferences over competing characteristics. *CEPR Discussion Paper No. DP7858*, 2010.

[45] J. Gardner, G. Pleiss, K. Q. Weinberger, D. Bindel, and A. G. Wilson. GPyTorch: Blackbox matrix-matrix Gaussian process inference with GPU acceleration. In *Advances in Neural Information Processing Systems*, pages 7576–7586, 2018.

[46] A. Genevay, L. Chizat, F. Bach, M. Cuturi, and G. Peyré. Sample complexity of Sinkhorn divergences. In *The 22nd International Conference on Artificial Intelligence and Statistics*, pages 1574–1583, 2019.

[47] A. Genevay, M. Cuturi, G. Peyré, and F. Bach. Stochastic optimization for large-scale optimal transport. In *Advances in Neural Information Processing Systems*, pages 3440–3448, 2016.

[48] A. Genevay, G. Peyré, and M. Cuturi. Learning generative models with Sinkhorn divergences. In *International Conference on Artificial Intelligence and Statistics*, pages 1608–1617, 2018.

[49] F. Gieseke, J. Heinermann, C. Oancea, and C. Igel. Buffer kd trees: processing massive nearest neighbor queries on GPUs. In *International Conference on Machine Learning*, pages 172–180, 2014.

[50] L. Greengard. *The rapid evaluation of potential fields in particle systems*. MIT press, 1988.

[51] L. Greengard and J.-Y. Lee. Accelerating the nonuniform fast Fourier transform. *SIAM review*, 46(3):443–454, 2004.

[52] L. Greengard and J. Strain. The fast Gauss transform. *SIAM Journal on Scientific and Statistical Computing*, 12(1):79–94, 1991.

[53] M. Greenspan and M. Yurick. Approximate kd tree search for efficient ICP. In *Fourth International Conference on 3-D Digital Imaging and Modeling, 2003. 3DIM 2003. Proceedings.*, pages 442–448. IEEE, 2003.

[54] R. Guo, P. Sun, E. Lindgren, Q. Geng, D. Simcha, F. Chern, and S. Kumar. Accelerating large-scale inference with anisotropic vector quantization. In *International Conference on Machine Learning*, 2020.

[55] W. Hackbusch. *Hierarchical matrices: algorithms and analysis*, volume 49. Springer, 2015.

[56] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer series in statistics. Springer, 2009.

[57] J. Hensman, N. Fusi, and N. D. Lawrence. Gaussian processes for big data. In *Uncertainty in Artificial Intelligence*, page 282. Citeseer, 2013.

[58] Intel AI. PlaidML: a framework for making deep learning work everywhere. *v0.7.0*, 2019.

[59] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[60] K. J., E. Smith, J.-F. Lafleche, C. Fuji Tsang, A. Rozantsev, W. Chen, T. Xiang, R. Lebaredian, and S. Fidler. Kaolin: A PyTorch library for accelerating 3d deep learning research. *arXiv:1911.05063*, 2019.

[61] W. Jakob, J. Rhinelander, and D. Moldovan. PyBind11 – seamless operability between C++11 and python, 2017. https://github.com/pybind/pybind11.

[62] H. Jegou, M. Douze, and C. Schmid. Product quantization for nearest neighbor search. *IEEE transactions on pattern analysis and machine intelligence*, 33(1):117–128, 2010.

[63] Z. Jia, S. Lin, C. R. Qi, and A. Aiken. Exploring hidden dimensions in parallelizing convolutional neural networks. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2018.

[64] Z. Jia, M. Zaharia, and A. Aiken. Beyond data and model parallelism for deep neural networks. In *Proceedings of the 2nd Conference on Machine Learning and Systems (MLSys)*, 2018.

[65] J. Johnson, M. Douze, and H. Jégou. Billion-scale similarity search with GPUs. *arXiv preprint arXiv:1702.08734*, 2017.

[66] E. Jones, T. Oliphant, P. Peterson, et al. SciPy: Open source scientific tools for Python, 2001. [Online; accessed 17th November 2019].

[67] W. Kahan. Further remarks on reducing truncation errors. *Communications of the ACM*, 8(1):40, 1965.

[68] L. Kantorovich. On the transfer of masses (in Russian). *Doklady Akademii Nauk*, 37(2):227–229, 1942.

[69] T. N. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[70] P. A. Knight, D. Ruiz, and B. Uçar. A symmetry preserving algorithm for matrix scaling. *SIAM journal on Matrix Analysis and Applications*, 35(3):931–955, 2014.

[71] J. J. Kosowsky and A. L. Yuille. The invisible hand algorithm: Solving the assignment problem with statistical physics. *Neural networks*, 7(3):477–490, 1994.

[72] J. Lamping, R. Rao, and P. Pirolli. A focus+context technique based on hyperbolic geometry for visualizing large hierarchies. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 401–408, 1995.

[73] Y. LeCun and C. Cortes. The MNIST database of handwritten digits. *http://yann. lecun. com/exdb/mnist/*, 1998.

[74] T.-M. Li, M. Gharbi, A. Adams, F. Durand, and J. Ragan-Kelley. Differentiable programming for image processing and deep learning in Halide. *ACM Transactions on Graphics (TOG)*, 37(4):1–13, 2018.

[75] Y. Li, R. Bu, M. Sun, W. Wu, X. Di, and B. Chen. PointCNN: Convolution on X-transformed points. In *Advances in neural information processing systems*, pages 820–830, 2018.

[76] Z. Liu, H. Tang, Y. Lin, and S. Han. Point-Voxel CNN for efficient 3d deep learning. In *Advances in Neural Information Processing Systems*, pages 963–973, 2019.

[77] Y. A. Malkov and D. A. Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE transactions on pattern analysis and machine intelligence*, 2018.

[78] G. Matheron. Principles of geostatistics. *Economic geology*, 58(8):1246–1266, 1963.

[79] L. McInnes, J. Healy, and J. Melville. UMAP: Uniform manifold approximation and projection for dimension reduction. *arXiv preprint arXiv:1802.03426*, 2018.

[80] G. Meanti, L. Carratino, L. Rosasco, and A. Rudi. Kernel methods through the roof: handling billions of points efficiently. *arXiv preprint arXiv:2006.10350*, 2020.

[81] G. Monge. Mémoire sur la théorie des déblais et des remblais. *Histoire de l'Académie Royale des Sciences*, pages 666–704, 1781.

[82] H. Nguyen. *Gpu gems 3*. Addison-Wesley Professional, 2007.

[83] M. Nickel and D. Kiela. Poincaré embeddings for learning hierarchical representations. In *Advances in neural information processing systems*, pages 6338–6347, 2017.

[84] R. Okuta, Y. Unno, D. Nishino, S. Hido, and C. Loomis. CuPy: A NumPy-compatible library for Nvidia GPU calculations. In *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Thirty-first Annual Conference on Neural Information Processing Systems (NIPS)*, 2017.

[85] S. M. Omohundro. *Five balltree construction algorithms*. International Computer Science Institute Berkeley, 1989.

[86] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in PyTorch. *NIPS 2017 Workshop Autodiff*, 2017.

[87] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[88] J. Pennington, R. Socher, and C. D. Manning. GloVe: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543, 2014.

[89] G. Peyré and M. Cuturi. Computational optimal transport. *arXiv:1610.06519*, 2017.

[90] A. Rahimi and B. Recht. Random features for large-scale kernel machines. In *Advances in neural information processing systems*, pages 1177–1184, 2008.

[91] RAPIDS Development Team. *RAPIDS: Collection of Libraries for End to End GPU Data Science*, 2018.

[92] N. Ravi, J. Reizenstein, D. Novotny, T. Gordon, W.-Y. Lo, J. Johnson, and G. Gkioxari. PyTorch3D. `https://github.com/facebookresearch/pytorch3d`, 2020.

[93] Y. Rubner, C. Tomasi, and L. J. Guibas. The earth mover's distance as a metric for image retrieval. *International Journal of Computer Vision*, 40(2):99–121, Nov. 2000.

[94] L. Ruschendorf et al. Convergence of the iterative proportional fitting procedure. *The Annals of Statistics*, 23(4):1160–1174, 1995.

[95] F. Sala, C. De Sa, A. Gu, and C. Re. Representation tradeoffs for hyperbolic embeddings. In J. Dy and A. Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 4460–4469, Stockholmsmässan, Stockholm Sweden, 10–15 Jul 2018. PMLR.

[96] B. Schmitzer. Stabilized sparse scaling algorithms for entropy regularized transport problems. *SIAM Journal on Scientific Computing*, 41(3):A1443–A1481, 2019.

[97] N. Shazeer, Y. Cheng, N. Parmar, D. Tran, A. Vaswani, P. Koanantakool, P. Hawkins, H. Lee, M. Hong, C. Young, R. Sepassi, and B. Hechtman. Mesh-TensorFlow: Deep learning for supercomputers. In *Neural Information Processing Systems*, 2018.

[98] R. Sinkhorn. A relationship between arbitrary positive matrices and doubly stochastic matrices. *Ann. Math. Statist.*, 35:876–879, 1964.

[99] G. J. Székely and M. L. Rizzo. Testing for equal distributions in high dimension. *InterStat*, 5(16.10), 2004.

[100] Y. Tay, M. Dehghani, D. Bahri, and D. Metzler. Efficient transformers: A survey. *arXiv preprint arXiv:2009.06732*, 2020.

[101] Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, May 2016.

[102] H. Thomas, C. R. Qi, J.-E. Deschaud, B. Marcotegui, F. Goulette, and L. J. Guibas. KPConv: Flexible and deformable convolution for point clouds. In *The IEEE International Conference on Computer Vision (ICCV)*, October 2019.

[103] J. Tierny, G. Favelier, J. A. Levine, C. Gueunet, and M. Michaux. The topology toolkit. *IEEE Transactions on Visualization and Computer Graphics*, 24(1):832–842, 2017.

[104] S. Van Der Walt, S. C. Colbert, and G. Varoquaux. The NumPy array: a structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):22, 2011.

[105] N. Vasilache, O. Zinenko, T. Theodoridis, P. Goyal, Z. DeVito, W. S. Moses, S. Verdoolaege, A. Adams, and A. Cohen. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv preprint arXiv:1802.04730*, 2018.

[106] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[107] C. Villani. *Optimal transport: old and new*, volume 338. Springer Science & Business Media, 2008.

[108] M. Wang, C.-c. Huang, and J. Li. Supporting very large models using automatic dataflow graph partitioning. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–17, 2019.

[109] M. Wang, L. Yu, D. Zheng, Q. Gan, Y. Gai, Z. Ye, M. Li, J. Zhou, Q. Huang, C. Ma, et al. Deep graph library: Towards efficient and scalable deep learning on graphs. *arXiv preprint arXiv:1909.01315*, 2019.

[110] Y. Wang, Y. Sun, Z. Liu, S. E. Sarma, M. M. Bronstein, and J. M. Solomon. Dynamic graph CNN for learning on point clouds. *ACM Transactions on Graphics (TOG)*, 38(5):1–12, 2019.

[111] C. K. Williams and M. Seeger. Using the Nyström method to speed up kernel machines. In *Advances in neural information processing systems*, pages 682–688, 2001.

[112] A. G. Wilson. The use of entropy maximising models, in the theory of trip distribution, mode split and route split. *Journal of Transport Economics and Policy*, pages 108–126, 1969.

[113] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, J. Davison, S. Shleifer, P. von Platen, C. Ma, Y. Jernite, J. Plu, C. Xu, T. L. Scao, S. Gugger, M. Drame, Q. Lhoest, and A. M. Rush. HuggingFace's Transformers: State-of-the-art natural language processing. *ArXiv*, abs/1910.03771, 2019.

[114] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and S. Y. Philip. A comprehensive survey on graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 2020.

[115] C. Yang, R. Duraiswami, N. A. Gumerov, and L. Davis. Improved fast Gauss transform and efficient kernel density estimation. In *Proceedings of the Ninth IEEE International Conference on Computer Vision*, volume 2, page 464. IEEE Computer Society, 2003.

[116] T. Yang, Y.-F. Li, M. Mahdavi, R. Jin, and Z.-H. Zhou. Nyström method vs random Fourier features: A theoretical and empirical comparison. In *Advances in neural information processing systems*, pages 476–484, 2012.

[117] G. U. Yule. On the methods of measuring association between two attributes. *Journal of the Royal Statistical Society*, 75(6):579–652, 1912.

[118] W.-L. Zhao. Approximate k-NN graph construction: a generic online approach. *arXiv preprint arXiv:1804.03032*, 2018.

[119] Y. Zhou, C. Wu, Z. Li, C. Cao, Y. Ye, J. Saragih, H. Li, and Y. Sheikh. Fully convolutional mesh autoencoder using efficient spatially varying kernels. In *arxiv*, 2020.

Extended benchmarks and implementations

Content of the supplementary materials. Our library is freely available on the PyPi (`pip install pykeops`) and CRAN (`install.packages("rkeops")`) repositories. Its user interface and inner workings are fully documented on our website (`www.kernel-operations.io`), with source code available under the permissive MIT license (`www.github.com/getkeops/keops`).

The full codes for our benchmarks have been integrated to our documentation as tutorials. In the pages below, we present supporting material for the discussion of Section 5: we include all the relevant equations, code samples and tables of results. The hardware configuration and datasets are described at the start of Section 5.

A Kernel methods

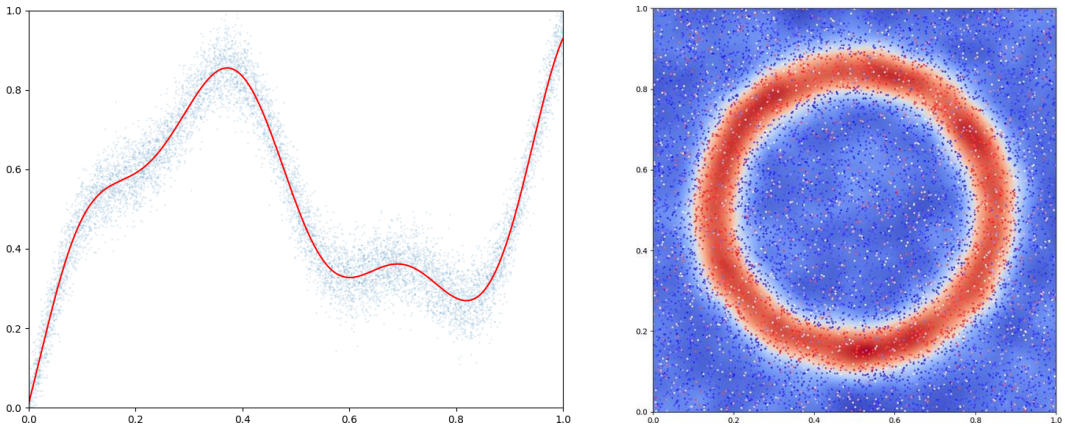
Kernels, Gaussian process regression. As discussed in Section 5.1, KeOps is ideally suited to the implementation of kernel methods: LazyTensors can be used to represent arbitrary kernel matrices with a low memory footprint and high performance. As an example, we show how to interface our library with the standard solvers of the `scipy.sparse.linalg` package [66] – a reference toolbox for e.g. the computation of Laplacian eigenvectors on graphs and meshes.

SciPy interface. If $x = (x_i) \in \mathbb{R}^{\mathrm{N}\times\mathrm{D}}$ is a cloud of N points in $\mathbb{R}^{\mathrm{D}}$ and if $b = (b_i) \in \mathbb{R}^{\mathrm{N}\times\mathrm{E}}$ is a signal of dimension E supported by the x_i's, the code below implements a fast conjugate gradient solver for the resolution of a linear system with respect to $a = (a_i) \in \mathbb{R}^{\mathrm{N}\times\mathrm{E}}$:

$$b \;=\; (\alpha\,\mathrm{Id} + K_{x,x})\,a \qquad\qquad \text{i.e.} \qquad\qquad a \;=\; (\alpha\,\mathrm{Id} + K_{x,x})^{-1}\,b\,, \tag{3}$$

where $K_{x,x} = (K_{x_i,x_j}) = (\exp(-\|x_i - x_j\|/\sigma)$ is a (N,N) symmetric positive definite matrix associated to an exponential kernel of radius $\sigma > 0$ and where $\alpha \geqslant 0$ is the strength of a L^2-Tikhonov regularization. This operation is a the heart of Gaussian process regression [57]: it is usually known as Kriging in geostatistics, kernel regression in data sciences or spline regression in imaging. We illustrate some typical use cases in Figure 3.

```python
import numpy as np  # NumPy arrays on the CPU
from scipy.sparse import diags  # Sparse diagonal matrices
from scipy.sparse.linalg import aslinearoperator, cg  # Conjugate gradient
from pykeops.numpy import LazyTensor  # Symbolic wrapper for NumPy arrays

# Toy problem in dimension D = 50:
N, D, E = 10 ** 6, 50, 1  # samples, features, signals
x = np.random.randn(N, D).astype('float32')  # float16, float32 and float64
b = np.random.randn(N, E).astype('float32')  # are all supported by KeOps.

sigma = .2  # radius of the exponential kernel
alpha = .5  # ridge/Tikhonov regularization

# Build the symbolic (N, N) kernel matrix:
x_i = LazyTensor(x.reshape(N, 1, D))      # (N, 1, D) data samples
x_j = LazyTensor(x.reshape(1, N, D))      # (1, N, D) data samples

D_ij = ((x_i - x_j) ** 2).sum(2).sqrt()  # (N, N) distances
K_ij = (- D_ij / sigma).exp()            # (N, N) exponential kernel matrix

# Turn the LazyTensor into a SciPy object, add Tikhonov regularization:
K = aslinearoperator(K_ij)  # Transparent duck typing from KeOps to SciPy
Ka = K + aslinearoperator(diags(alpha * np.ones(N)))  # Standard SciPy syntax
Ka.dtype = np.dtype('float32')  # Use the correct precision

# Interface KeOps with all the standard solvers of scipy.sparse.linalg:
a = cg(Ka, b)  # Conjugate gradient: eigenproblem solvers, etc. are also supported.
```



(a) Kernel regression in 1D. (b) Kriging in 2D.

Figure 3: **Kriging**, also known as **kernel**, **spline** or **Gaussian process regression** is a fundamental tool in data sciences that relies on the resolution of large kernel linear systems (3). Out-of-the-box, our symbolic tensors can be interfaced with standard libraries for linear algebra such as SciPy [66]. This lets users scale up standard iterative solvers to datasets of $\mathrm{N} = 10\mathrm{k}$ to 10M samples in seconds or minutes. (a) As a first example, we work with a Gaussian kernel $k(x, y) = \exp(-|x-y|^2/2\sigma^2)$ of deviation $\sigma = \texttt{0.1}$ on the real line. We use $\mathrm{N} = 10\mathrm{k}$ samples in dimension $\mathrm{D} = 1$, with a scalar-valued signal ($\mathrm{E} = 1$): we represent the data with blue points (x_i, b_i) in the graph. The red curve corresponds to the kernel regression $x \mapsto \sum_{i=1}^{\mathrm{N}} k(x, x_i)\, a_i$, a smooth curve that does not overfit to noise thanks to the Tikhonov regularization. (b) The second example is representative of applications to geostatistics: we work with $\mathrm{N} = 10\mathrm{k}$ samples in dimension $\mathrm{D} = 2$. Thanks to an exponential kernel of deviation $\sigma = \texttt{0.1}$ and a small amount of Tikhonov regularization, we retrieve a continuous interpolation of a noisy scalar signal ($\mathrm{E} = 1$) on the whole domain: a plausible terrain model, displayed as an image in the background while every point corresponds to a sample x_i with color b_i. The flexible structure of our library empowers researchers, who can use LazyTensors to perform fast kernel regressions on arbitrary domains, such as the sphere or the Poincaré plane.

Performance. Providing rigorous and precise benchmarks for iterative linear solvers is an arduous task: a wide range of methods have been proposed to accelerate the resolution of systems that involve e.g. smooth kernel functions. Depending on their specific needs, users often have to pick a method and parameter values that reach a satisfying trade-off between speed and accuracy.

Nevertheless, according to our experiments with default precision settings, $\mathrm{N} = 1\mathrm{k}$ to 10M points in dimension $\mathrm{D} = 1$ to 100 and varied kernel functions (Gaussian, exponential, Cauchy, etc.), we observe that SciPy+KeOps implementations are consistently $\times$10-50 times faster than their standard PyTorch counterparts (`torch.solve(...)`) and $\times$1,000-5,000 times faster than a vanilla resolution with SciPy on the CPU. These speed-ups come from our efficient use of CUDA registers and could be applied to accelerate most large-scale solvers in the field [57]: we believe that our library will be of interest to many researchers who work with Gaussian processes or kernel matrices.

B Clustering

B.1 K-Means: Lloyd's algorithm

Fast clustering with K-means. We now discuss the applications of symbolic tensors to clustering. We first consider the problem of partitioning a dataset $(x_i) \in \mathbb{R}^{\mathrm{N}\times\mathrm{D}}$ of N points in $\mathbb{R}^{\mathrm{D}}$ in K distinct clusters. The K-means method or (discrete) "Lloyd's algorithm" is probably the most common approach to the question: we work with a collection $(c_k) \in \mathbb{R}^{\mathrm{K}\times\mathrm{D}}$ of K cluster "centroids" in $\mathbb{R}^{\mathrm{D}}$, class labels $(l_i) \in [\![1, \mathrm{K}]\!]^{\mathrm{N}}$ for every point x_i and update both parameters alternatively to minimize

the within-cluster sum of squared distances:

$$\text{SSD}(c_k, l_i) \;=\; \sum_{k=1}^{\text{K}} \sum_{l_i=k} \|x_i - c_k\|^2. \tag{4}$$

At every iteration of the K-means loop, we first assign each point x_i to the closest centroid c_k (i.e. minimize SSD with respect to the l_i's) before updating c_k as the mean of all points x_i such that $l_i = k$. Using KeOps for the assignment step, we can write a fast and simple implementation of this algorithm as follows:

```
def KMeans(x, K, niter, verbose=True):
    """
     points  ->  labels, centroids
     (N, D)  ->   (N,),   (K, D)
    """
    N, D = x.shape # Number of samples, dimension of the ambient space

    c = x[:K, :].clone() # Simple initialization for the centroids
    # Encoding as symbolic tensors:
    x_i = LazyTensor(x.view(N, 1, D)) # (N, 1, D) symbolic tensor
    c_j = LazyTensor(c.view(1, K, D)) # (1, K, D) symbolic tensor

    for _ in range(niter):  # K-means loop
        # Assignment step:
        D_ij = ((x_i - c_j) ** 2).sum(-1) # (N, K) squared distances
        l = D_ij.argmin(dim=1).long().view(-1) # Points -> Nearest cluster

        # Compute the cluster mean values:
        weights = torch.bincount(l).type_as(x)
        for d in range(D):  # In-place update of the centroids:
            c[:, d] = torch.bincount(l, weights=x[:, d]) / weights

    return l, c  # Labels, centroids
```

Note that we use a weighted `torch.bincount` method for the update step, which avoids looping over the class index in $[\![1, K]\!]$. In practice, this second step relies on scattered memory accesses and is the bottleneck of the K-means loop for small-scale problems. On our system, with N =1M, K =1k and D = 100, this implementation performs 10 iterations in less than a second (0.81s on average).

Manhattan distance. Our library is versatile, and lets users prototype arbitrary generalizations of standard algorithms. For instance, we can easily implement an L^1-Manhattan variant of Lloyd's algorithm to minimize the robust cost function:

$$\text{SD}_{L^1}(c_k, l_i) \;=\; \sum_{k=1}^{\text{K}} \sum_{l_i=k} \|x_i - c_k\|_1 \;=\; \sum_{k=1}^{\text{K}} \sum_{l_i=k} \sum_{d=1}^{\text{D}} |\, x_i[d] - c_k[d] \,|\,. \tag{5}$$

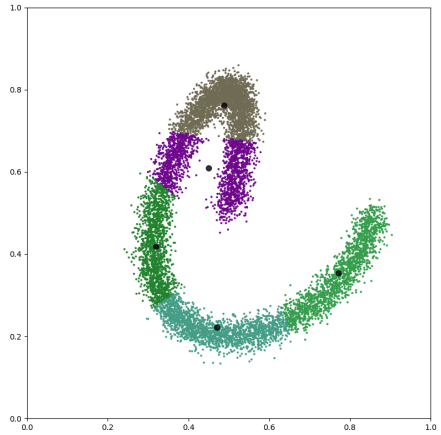
In the assignment step, we replace the squared Euclidean norms by Manhattan distances, prior to the nearest neighbour search:

```
        D_ij = ((x_i - c_j).abs()).sum(-1) # (N, K) Manhattan distances
```

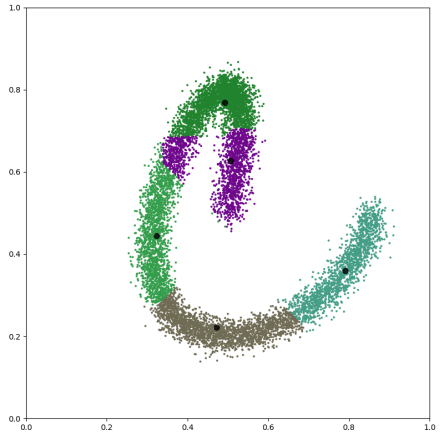
As for the update step, we replace means by medians to compute the new centroids c_k:

```
        # Update cluster centroids:
        for k in range(K):
            c[k,:] = torch.median(x[l==k,:], dim=0)[0]
```

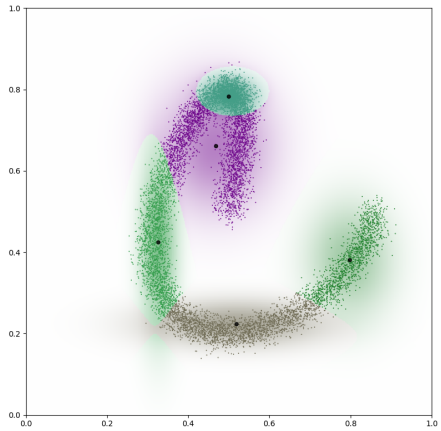
Note that in this case, there is no simple way of avoiding a loop over K for the update step with PyTorch. As a consequence, the performance of this implementation drops to an average of 1.85s for the same test dimensions – N =1M, K =1k, D =100 and 10 iterations.



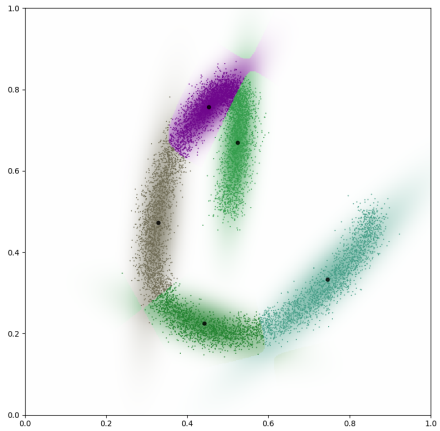
(a) Lloyd's algorithm.



(b) Lloyd's algorithm, L^1 variant.



(c) GMM with diagonal covariances.



(d) GMM with full covariances.

Figure 4: Clustering of a synthetic 2D dataset ($\mathrm{N} = 10\mathrm{k}$, $\mathrm{D} = 2$), into $\mathrm{K} = 5$ classes with four different methods. (a) The standard Lloyd's algorithm for the L^2-Euclidean metric. (b) A variant of Lloyd's algorithm for the L^1-Manhattan metric. (c) The EM algorithm on a Gaussian mixture model with diagonal covariances. (d) The EM algorithm on a Gaussian mixture model with full covariances. We display the points x_i in the unit square, colored according to the class labels l_i. For the Gaussian mixture models (c-d), we also display the model likelihood (6) in the background, with colors that reflect the dominant cluster at any given location. All the experiments were performed using KeOps, following the implementations of Section B.

B.2 Gaussian mixture models: the EM algorithm

Notations. We now detail the content of Table 1. We consider a dataset $(x_i) \in \mathbb{R}^{\mathrm{N}\times\mathrm{D}}$ of N points in $\mathbb{R}^{\mathrm{D}}$ and fit a Gaussian mixture model $\mathrm{GMM}(w_j, \mu_j, \Sigma_j; j \in [\![1, \mathrm{K}]\!])$ that is parameterized by a collection of K:

1. **weights** $w_j \geqslant 0$ that sum up to 1,
2. **mean values** $\mu_j \in \mathbb{R}^{\mathrm{D}}$,
3. **covariance matrices** $\Sigma_j \in \mathbb{R}^{\mathrm{D}\times\mathrm{D}}$.

The likelihood of the model at any point $x \in \mathbb{R}^{\mathrm{D}}$ is given by:

$$\text{likelihood}(x) \;=\; \sum_{j=1}^{\mathrm{K}} \frac{w_j}{(2\pi)^{\mathrm{D}/2}\sqrt{\det(\Sigma_j)}} \exp\Big(-\tfrac{1}{2}(x-\mu_j)^\top \Sigma_j^{-1}(x-\mu_j)\Big)\,. \tag{6}$$

EM iterations. Starting from a random initialization, we fit the model to the data using the standard Expectation-Maximization algorithm. Its iterations read as follows:

1. **E-step:** compute membership probabilities. For every point x_i and component (w_j, μ_j, Σ_j), we compute the likelihood ratio:

$$\pi_{i,j} = \frac{\text{likelihood}_{j\text{-th component}}(x_i)}{\text{likelihood}_{\text{full}}(x_i)} \tag{7}$$

$$= \frac{w_j \, \exp\big(-\frac{1}{2}(x_i-\mu_j)^\top \Sigma_j^{-1}(x_i-\mu_j)\big)/\sqrt{\det(\Sigma_j)}}{\sum_{k=1}^{\mathrm{K}} w_k \, \exp\big(-\frac{1}{2}(x_i-\mu_k)^\top \Sigma_k^{-1}(x_i-\mu_k)\big)/\sqrt{\det(\Sigma_k)}} \tag{8}$$

   $(\pi_{i,j}) \in \mathbb{R}^{\mathrm{N}\times\mathrm{K}}_{\geqslant 0}$ is encoded as a (N, K) array whose lines sum up to 1.

2. **M-step:** update the model parameters. We execute sequentially the following equations:

$$P_j \leftarrow \sum_{i=1}^{\mathrm{N}} \pi_{i,j} \,, \qquad w_j \leftarrow P_j/\mathrm{N} \,, \tag{9}$$

$$\mu_j \leftarrow \frac{1}{P_j}\sum_{i=1}^{\mathrm{N}} \pi_{i,j} x_i \,, \qquad \Sigma_j \leftarrow \frac{1}{P_j}\sum_{i=1}^{\mathrm{N}} \pi_{i,j}(x_i-\mu_j)(x_i-\mu_j)^\top \,. \tag{10}$$

   For the sake of numerical stability, we add a small value $\varepsilon = 10^{-7}$ to the class scores P_j when they are used as denominators. Alternatively, we could work with their logarithms and stabilized log-sum-exp reductions: these are fully supported by our library.

Inverting the K covariance matrices Σ_j to compute the precisions $\Sigma_j^{-1} \in \mathbb{R}^{\mathrm{D}\times\mathrm{D}}$ for every E-step can be costly. In Table 1, we also benchmark an alternative version of the algorithm where the covariances are assumed to be diagonal matrices and encoded as positive vectors $\sigma_j \in \mathbb{R}^{\mathrm{D}}$.

```python
# Input:  points is (N, D)
# Params: weights is (K,), means is (K, D), covariances is (K, D, D)

for _ in range(niter):
    # Expectation step: compute membership probabilities ------------------
    # Compute mixture weights:
    precisions = covariances.inverse()  # (K, D, D)
    w = weights * torch.sqrt(precisions.det())  # (K,)

    # Encoding as symbolic tensors:
    x_i = LazyTensor(points.view(N, 1, D))    # (N, 1, D)
    m_j = LazyTensor( means.view(1, K, D))    # (1, K, D)
    w_j = LazyTensor(w.view(1, K, 1))         # (1, K, 1)

    # Gaussian likelihoods:
    P_j = LazyTensor(precisions.reshape(1, K, D * D))  # (1, K, D*D)
    D_ij = ((x_i - m_j) * P_j.matvecmult(x_i - m_j)).sum(dim=2)  # (N, K)
    K_ij = (- D_ij / 2).exp() * w_j  # (N, K)

    # Bayes normalization constant:
    BN = K_ij.sum(dim=1)  # (N,)
    BN_i = LazyTensor(BN.view(N, 1, 1) + eps)  # (N, 1)

    # Compute the membership probabilities:
    P_ij = K_ij / BN_i    # (N, K)

    # Maximization step: update the mixture parameters -------------------
    P = P_ij.sum(dim=0)         # (K, 1)
    weights = P.view(-1) / N  # (K,)
    means = (P_ij * x_i).sum(dim=0) / (P + eps)  # (K, D)

    # New means to compute the adjusted covariances:
    m_j = LazyTensor(means.view(1, K, D))  # (1, K, D)

    # Covariance matrices
    covariances = (P_ij * (x_i-m_j).tensorprod(x_i-m_j)).sum(0).view(K,D,D)
    covariances = covariances / (P.view(K, 1, 1) + eps)  # (K, D, D)
```

C Dimensionality reduction

As discussed in Section 5.3, we benchmark: the original UMAP implementation on the CPU; the CuML+FAISS implementation on the GPU; a CUML+KeOps pipeline which relies on symbolic tensors to build the KNN graph of a dataset before relying on CuML to construct the low-dimensional embedding. This pipeline allows us to compute UMAP embeddings with arbitrary metrics on the input datasets: it is indicative of the versatility of KeOps, which can be interfaced with a wide range of standard libraries. Results are presented in Table 5, with examples of embeddings shown in Figure 5.

Note on the hyperbolic metric. The HyperE-10 and -50 datasets provide reference embeddings of real world data into hyperbolic spaces of dimensions 10 and 50. In practice, the datasets both rely on the Poincaré ball model and provide a scaling factor that should be used to recover the hyperbolic distance between any two vectors. If x_i and x_j are two samples in the dataset, encoded as vectors of norms $\|x_i\|, \|x_j\| < 1$ in $\mathbb{R}^{\mathrm{D}}$, the hyperbolic distance between them is given by:

$$\mathrm{d}(x_i, x_j) \;=\; \operatorname{arccosh}\left(1 + 2\,\frac{\|x_i - x_j\|^2}{(1-\|x_i\|^2)(1-\|x_j\|^2)}\right) \;/\; \text{ScalingFactor}\,, \tag{11}$$

where $\|\cdot\|$ denotes the standard Euclidean norm in $\mathbb{R}^{\mathrm{D}}$. In order to perform a KNN search efficiently, we remark that $x \mapsto \operatorname{arccosh}(1+2x)$ is an increasing mapping. Since the values of $(1-\|x_i\|^2)$ can be computed ahead of the KNN reduction, we can build our KNN graph with:

```
# x is a (N, D) array with double precision. We first compute the scaling factors:
u = 1. / (1. - (x ** 2).sum(dim=1))  # With double = float64 precision.
x, u = x.float(), u.float()  # We can use float32 precision after this step.

# And encode our variables as symbolic tensors:
x_i = LazyTensor(x.view(N, 1, D))
x_j = LazyTensor(x.view(1, N, D))
u_i = LazyTensor(u.view(N, 1, 1))
u_j = LazyTensor(u.view(1, N, 1))

# We can then perform the KNN search efficiently:
D_ij = ((x_i - x_j) ** 2).sum(dim=2) * u_i * u_j
distances, indices = D_ij.Kmin_argKmin(K, dim = 1)

# And compute the genuine hyperbolic distances to the K-nearest neighbors:
acosh = lambda x : torch.log( x + (x ** 2 - 1.) ** 0.5)
distances = arccosh(1. + 2 * distances) / scaling_factor
```

Table 5: Dimension reduction using the UMAP algorithm. We record the time to embed datasets in the Euclidean plane. When the input metric is Euclidean, the dataset is first pre-processed with a PCA as advised by the UMAP documentation: we keep 95% of the total variance.

Dataset	Metric	N	D	PCA preprocessing	Umap	CuML	CuML+**Ours**
Digits	L^2	1.8k	64	4 ms $\to D'=28$	5.8 s	170 ms	**32 ms**
MNIST	L^2	60k	784	68 ms $\to D'=153$	38 s	**450 ms**	670 ms
MNIST	L^1	60k	784	—-	43 s	—-	**2.3 s**
SIFT	L^2	1M	128	64 ms $\to D'=71$	1,380 s	**28 s**	53 s
GloVe-25	$\langle\,,\rangle$	1.2M	25	—-	1,660 s	31 s	**29 s**
HyperE-10	$\mathbb{H}^{\mathrm{D}}$	105k	10	—-	150 s	—-	**560 ms**
HyperE-50	$\mathbb{H}^{\mathrm{D}}$	105k	50	—-	200 s	—-	**900 ms**

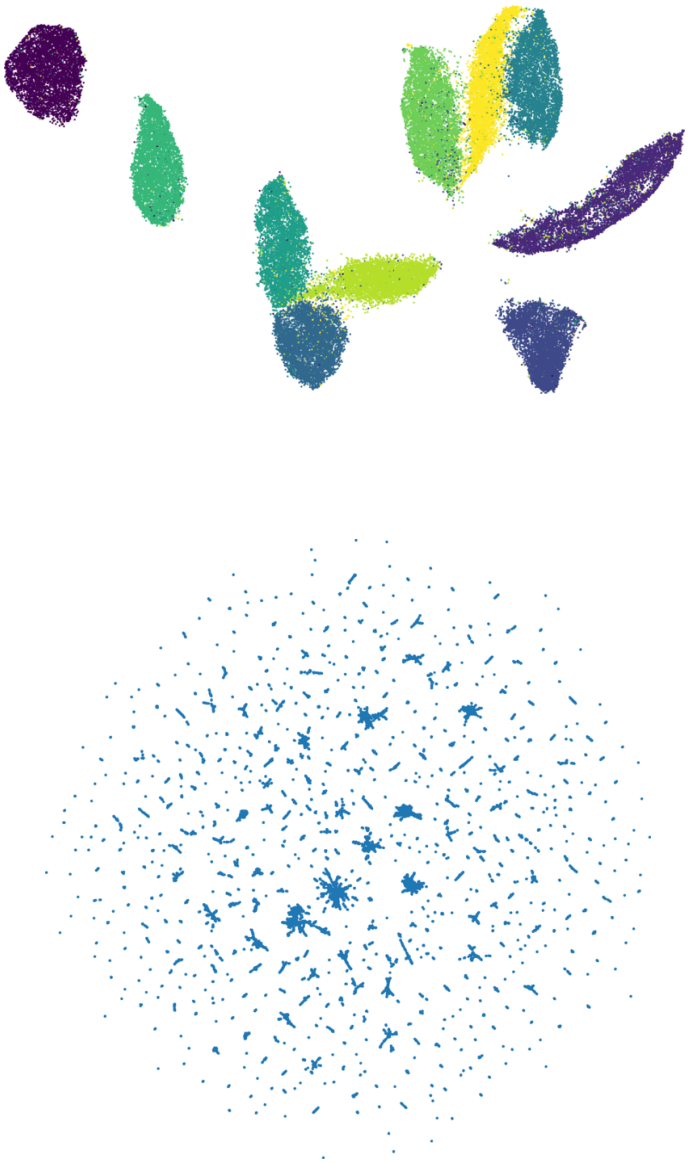


Figure 5: UMAP embeddings into the Euclidean plane. Top: MNIST dataset with a Manhattan input metric, colored by label. Bottom: HyperE-50 (WordNet) dataset with a hyperbolic input metric.

D Geometric primitives

Weighted average on a neighborhood. We now detail the results of Table 4. As described in Section 5.3, we consider batches of B point clouds $(x_i) \in \mathbb{R}^{\mathrm{N}\times\mathrm{D}}$, with $\mathrm{N} = 2{,}048$ and $\mathrm{D} = 3$. Depending on the memory footprint of the computations, B is equal to 1, 10 or 100 and ensures that GPU cores are used efficiently: batches of point clouds are encoded as large $(\mathrm{B}, \mathrm{N}, \mathrm{D})$ arrays. Following standard procedure for point cloud processing, we compute local features as:

$$a_i \leftarrow \frac{\sum_{j=1}^{\mathrm{M}} w(x_i, x_j) F(x_i, x_j)}{\sum_{j=1}^{\mathrm{M}} w(x_i, x_j)}\ , \quad \forall i \in [\![1, \mathrm{N}]\!] \tag{12}$$

where $w(x_i, x_j) \geqslant 0$ is a weight on the interaction (x_i, x_j) and F is a vector-valued function.

In our benchmarks, we consider two types of weight functions w:

1. A KNN window $w(x_i, x_j)$ which is equal to 1 if x_j is one of the $\mathrm{K} = 40$ nearest neighbors of x_i and 0 otherwise. It is implemented using a batched KNN search in dimension $\mathrm{D} = 3$ and advanced indexing operators. Just as in Table 3, we use KeOps to accelerate the construction of the KNN graph and otherwise rely on standard PyTorch syntax to build up local neighborhoods as $(\mathrm{B}, \mathrm{N}, \mathrm{K}, \mathrm{D})$ arrays.
2. A Gaussian window of radius $\sigma > 0$:
$$w(x_i, x_j) \ = \ \exp\big(-\|x_i - x_j\|^2 \,/\, 2\sigma^2\big)\ . \tag{13}$$
It is implemented using symbolic operations, as detailed below.

Local mean. In our first example, we compute the local average:

$$\mu_i \leftarrow \frac{\sum_{j=1}^{\mathrm{M}} w(x_i, x_j)\, x_j}{\sum_{j=1}^{\mathrm{M}} w(x_i, x_j)} \ \in \mathbb{R}^{\mathrm{D}}, \ \ \forall i \in [\![1, \mathrm{N}]\!]\ . \tag{14}$$

In the code below, we compute both the numerator and denominator in one pass through the data. The trick is to append a "1" to the feature vectors x_j in order to retrieve both:

$$w_i \leftarrow \sum_{j=1}^{\mathrm{M}} w(x_i, x_j) \cdot 1 \qquad \text{and} \qquad m_i \leftarrow \sum_{j=1}^{\mathrm{M}} w(x_i, x_j)\, x_j \tag{15}$$

with a single reduction call.

```python
def local_mean(points, radius):
    """
      points,  radius ->   means
    (B, N, D),    1   -> (B, N, D)
    """
    B, N, D = point.shape  # Batch-size, number of points, features
    points = points / radius  # Normalize the window size to 1

    # Add a "1" at the start of every vector, retrieve a (B, N, D+1) array:
    x = torch.cat((torch.ones_like(points[:,:,:1]), points), dim = -1)

    # Encode as symbolic tensors:
    x_i = LazyTensor(x.view(B, N, 1, D+1))  # (B, N, 1, D+1)
    x_j = LazyTensor(x.view(B, 1, N, D+1))  # (B, 1, N, D+1)

    # Neighborhood window - a Gaussian function:
    D_ij = ((x_i - x_j) ** 2).sum(-1)  # (B, N, N), squared distances
    K_ij = (- D_ij / 2).exp()          # (B, N, N), Gaussian kernel

    # Local sum:
    M_ij = K_ij * x_j  # (B, N, N, D+1)
    M_i  = M_ij.sum(dim = 2)  # (B, N, D+1) : weights and sums

    # Normalize by the sum of the weights:
    w_i = M_i[:,:,:1]  # (B, N, 1)
    m_i = M_i[:,:,1:]  # (B, N, D)
    return radius * m_i / w_i  # (B, N, D)
```

Local covariance. In our second example, we compute the local covariance matrices:

$$\Sigma_i \;\leftarrow\; \frac{\sum_{j=1}^{\mathrm{M}} w(x_i, x_j)\,(x_j - \mu_i)(x_j - \mu_i)^\top}{\sum_{j=1}^{\mathrm{M}} w(x_i, x_j)} \;\in\; \mathbb{R}^{\mathrm{D}\times\mathrm{D}}, \quad \forall i \in [\![1, \mathrm{N}]\!]\,, \tag{16}$$

where μ_i is defined as in (14). Using standard identities, we can rewrite this local descriptor as:

$$\Sigma_i \;\leftarrow\; \frac{1}{w_i}\big(c_i - \frac{1}{w_i} m_i m_i^\top\big)\,, \tag{17}$$

where:

$$w_i \;\leftarrow\; \sum_{j=1}^{\mathrm{M}} w(x_i, x_j)\,, \tag{18}$$

$$m_i \;\leftarrow\; \sum_{j=1}^{\mathrm{M}} w(x_i, x_j)\, x_j\,, \tag{19}$$

$$c_i \;\leftarrow\; \sum_{j=1}^{\mathrm{M}} w(x_i, x_j)\, x_j x_j^\top\,. \tag{20}$$

For optimal performances, we rely on the same trick as in (15) to compute all these quantities in one pass through the data. We append a "1" at the start of every vector x_j and compute:

$$\left[\begin{array}{c|c} w_i & m_i \\ \hline m_i^\top & c_i \end{array}\right] \;=\; C_i \;\leftarrow\; \sum_{j=1}^{\mathrm{M}} w(x_i, x_j)\,[1, x_j][1, x_j]^\top \tag{21}$$

```python
def local_covariance(points, radius):
    """
      points,  radius ->  covariances
    (B, N, D),    1   ->  (B, N, D, D)
    """
    B, N, D = point.shape  # Batch-size, number of points, features
    points = points / radius  # Normalize the window size to 1

    # Add a "1" at the start of every vector, retrieve a (B, N, D+1) array:
    x = torch.cat((torch.ones_like(points[:,:,:1]), points), dim=-1)  # (B, N, D+1)

    # Encode as symbolic tensors:
    x_i = LazyTensor(x[:,:,None,:])  # (B, N, 1, D+1)
    x_j = LazyTensor(x[:,None,:,:])  # (B, 1, N, D+1)

    # Neighborhood window - a Gaussian function:
    D_ij = ((x_i - x_j) ** 2).sum(-1)  # (B, N, N), squared distances
    K_ij = (- D_ij / 2).exp()          # (B, N, N), Gaussian kernel

    # Local sum - compute descriptors of order 0, 1 and 2:
    C_ij = (K_ij * x_j).tensorprod(x_j)  # (B, N, N, (D+1)*(D+1))
    C_i  = C_ij.sum(dim = 2).view(B, N, D+1, D+1)  # (B, N, D+1, D+1)

    # Extract local descriptors of order 0, 1 and 2:
    w_i = C_i[:,:,:1,:1]                # (B, N, 1, 1), weights
    m_i = C_i[:,:,:1,1:] *  radius      # (B, N, 1, D), sum
    c_i = C_i[:,:,1:,1:] * (radius**2)  # (B, N, D, D), outer products

    # Compute the covariance matrix:
    cov_i  = (c_i - (m_i.transpose(3, 2) * m_i) / w_i) / w_i  # (B, N, D, D)
    return cov_i
```

MLP features. Going further, we show how to use our library to compute neural features. Following standard practice in geometric deep learning, we rely on a multi-layer perceptron:

$$F \,:\, x \in \mathbb{R}^{\mathrm{D}} \;\mapsto\; A_2\,\mathrm{ReLU}(A_1 x + b_1) \;+\; b_2 \;\in\; \mathbb{R}^{\mathrm{O}} \tag{22}$$

parameterized by weight matrices $A_1 \in \mathbb{R}^{\mathrm{H}\times\mathrm{D}}$, $A_2 \in \mathbb{R}^{\mathrm{O}\times\mathrm{H}}$ and bias vectors $b_1 \in \mathbb{R}^{\mathrm{H}}$, $b_2 \in \mathbb{R}^{\mathrm{O}}$. "ReLU" denotes the rectified linear unit, or positive part, applied coordinate-wise on vectors of $\mathbb{R}^{\mathrm{H}}$. For the sake of simplicity, we compute the MLP correlations:

$$a_i \;\leftarrow\; \sum_{j=1}^{\mathrm{M}} w(x_i, x_j)\, F(x_j - x_i) \tag{23}$$

$$= \sum_{j=1}^{\mathrm{M}} w(x_i, x_j)\, \left(A_2\,\mathrm{ReLU}(f_j - f_i + b_1) + b_2\right), \tag{24}$$

where the hidden features $f_i = A_1 x_i$ are computed ahead of the sum reduction. We stress that the code below is fully differentiable: gradients can be computed with respect to all parameters.

We note that the matrix-vector product with A_2 is a $\mathcal{O}(\mathrm{OH})$ operation. In practice, our bruteforce CUDA engine is most efficient if the product $\mathrm{O}\cdot\mathrm{H}$ is smaller than 100: beyond this threshold, performance decrease sharply as in e.g. the fourth line of Table 4. KNN implementations are ideally suited to the computation of localized but complex features, whereas symbolic matrices let us compute efficiently simple descriptors at all scales.

```python
def MLP_features(points, A_1, B_1, A_2, B_2, radius):
    """
     points,  weights_1, bias_1, weights_2, bias_2, radius  ->  features
    (B, N, D), (H, D),    (H,),    (O, H),    (O,),   1     ->  (B, N, O)
    """
    B, N, D = points.shape
    x = points / radius  # Normalize the window size to 1

    # Apply the first linear operator on the features:
    f = points @ A_1.t()  # (B, N, H)

    # Encode the variables as symbolic tensors:
    # Positions:
    x_i = LazyTensor(x.view(B, N, 1, D))  # (B, N, 1, D)
    x_j = LazyTensor(x.view(B, 1, N, D))  # (B, 1, N, D)
    # Features:
    f_i = LazyTensor(f.view(B, N, 1, -1))  # (B, N, 1, H)
    f_j = LazyTensor(f.view(B, 1, N, -1))  # (B, 1, N, H)
    # MLP parameters:
    b_1 = LazyTensor(B_1.view(1, 1, 1, -1))  # (1, 1, 1, H)
    a_2 = LazyTensor(A_2.view(1, 1, 1, -1))  # (1, 1, 1, O * H)
    b_2 = LazyTensor(B_2.view(1, 1, 1, -1))  # (1, 1, 1, O)

    # Compute the MLP values:
    M_ij = (f_j - f_i + b_1).relu()    # (B, N, N, H)
    M_ij = a_2.matvecmult(M_ij) + b_2  # (B, N, N, O)

    # Neighborhood window - a Gaussian function:
    D_ij = ((x_i - x_j) ** 2).sum(-1)  # (B, N, N), squared distances
    K_ij = (- D_ij / 2).exp()  # (B, N, N), Gaussian kernel

    # Sum on the neighborhood:
    C_ij = K_ij * M_ij  # (B, N, N, O)
    features  = C_ij.sum(dim = 2)  # (B, N, O)

    return features
```

Chamfer loss. Beyond geometric descriptors, symbolic tensors let us work efficiently with global, geometric loss functions. If $(x_i) \in \mathbb{R}^{\mathrm{N}\times\mathrm{D}}$ and $(y_j) \in \mathbb{R}^{\mathrm{M}\times\mathrm{D}}$ are two clouds of N and M points in $\mathbb{R}^{\mathrm{D}}$, the "chamfer" or "soft-Hausdorff" loss between them reads:

$$\text{Chamfer}(x_i, y_j) \;=\; \frac{1}{2\mathrm{N}} \sum_{i=1}^{\mathrm{N}} \min_{j=1}^{\mathrm{M}} \|x_i - y_j\| \;+\; \frac{1}{2\mathrm{M}} \sum_{j=1}^{\mathrm{M}} \min_{i=1}^{\mathrm{N}} \|x_i - y_j\| \,. \tag{25}$$

Variants of this formula are used, for instance, in the Iterative Closest Point (ICP) algorithm. We leverage our fast nearest neighbor finder to implement it as follows:

```python
def squared_distances(x, y):
    """
      source,   target   -> squared distances
    (B, N, D), (B, M, D) ->    (B, N, M)
    """
    B, N, D = x.shape  # Batch size, number of source points, features
    _, M, _ = y.shape  # Batch size, number of target points, features

    # Encode as symbolic tensors:
    x_i = LazyTensor(x.view(B, N, 1, D))  # (B, N, 1, D)
    y_j = LazyTensor(y.view(B, 1, M, D))  # (B, 1, M, D)

    # Symbolic matrix of squared distances:
    D_ij = ((x_i - y_j)**2).sum(-1)     # (B, N, M), squared distances
    return D_ij


def chamfer_loss(x, y):
    """
      source,   target   -> loss values
    (B, N, D), (B, M, D) ->    (B,)
    """
    D_ij = squared_distances(x, y)  # (B, N, M) symbolic matrix
    D_xy = D_ij.min(dim=2).sqrt()   # (B, N), distances from x to y
    D_yx = D_ij.min(dim=1).sqrt()   # (B, M), distances from y to x
    return (D_xy.mean(dim=1) + D_yx.mean(dim=1)).view(-1) / 2  # (B,)
```

Energy distance. Going further, we can combine symbolic tensors and sum reductions to compute generic kernel norms, which produce smoother gradients for e.g. shape registration. These quantities are also known as Maximum Mean Discrepancies (MMDs) in statistics or generalized electrostatic energies in physics. As an example, the code below implements the Energy Distance [99]:

$$\begin{aligned} \text{ED}(x_i, y_j) \;=\; & \frac{1}{\mathrm{NM}} \textstyle\sum_{i=1}^{\mathrm{N}} \sum_{j=1}^{\mathrm{M}} \|x_i - y_j\| \\ & - \frac{1}{2\mathrm{N}^2} \textstyle\sum_{i=1}^{\mathrm{N}} \sum_{j=1}^{\mathrm{N}} \|x_i - x_j\| \;-\; \frac{1}{2\mathrm{M}^2} \sum_{i=1}^{\mathrm{M}} \sum_{j=1}^{\mathrm{M}} \|y_i - y_j\| \,. \end{aligned} \tag{26}$$

```python
def energy_distance(x, y):
    """
      source,   target   -> loss values
    (B, N, D), (B, M, D) ->    (B,)
    """
    N, M = x.shape[1], y.shape[1]  # Numbers of source and target points

    D_xy = squared_distances(x, y).sqrt().sum(dim=2)  # (B, N), distances x<->y
    D_xx = squared_distances(x, x).sqrt().sum(dim=2)  # (B, N), distances x<->x
    D_yy = squared_distances(y, y).sqrt().sum(dim=2)  # (B, M), distances y<->y

    return (D_xy.sum(dim=1) /   (N*M)
          - D_xx.sum(dim=1) / (2*N*N)
          - D_yy.sum(dim=1) / (2*M*M)).view(-1)  # (B,)
```

E Optimal transport

The optimal transport problem. As discussed in Section 5.3, optimal transport generalizes sorting to spaces of dimension $\mathrm{D} > 1$. We now consider two point clouds $(x_i) \in \mathbb{R}^{\mathrm{N}\times\mathrm{D}}$, $(y_j) \in \mathbb{R}^{\mathrm{M}\times\mathrm{D}}$ with non-negative weights $(\alpha_i) \in \mathbb{R}^{\mathrm{N}}_{\geqslant 0}$ and $(\beta_j) \in \mathbb{R}^{\mathrm{M}}_{\geqslant 0}$ that sum up to 1. These arrays encode two discrete probability measures α and β on $\mathbb{R}^{\mathrm{D}}$, understood as weighted sums of Dirac masses δ_x:

$$\alpha \;=\; \sum_{i=1}^{\mathrm{N}} \alpha_i \delta_{x_i} \qquad\qquad \text{and} \qquad\qquad \beta \;=\; \sum_{j=1}^{\mathrm{M}} \beta_j \delta_{y_j} \;. \tag{27}$$

If $\mathbf{C}(x_i, y_j)$ denotes an arbitrary cost function on $\mathbb{R}^{\mathrm{D}} \times \mathbb{R}^{\mathrm{D}}$, the optimal transport cost between the two discrete measures α and β reads:

$$\mathrm{OT}(\alpha_i, x_i, \beta_j, y_j) \;=\; \min_{(\pi_{i,j}) \in \mathbb{R}^{\mathrm{N}\times\mathrm{M}}_{\geqslant 0}} \;\sum_{i=1}^{\mathrm{N}} \sum_{j=1}^{\mathrm{M}} \pi_{i,j}\, \mathbf{C}(x_i, y_j) \tag{28}$$

$$\text{subject to} \quad \forall\, i, j, \;\; \pi_{i,j} \;\geqslant\; 0, \quad (\pi \mathbf{1})_i \;=\; \sum_{j=1}^{\mathrm{M}} \pi_{i,j} \;=\; \alpha_i, \quad (\pi^\top \mathbf{1})_j \;=\; \sum_{i=1}^{\mathrm{N}} \pi_{i,j} \;=\; \beta_j \;.$$

The optimal transport plan $(\pi_{i,j})$ is a non-negative (N, M) array whose lines sum up to (α_i) and whose columns sum up to (β_j). In the remainder of this section, we use the quadratic cost $\mathbf{C}(x_i, y_j) = \frac{1}{2}\|x_i - y_j\|^2$: up to a factor $1/2$, the cost value $\mathrm{OT}(\alpha_i, x_i, \beta_j, y_j)$ is the squared Wasserstein-2 distance between α and β.

Dual problem. A fundamental remark was made by Kantorovitch in [68]: the linear optimization problem (28) is equivalent to a simpler dual problem on variables of size N and M:

$$\mathrm{OT}(\alpha_i, x_i, \beta_j, y_j) \;=\; \max_{\substack{(f_i) \in \mathbb{R}^{\mathrm{N}} \\ (g_j) \in \mathbb{R}^{\mathrm{M}}}} \sum_{i=1}^{\mathrm{N}} \alpha_i f_i + \sum_{j=1}^{\mathrm{M}} \beta_j g_j \quad \text{s.t. } \forall\, i, j, \;\; f_i + g_j \;\leqslant\; \mathbf{C}(x_i, y_j) \;. \tag{29}$$

The dual vectors (f_i) and (g_j) are unique up to an additional constant. They are often understood as the sampled values $f_i = f(x_i)$, $g_j = g(y_j)$ of continuous dual potentials on the input point clouds.

Entropic regularization. Optimal transport solvers compute the optimal dual vectors (f_i) and (g_j) associated to any discrete input configuration $(\alpha_i, x_i, \beta_j, y_j)$. To this end, a common strategy is to add a small entropic barrier to the primal problem (28). If $\varepsilon > 0$ is a positive temperature, we can apply the Fenchel-Rockafellar theorem and write the regularized primal and dual problems as:

$$\begin{aligned} \mathrm{OT}_\varepsilon(\alpha_i, x_i, \beta_j, y_j) \;=\; &\min_{(\pi_{i,j}) \in \mathbb{R}^{\mathrm{N}\times\mathrm{M}}_{\geqslant 0}} \;\sum_{i=1}^{\mathrm{N}} \sum_{j=1}^{\mathrm{M}} \pi_{i,j}\, \mathbf{C}(x_i, y_j) \\ &+\; \varepsilon \sum_{i=1}^{\mathrm{N}} \sum_{j=1}^{\mathrm{M}} \pi_{i,j} \log \tfrac{\pi_{i,j}}{\alpha_i \beta_j} - \pi_{i,j} + \alpha_i \beta_j \end{aligned} \tag{30}$$

$$\text{subject to} \quad \forall\, i, j, \;\; \pi_{i,j} \;\geqslant\; 0, \quad (\pi \mathbf{1})_i \;=\; \sum_{j=1}^{\mathrm{M}} \pi_{i,j} \;=\; \alpha_i, \quad (\pi^\top \mathbf{1})_j \;=\; \sum_{i=1}^{\mathrm{N}} \pi_{i,j} \;=\; \beta_j$$

$$= \max_{\substack{(f_i) \in \mathbb{R}^{\mathrm{N}} \\ (g_j) \in \mathbb{R}^{\mathrm{M}}}} \sum_{i=1}^{\mathrm{N}} \alpha_i f_i \;+\; \sum_{j=1}^{\mathrm{M}} \beta_j g_j \;+\; \varepsilon \sum_{i=1}^{\mathrm{N}} \sum_{j=1}^{\mathrm{M}} \alpha_i \beta_j \left(1 - \exp \tfrac{1}{\varepsilon} \big[f_i + g_j - \mathbf{C}(x_i, y_j) \big] \right) . \tag{31}$$

Up to a small perturbation, the optimal transport problem can thus be reduced to the resolution of (31), a concave maximization problem on the dual vectors $(f_i) \in \mathbb{R}^{\mathrm{N}}$, $(g_j) \in \mathbb{R}^{\mathrm{M}}$ that is smooth and without constraints. The optimal dual potentials encode, implicitly, an optimal transport plan:

$$\pi_{i,j} \;=\; \alpha_i \beta_j \, \exp \tfrac{1}{\varepsilon} \big[f_i + g_j - \mathbf{C}(x_i, y_j) \big] \tag{32}$$

that satisfies the marginal constraints of (30), with an optimal transport cost that reads:

$$\mathrm{OT}_\varepsilon(\alpha_i, x_i, \beta_j, y_j) \;=\; \sum_{i=1}^{\mathrm{N}} \alpha_i f_i \;+\; \sum_{j=1}^{\mathrm{M}} \beta_j g_j \;. \tag{33}$$

The Sinkhorn algorithm. The standard Sinkhorn algorithm is equivalent to an alternate maximization of (31) with respect to the dual vectors (f_i) and (g_j). Starting from null potentials $f_i = 0$ and $g_j = 0$, its updates read:

$$f_i \;\leftarrow\; -\varepsilon \log \sum_{j=1}^{\mathrm{M}} \beta_j \, \exp \tfrac{1}{\varepsilon} \big[g_j - \mathbf{C}(x_i, y_j) \big] \;, \qquad \forall i \in [\![1, \mathrm{N}]\!] \;, \tag{34}$$

$$g_j \;\leftarrow\; -\varepsilon \log \sum_{i=1}^{\mathrm{N}} \alpha_i \, \exp \tfrac{1}{\varepsilon} \big[f_i - \mathbf{C}(x_i, y_j) \big] \;, \qquad \forall j \in [\![1, \mathrm{M}]\!] \;. \tag{35}$$

This method has been (re-)discovered in many applied fields since the 1960's [27, 29, 34, 41, 44, 94, 98, 112, 117], with minor variations on the exact formulation of the regularized problem. Most authors work with the exponentiated variables:

$$u_i \;=\; \exp(f_i/\varepsilon) \qquad\qquad \text{and} \qquad\qquad v_j \;=\; \exp(g_j/\varepsilon) \;. \tag{36}$$

The dual variables $u = (u_i) \in \mathbb{R}^{\mathrm{N}}_{>0}$ and $v = (v_j) \in \mathbb{R}^{\mathrm{M}}_{>0}$ are then initialized as uniform vectors of 1, with updates that read:

$$u \;\leftarrow\; \frac{1}{K(\beta v)} \qquad\qquad \text{and} \qquad\qquad v \;\leftarrow\; \frac{1}{K^\top(\alpha u)} \;. \tag{37}$$

In the equations above, the inversions and multiplications βv, αu are applied coordinate-wise. The (N, M) matrix $K = (K_{i,j})$ is the Gibbs kernel associated to $\mathbf{C}(x_i, y_j)$ at temperature ε with coefficients:

$$K_{i,j} \;=\; \exp\big(-\mathbf{C}(x_i, y_j)/\varepsilon \big) \;. \tag{38}$$

When $\mathbf{C}(x_i, y_j) = \frac{1}{2}\|x_i - y_j\|^2$, K is a Gaussian kernel matrix of deviation $\sigma = \sqrt{\varepsilon}$: this quantity is best understood as the **blur scale** of the Gaussian smoothing that we apply on the transport plan $\pi_{i,j}$ to lower the complexity of the optimization problem.

Stabilization. As detailed in the `sinkhorn_loop_simple` routine below, our library can be used to implement efficiently the exponentiated Sinkhorn updates of (37). In practice though, these iterations may induce numerical overflows and are notoriously unstable when $\sqrt{\varepsilon}$ is too small. Following [23, 37, 70], we rely instead on symmetrized updates performed in the logarithmic domain:

$$\widetilde{f}_i \;\leftarrow\; -\varepsilon \log \textstyle\sum_{j=1}^{\mathrm{M}} \beta_j \, \exp \frac{1}{\varepsilon} \big[g_j - \mathbf{C}(x_i, y_j) \big] \;, \qquad \forall i \in [\![1, \mathrm{N}]\!] \;, \tag{39}$$

$$\widetilde{g}_j \;\leftarrow\; -\varepsilon \log \textstyle\sum_{i=1}^{\mathrm{N}} \alpha_i \, \exp \frac{1}{\varepsilon} \big[f_i - \mathbf{C}(x_i, y_j) \big] \;, \qquad \forall j \in [\![1, \mathrm{M}]\!] \;, \tag{40}$$

$$f_i \;\leftarrow\; \tfrac{1}{2}(f_i + \widetilde{f}_i) \;, \qquad \forall\, i \in [\![1, \mathrm{N}]\!] \;, \tag{41}$$

$$g_j \;\leftarrow\; \tfrac{1}{2}(g_j + \widetilde{g}_j) \;, \qquad \forall j \in [\![1, \mathrm{M}]\!] \;. \tag{42}$$

This robust algorithm is implemented in the `sinkhorn_loop_stable` routine detailed below. Our CUDA engine performs the `.logsumexp()` reduction using an online version of the Log-Sum-Exp trick – with a running maximum – that guarantees numerical stability with a negligible computational overhead.

Annealing. In practice, the Sinkhorn loop converges to a set numerical tolerance in $\mathcal{O}(\max_{i,j} \mathbf{C}(x_i, y_j) \,/\, \varepsilon)$ iterations. To accelerate convergence, a common heuristic is to let the temperature ε decrease following an exponential annealing schedule [71]. If Δ is an estimation of the diameter $\max_{i,j} \|x_i - y_j\|$, ε is a target temperature and n_{its} is a prescribed number of iterations, we use decreasing values of the temperature:

$$\varepsilon_n \;=\; \Delta^2 \, q^n \qquad\qquad \text{with} \qquad\qquad q \;=\; (\varepsilon/\Delta^2)^{1/n_{\text{its}}} \tag{43}$$

at every iteration $n \in [\![1, n_{\text{its}}]\!]$ of the Sinkhorn loop. For faster convergence, the dual potentials are initialized using a closed-form solution of (31) when $\varepsilon = +\infty$:

$$f_i \;=\; \sum_{j=1}^{\mathrm{M}} \beta_j \, \mathbf{C}(x_i, y_j) \qquad\qquad \text{and} \qquad\qquad g_j \;=\; \sum_{i=1}^{\mathrm{N}} \alpha_i \, \mathbf{C}(x_i, y_j) \;. \tag{44}$$

Overall, this method usually lets the Sinkhorn loop converge to a satisfying tolerance in 5 to 20 iterations, even for small values of ε.

```python
def sinkhorn_loop_simple(a, x, b, y, eps, nits):
    """
    weights, points,  weights', points'    ->  f(x),   g(y)
    (B, N), (B, N, D), (B, M), (B, M, D),  -> (B, N), (B, M)
    """
    B, N, D = x.shape  # Batch size, source points, features
    _, M, _ = y.shape  # Batch size, target points, features

    # Dual variables, (B, N) and (B, M):
    a, b = a.view(B, N, 1), b.view(B, M, 1)
    u_x, v_y = torch.ones_like(a), torch.ones_like(b)
    # Encoding as symbolic tensors:
    x_i = LazyTensor(x.view(B, N, 1, D))  # (B, N, 1, D)
    y_j = LazyTensor(y.view(B, 1, M, D))  # (B, 1, M, D)

    # Symbolic cost matrix and Gibbs kernel:
    C_ij = ((x_i - y_j) ** 2).sum(-1) / 2  # (B, N, M)
    K_ij = (- C_ij / eps).exp()            # (B, N, M)

    # Sinkhorn iterations:
    for _ in range(nits):
        u_x = 1 / (K_ij     @ (b * v_y))  # (B, N, M) @ (B, M, 1) = (B, N, 1)
        v_y = 1 / (K_ij.t() @ (a * u_x))  # (B, M, N) @ (B, N, 1) = (B, M, 1)

    f_x, g_y = eps * u_x.log(), eps * v_y.log()
    return f_x.view(B, N), g_y.view(B, M)


def sinkhorn_loop_stable(a, x, b, y, eps, nits):
    """
    weights, points,  weights', points'    ->  f(x),   g(y)
    (B, N), (B, N, D), (B, M), (B, M, D),  -> (B, N), (B, M)
    """
    B, N, D = x.shape  # Batch size, source points, features
    _, M, _ = y.shape  # Batch size, target points, features

    # Dual potentials, (B, N) and (B, M):
    f_x, g_y = torch.zeros_like(a), torch.zeros_like(b)
    # Log of the weights, (B, N) and (B, M):
    a_logs, b_logs = a.log(), b.log()

    # Encoding as symbolic tensors:
    # Points:
    x_i = LazyTensor(x.view(B, N, 1, D))  # (B, N, 1, D)
    y_j = LazyTensor(y.view(B, 1, M, D))  # (B, 1, M, D)
    # Dual potentials:
    f_i = LazyTensor(f_x.view(B, N, 1, 1))  # (B, N, 1, 1)
    g_j = LazyTensor(g_y.view(B, 1, M, 1))  # (B, 1, M, 1)
    # Log-weights:
    log_a_i = LazyTensor(a_logs.view(B, N, 1, 1))  # (B, N, 1, 1)
    log_b_j = LazyTensor(b_logs.view(B, 1, M, 1))  # (B, 1, M, 1)

    # Symbolic cost matrix:
    C_ij = ((x_i - y_j) ** 2).sum(-1) / 2   # (B, N, M, 1)

    # Symmetric Sinkhorn iterations, written in the log-domain:
    for _ in range(nits):
        ft_x = - eps * ((g_j - C_ij) / eps + log_b_j).logsumexp(dim=2).squeeze(-1)
        gt_y = - eps * ((f_i - C_ij) / eps + log_a_i).logsumexp(dim=1).squeeze(-1)
        # Use in-place updates to keep a small memory footprint:
        f_x[:] = (f_x + ft_x) / 2
        g_y[:] = (g_y + gt_y) / 2

    return f_x, g_y
```

Multiscale solvers. Going further, a recent line of work puts the emphasis on multiscale implementations of the Sinkhorn loop [10, 37, 96]. Following these papers, we use our library to provide a two-scale solver for the regularized optimal transport problem (31). Its behaviour can be described in four steps:

1. We compute **coarse approximations** of the input measures α and β. In practice, we use a K-means clustering with $\mathrm{N}_c = \sqrt{\mathrm{N}}$ (resp. $\mathrm{M}_c = \sqrt{\mathrm{M}}$) clusters on the input point clouds $(x_i) \in \mathbb{R}^{\mathrm{N}\times\mathrm{D}}$ (resp. $(y_j) \in \mathbb{R}^{\mathrm{M}\times\mathrm{D}}$): each cluster is represented by its centroid $\overline{x}_i$ with total weight $\overline{\alpha}_i$ (resp. $\overline{y}_j$ with total weight $\overline{\beta}_j$). This corresponds to a quantization of the discrete measures α and β, as:

$$\sum_{i=1}^{\mathrm{N}_c} \overline{\alpha}_i \delta_{\overline{x}_i} \;\simeq\; \sum_{i=1}^{\mathrm{N}} \alpha_i \delta_{x_i} \qquad \text{and} \qquad \sum_{j=1}^{\mathrm{M}_c} \overline{\beta}_j \delta_{\overline{y}_j} \;\simeq\; \sum_{j=1}^{\mathrm{M}} \beta_j \delta_{y_j} \tag{45}$$

   for the weak-$\star$ topology, as measured e.g. by the Wasserstein-2 distance. We sort the (N, D) and (M, D) arrays (x_i) and (y_j) to ensure that all the clusters are contiguous in memory.

2. We start the annealing descent on the **coarse measures**. We use the stabilized iterations of `sinkhorn_loop_stable` on the symbolic $(\mathrm{N}_c, \mathrm{M}_c)$ cost matrix $\mathbf{C}(\overline{x}_i, \overline{y}_j)$ and update coarse dual vectors $(\overline{f}_i) \in \mathbb{R}^{\mathrm{N}_c}$, $(\overline{g}_j) \in \mathbb{R}^{\mathrm{M}_c}$.

3. When the blur scale $\sqrt{\varepsilon_n}$ goes below the largest diameter of the K-means clusters, we perform a **coarse-to-fine extrapolation step**. We use the optimality equations (34-35) to extrapolate the $\overline{f}_i$'s and $\overline{g}_j$'s, supported by the $\overline{x}_i$'s and $\overline{y}_j$'s, onto new values $(f_i) \in \mathbb{R}^{\mathrm{N}}$ and $(g_j) \in \mathbb{R}^{\mathrm{M}}$ supported by the x_i's and y_j's. As discussed in Section 4, we also compute a block-sparsity mask on the symbolic (N, M) cost matrix $\mathbf{C}(x_i, y_j)$: following [96], it corresponds to pruning out pair-wise interactions between clusters such that:

$$\overline{f}_i + \overline{g}_j \;<\; \mathbf{C}(\overline{x}_i, \overline{y}_j) \;-\; \tau\varepsilon_n \;, \tag{46}$$

   where τ is a cutoff parameter that we set to 5, since $1 \gg \exp(-5)$.

4. We perform the last iterations of the stabilized Sinkhorn loop on the **full point clouds** (x_i) and (y_j): these updates correspond to the values of $\sqrt{\varepsilon_n}$ that range between the average cluster diameter and the target blur value $\sqrt{\varepsilon}$. We use the block-sparsity mask computed at step 3 to prune out negligible interactions from the full (N, M) cost matrix $\mathbf{C}(x_i, y_j)$: this is a GPU-friendly implementation of the **kernel truncation** trick of [96].

The resulting code is too technical to fit in these supplementary materials: we package and fully document this solver on our website (`www.kernel-operations.io/geomloss`) [37, 40].

Benchmarks. As discussed above, our library is well suited to research in optimal transport theory: simple algorithms and advanced solvers can all be implemented with symbolic LazyTensors. To showcase the performances of our implementations, we now benchmark several solvers: a baseline linear solver for the exact transport problem, implemented in C++ on the CPU [13, 42]; a stabilized Sinkhorn loop with $1/\varepsilon$ iterations, implemented using either PyTorch JIT or KeOps; a stabilized Sinkhorn loop with annealing and 10 iterations, implemented using either PyTorch JIT or KeOps; a multiscale solver, implemented with KeOps as discussed above. In practice, the parameters of our solvers ensure that the relative error made on the regularized Wasserstein-2 "distance" $\sqrt{2 \cdot \mathrm{OT}_\varepsilon(\alpha_i, x_i, \beta_j, y_j)}$ is always smaller than 1%. This level of accuracy is satisfying for most practical purposes in shape analysis and machine learning.

To illustrate the two main use cases of optimal transport theory in the field, we tackle two separate problems:

1. A high-precision matching in dimension $\mathrm{D} = 3$. The input point clouds (x_i) and (y_j) are sampled from the Stanford dragon and are deformed using random affine transformations. They are then centered and normalized: we use an estimate $\Delta = 2$ of the diameter of the configuration in the annealing descent. The blur scale $\sqrt{\varepsilon}$ is set to `0.01`: we retrieve a precise transport plan $\pi_{i,j}$ with (32) that is essentially accurate up to a $< 1\%$ tolerance.

2. A low-precision matching in dimension $\mathrm{D} = 25$. The input point clouds (x_i) and (y_j) are sampled from the Glove-25 dataset and are deformed using random affine transformations. They are then centered and normalized: we use an estimate $\Delta = 2$ of the diameter of the configuration in the annealing descent. The blur scale $\sqrt{\varepsilon}$ is set to `0.3`: we retrieve a fuzzy transport plan $\pi_{i,j}$ with (32) that captures large-scale deformations while being relatively robust to statistical noise.

Our results are summarized in Table 6 and discussed at the end of the main paper. In practice KeOps consistently improves the runtimes of optimal transport solvers on the GPU: researchers can now scale up their methods to large datasets without memory overflows. As predicted by the theory, multiscale strategies are most useful for large point clouds in low-dimensional spaces ($\mathrm{N} \geqslant$100k, $\mathrm{D} \leqslant 3$), while annealing strategies provide a good deterministic baseline in all the other settings.

Table 6: Scaling up optimal transport to large datasets.

N	D	$\sqrt{\varepsilon}$	POT Exact	PyTorch $\rightarrow$ Sinkhorn $\rightarrow$ $1/\varepsilon$ its	**Ours** Sinkhorn $1/\varepsilon$ its	PyTorch $\rightarrow$ annealing 10 its	**Ours** annealing 10 its	**Ours** multiscale 10 its
`1k`	`3`	`.01`	121 ms	2,000 ms	241 ms	1,960 µs	**82 µs**	25.7 ms
`10k`	`3`	`.01`	12.2 s	203 s	7.65 s	211 ms	**8 ms**	26 ms
`100k`	`3`	`.01`	∞	mem	645 s	mem	669 ms	**230 ms**
`1M`	`3`	`.01`	∞	mem	∞	mem	62 s	**2.70 s**
`1k`	`25`	`.3`	143 ms	2,200 µs	375 µs	1,960 µs	**360 µs**	36.5 ms
`10k`	`25`	`.3`	12.6 s	227 ms	35 ms	211 ms	**34 ms**	101 ms
`100k`	`25`	`.3`	∞	mem	3.48 s	mem	**3.37 s**	3.40 s
`1M`	`25`	`.3`	∞	mem	319 s	mem	338 s	**294 s**

F Structure of the inner KeOps++ engine

This Section provides an overview of the low-level structure of the KeOps engine: more details and explanations can be found on our website (`www.kernel-operations.io`).

The compilation stack. As described in Section 4, effective KeOps computations are triggered by reductions over one of the "symbolic" axes of a `LazyTensor`, at positions -2 or -3. Calculations are performed by custom binaries that are generated as required by the engine, and stored on the hard drive for later use. Under the hood, the formula F of Eq. (1) is encoded as a string of characters that is attached to the `LazyTensor` object: this simple descriptor is sent to the `C++/CUDA` compiler via a preprocessor macro through the `cmake` build engine.

In practie, after the compilation step, two dynamic libraries are generated with extension `.so` on Unix, `.dll` on Windows or `.dylib` on MacOS. The first one contains the `C++/CUDA` functions that perform the actual computation on the GPU, whereas the second one makes the interface between the `C++/CUDA` code and Python via the `PyBind11` library [61]. We note that this second shared object (the binder) can be changed to fit the requirements of other scripted languages: for instance, our `gnu/R` interface relies on the `Rcpp` [33] framework. Each binary has a unique name, created using a standard hash function, that identifies the formula F and several other parameters: the Python version, GPU Id, *etc*... KeOps binaries are ultimately gathered in a cache directory that is listed in the `PYTHON_PATH`: they can be imported from `Python` using the standard `import` statement.

Building formulas, automatic differentiation. Internally, KeOps encodes formulas as recursively templated `C++` classes: every single mathematical operation that make up our formulas is defined as a templated `struct` that takes a sub-formula as an input. The recursion ends when the compiler encounters a class that corresponds to a variable or a constant, whose value is known. Every KeOps `struct` that encodes a mathematical operator comes with two attributes:

1. a *forward* function that implements the actual computation in `C++/CUDA`. This piece of code will be inlined in the final `CUDA` kernel.

2. a *backward* function that encodes a symbolic expression for the gradient (i.e. the adjoint of the differential), expressed using KeOps recursive templates.

As an example, the element-wise, vector-valued exponential function is encoded with:

```
template < class F >
struct Exp : UnaryOp< Exp, F > {
  // dimension of the output: Exp(F) has the same dimension as F
  static const int DIM = F::DIM

  // Forward: actual computation, to be inlined inside the Cuda code
  static DEVICE INLINE void Operation(TYPE *out, TYPE *in) {
    #pragma unroll
    for (int k=0; k<DIM; k++) { out[k] = exp(in[k]); }
  }

  // Backward: templated expression for the adjoint of the differential
  // operator of Exp w.r.t. the variable V, and applied to GRADIN input
  // vector: ∇_V(Exp(F)).GRADIN = ∇_V(F).(Exp(F)×GRADIN)
  template < class V, class GRADIN >
  using DiffT = typename F::template DiffT< V, Mult< Exp< F >, GRADIN > >;
};
```

Using similar definitions for other mathematical operations, we can then express a Gaussian matrix-vector product:

$$F(x, y, b) = \sum_j \exp(-\|x_i - y_j\|^2)\, b_j \tag{47}$$

as a sum reduction over the index "j" of the formula:

```
auto F = Scal< Exp< Minus< SqDist< X, Y> > >, B >
auto SF = Sum_Reduction< F, 0 >  // the 0 flag specifies a reduction over j
```

In the code above, `X`, `Y` and `B` are special classes that represent data loaders (i.e. the variables that are fed to the symbolic formula). `LazyTensor` objects build such templated formulas on their own, whenever required: end-users only have to deal with our high-level `Python` syntax.

The templated structure of our inner engine has two main advantages. First, the code for evaluating the full formula F is built up at compile time, which allows the `C++/CUDA`compiler to optimize the resulting code. Many checks can be performed during the compilation (*e.g.* with `static_assert` expressions) to avoid overheads at run time. Moreover, all loops whose indices are known at compile time (e.g. to compute the norm of a vector of size D) are unrolled aggressively.

Second, we can use the recursive template mechanics to implement a fully-fledged automatic differentiation engine. We recall here that a given KeOps shared object can only compute a single formula. Consequently, in order to compute the gradient ∇F of a formula F, we need to build new shared objects to take care of partial derivatives with respect to all the input variables. As in the forward evaluation, this is done on-the-fly during the call to the Pytorch `.backward()` or `.grad()` methods in a way that is transparent to end-users. For instance, the partial derivative of a Gaussian matrix-vector product with respect to a variable x, can be computed by adding the symbolic `Grad< >` operator in front of the formula `SF` that encodes the sum reduction of a formula `F`:

```
auto GSF = Grad< SF, X, E >
```

Here, the input variable `E` is the gradient to back-propagate from the outputs. The effect of the `Grad< >` operator is then surprisingly simple: instead of injecting the code of the forward function, the compiler inlines the code contained in the `DiffT` method – the backward function. The resulting `C++/CUDA` function then outputs the chain rule derivative of `SF` without hassle.