Kernel Operations on the GPU, with Autodiff,without Memory Overflows

Benjamin Charlier, Jean Feydy, Joan Glaunès, François-David Collin, Ghislain Durif

Kernel Operations on the GPU, with Autodiff, without Memory Overflows

Benjamin Charlier* BENJAMIN.CHARLIER@UMONTPELLIER.FR
IMAG
Université de Montpellier, CNRS
Montpellier, France

Jean Feydy* JEAN.FEYDY@ENS.FR
DMA
École Normale Supérieure
Paris, France

Joan Alexis Glaunès* ALEXIS.GLAUNES@PARISDESCARTES.FR
MAP5
Université de Paris, CNRS
Paris, France

François-David Collin FRANCOIS-DAVID.COLLIN@UMONTPELLIER.FR
Ghislain Durif GHISLAIN.DURIF@UMONTPELLIER.FR
IMAG
Université de Montpellier, CNRS
Montpellier, France

* equal contribution

Abstract

The `KeOps` library provides a fast and memory-efficient GPU support for tensors whose entries are given by a mathematical formula, such as kernel and distance matrices. `KeOps` alleviates the main bottleneck of tensor-centric libraries for kernel and geometric applications: memory consumption. It also supports automatic differentiation and outperforms standard GPU baselines, including `PyTorch CUDA` tensors or the `Halide` and `TVM` libraries. `KeOps` combines optimized `C++/CUDA` schemes with binders for high-level languages: `Python` (`Numpy` and `PyTorch`), `Matlab` and `GNU R`. As a result, high-level "quadratic" codes can now scale up to large data sets with millions of samples processed in seconds. `KeOps` brings graphics-like performances for kernel methods and is freely available on standard repositories (PyPi, CRAN). To showcase its versatility, we provide tutorials in a wide range of settings online at `www.kernel-operations.io`.

Keywords: kernel methods, Gaussian processes, GPU, automatic differentiation

1. Introduction

Recent advances in machine learning have been driven by the diffusion of two pieces of software: automatic differentiation and GPU backends for tensor computations. Today, thanks to e.g. the `TensorFlow` or `PyTorch` libraries (Abadi et al., 2015; Paszke et al., 2017), users routinely perform gradient descent on functions that involve millions of parameters.

These high-level `Python` frameworks unlock the use of massively parallel hardware for machine learning research. Under the hood, they rely on `C++` routines that are often supported by hardware manufacturers: the `cuBLAS` and `cuDNN` libraries edited by Nvidia provide the binaries for linear algebra and convolutions that power a majority of deep learning models. In practice, the presence of a complete software stack (from low-level binaries to well-documented `Python` libraries) is a prerequisite for the widespread adoption of a research idea by the machine learning community. The `KeOps` library intends to provide such a solid numerical foundation for all methods that involve large distance or kernel matrices. A motivating example is the computation of pair-wise interactions of the form:

$$a_i \;\leftarrow\; \sum_{j=1}^{\mathrm{N}} \exp(-\|x_i - y_j\|^2/2\sigma^2)\; b_j \;=\; \sum_{j=1}^{\mathrm{N}} k(x_i, y_j)\, b_j \qquad \text{for} \quad i = 1, \dots, \mathrm{M} \tag{1}$$

where M and N range from a hundred to a billion, $x_1, \dots, x_{\mathrm{M}}$ and $y_1, \dots, y_{\mathrm{N}}$ all belong to a common vector space $\mathbb{R}^{\mathrm{D}}$, the signals $b_1, \dots, b_{\mathrm{N}}$ are real numbers and $k(x, y)$ is a Gaussian kernel of deviation $\sigma > 0$. This operation is often understood as a matrix-vector product with a Gaussian kernel matrix $(K_{i,j}) = (k(x_i, y_j))$, or equivalently as a convolution with the kernel function k that is sampled on the point clouds (x_i) and (y_j).

To perform this computation, common practice is to create an explicit M-by-N buffer $(K_{i,j})$ and compute Eq. (1) with a linear algebra routine. Unfortunately, this method requires the *storage* of the kernel matrix as a contiguous array in memory and does not scale when M and N are in the order of 50k or more. To work around this problem, a common strategy is thus to decompose Eq. (1) as a collection of smaller matrix-vector products using a `Python` or `Matlab` "`for`" loop. This method alleviates memory issues but remains inefficient in spaces of dimension $\mathrm{D} \leqslant 100$: the *transfer* of the tiles of the kernel matrix $(K_{i,j})$ between different layers of GPU memory remains a narrow bottleneck for computations. `KeOps` leverages efficient `C++` schemes from the graphics literature (Nguyen, 2007, Chapter 31) to streamline the use of registers and reach optimal performance in this setting. It is fast, transparent to use and targets a single yet powerful abstraction: semi-symbolic arrays whose entries $M_{i,j}$ are given by a mathematical formula "$F(x_i, y_j)$", evaluated on data arrays that are indexed by line and column numbers "i" and "j". It can be called from the major scripting languages used in the scientific community: `Python` (`Numpy` and `PyTorch`), `Matlab`, and `GNU R`. In this introduction, we focus on the `Python` interface and refer to our documentation for additional tutorials, applications and benchmarks.

2. `KeOps` Purpose and Usage

A generic reduction framework. The workhorse of the `KeOps` library is a `C++` engine for generic reductions on sampled data. Let us assume that we have at hand:

1. **parameters**: a collection $p^1 \in \mathbb{R}^{d_p^1}, \dots, p^P \in \mathbb{R}^{d_p^P}$ of vectors;
2. ***i*-variables**: a collection $x^1 \in \mathbb{R}^{M \times d_x^1}, \dots, x^X \in \mathbb{R}^{M \times d_x^X}$ of matrices, with rows indexed by $i \in [\![1, \mathrm{M}]\!]$ (hence for each k and i, x_i^k is a vector in $\mathbb{R}^{d_x^k}$);
3. ***j*-variables**: a collection $y^1 \in \mathbb{R}^{N \times d_y^1}, \dots, y^Y \in \mathbb{R}^{N \times d_y^Y}$ of matrices, with rows indexed by $j \in [\![1, \mathrm{N}]\!]$ (hence for each k and j, y_j^k is a vector in $\mathbb{R}^{d_y^k}$);
4. a vector-valued **symbolic formula** $F(p^1, \dots, p^P, x_i^1, \dots, x_i^X, y_j^1, \dots, y_j^Y) \in \mathbb{R}^{d_{\mathrm{out}}}$;
5. a **reduction** operation such as a sum, max, argmin, log-sum-exp, etc.

Then, a single call to the `KeOps C++` engine allows users to evaluate the expression:

$$a_i \leftarrow \underset{j=1,\dots,\mathrm{N}}{\mathrm{Reduction}} \left[F(p^1,\dots,p^P,x_i^1,\dots,x_i^X,y_j^1,\dots,y_j^Y)\right] \quad \text{for} \quad i=1,\dots,\mathrm{M} \tag{2}$$

efficiently, with a linear memory footprint on GPUs and CPUs. As illustrated in our gallery of tutorials, this level of generality allows `KeOps` to handle off-grid convolutions, k-nearest neighbors classification, k-means clustering and many other tasks.

The `LazyTensor` abstraction. The "`LazyTensor`" wrapper for `NumPy` arrays and `PyTorch` tensors lets users specify computations along the lines of Eq. (2) with a tensor-like interface. For instance, we can specify the Gaussian matrix-vector product of Eq. (1) with:

```
1 from pykeops.torch import LazyTensor  # Wrapper for PyTorch Tensors
2 x_i  = LazyTensor(x[:,None,:])      # (M,D) Tensor -> (M,1,D) Symbolic Tensor
3 y_j  = LazyTensor(y[None,:,:])      # (N,D) Tensor -> (1,N,D) Symbolic Tensor
4 D_ij = ((x_i - y_j)**2).sum(dim=2)  # (M,N,1) Symbolic matrix of squared distances
5 K_ij = (- D_ij / (2 * s**2)).exp()  # (M,N,1) Symbolic Gaussian kernel matrix
6 a    = K_ij @ b  # Genuine torch Tensor. (M,N,1) @ (N,D) = (M,D)
```

In the script above, no computation is performed at lines 4 and 5: lazily, the `KeOps` engine builds up a symbolic formula F encoded as a string attribute of the `LazyTensor` `K_ij`. The lazy chain is only terminated by the virtual matrix product of line 6, a generic reduction that triggers the real computation: no intermediate N-by-M buffer is created in the global device memory.

Note that variable types (i-, j-variable or parameter) are inferred from the shapes of the input tensors at lines 2 and 3: in practice, symbolic tensors are as easy to use as sparse matrices. The `LazyTensor` wrapper turns a dense array into a symbolic matrix whose axes -3 and -2 are understood as "virtual" dimensions; a reduction on these axes is the signal that triggers a call to the `KeOps C++` engine.

As showcased on our website and at the end of this paper, `KeOps` scripts for kernel and geometric applications generally outperform their `Numpy` and `PyTorch` counterparts by several orders of magnitude while keeping a linear memory footprint. `LazyTensors` support a wide range of mathematical operations that mimic the usual interface for `NumPy` arrays and `PyTorch` tensors. They fully support broadcasting and batch dimensions, as well as a decent collection of reduction operations: `.sum()`, `.logsumexp()`, `.max()` and `.min()` but also `.argmin()` or `.argKmin(K=...)` that return the *indices* of the smallest (or K-smallest) elements of the rows of a symbolic tensor.

Inner engine. Internally, `KeOps` creates an optimized `C++` code for every new reduction and formula F that it encounters. Binaries are then compiled and stored on the hard drive for later use: compilation relies on the standard `CUDA` stack (`nvcc`, `gcc` and/or `clang` compilers) and is only performed once per reduction.

Backpropagation. Crucially, `KeOps` supports automatic differentiation up to arbitrary orders of differentiation: a new binary is created automatically for every new partial derivative that is required by the user's computations. This mechanism is fully integrated with the `torch.autograd` engine and lets users "backprop" through `KeOps` calls using the usual `torch.autograd.grad()` and `.backward()` methods.

3. Performance evaluation

KeOps is geared towards computations that fit the mould of Eq. (2). In this specific context, it combines a fully transparent interface with state-of-the-art performance. To illustrate this, we compare KeOps to similar scientific computing libraries – PyTorch, TensorFlow, Halide (Ragan-Kelley et al., 2017) and TVM (Chen et al., 2018) – on a simple benchmark: the Gaussian kernel matrix-vector product of Eq. (1) with an increasing number of points M = N in dimension D = 3. All experiments are performed with float32 precision on a Nvidia RTX 2080 Ti GPU, with the exception of the PyTorch-TPU column that was run in Google Colab; code is available on our repository in the benchmarks folder. Our gallery also includes comparisons with JAX (Bradbury et al., 2018) and domain-specific libraries such as FAISS (Johnson et al., 2017) for K-Nearest Neighbors search.

	PyTorch	PyTorch-TPU	TF-XLA	Halide	TVM	PyKeOps	KeOps++
N = 10k	9 ms	10 ms	13 ms	1.0 ms	3.80 ms	**0.7 ms**	0.4 ms
N = 100k	out of mem	out of mem	89 ms	34.1 ms	36.8 ms	**15.0 ms**	14.6 ms
N = 1M	out of mem	out of mem	out of mem	3.8 s	2.79 s	**1.39 s**	1.38 s
Lines of code	5	5	5	15	17	5	55
Interface	Numpy-like	Numpy-like	Numpy-like	C++	low-level Python	**Numpy-like**	C++

As evidenced by this table, KeOps turns NumPy-like scripts into highly competitive binaries. Going further, it can be neatly interfaced with the iterative linear solvers of the Scipy (Jones et al., 2001) or GPytorch (Gardner et al., 2018) libraries and supports the specification of cluster-wise block-sparsity patterns: this allows users to solve large kernel linear systems efficiently, with applications to geology (Kriging), imaging (splines), statistics (Gaussian processes) and data sciences (kernel methods).

4. Intended Use, Limitations and Future Works

KeOps fills a specific but important niche in machine learning research. Unlike most other compilers for deep learning computations, such as Halide and TVM, it is meant to be used directly by theorists of the machine learning community. Our focus on the simple yet powerful concept of *symbolic matrices* allows us to keep a transparent interface, while being more efficient than the generalist PyTorch and XLA frameworks on a wide range of computations. In future works, we intend to add support for approximation schemes such as the Nyström and FFM methods (Yang et al., 2012; Aussal and Bakry, 2019), beyond the block-sparse truncation rule that is currently supported. These features will be valuable to many researchers in the field, while being out of scope for most deep learning libraries.

The core strength of the KeOps C++ engine is that it optimizes the use of GPU registers for computations that fit the template of Eq. (2). In practice, it is therefore of limited use in situations where a kernel matrix can fit in memory and is re-used a large number of times, or when the evaluation of the formula "$F(p, x_i, y_j)$" takes a significant amount of time. Keeping in mind the motivating example of Eq. (1), we believe that KeOps will be most useful for computations that involve 10k points or more in a space of dimension $D \leqslant 100$. Going forward, our main priority will be to ease the deployment of pre-compiled

KeOps binaries, reduce compilation times to at most a handful of seconds per routine and add support for the newly released CUDA Tensor cores.

Acknowledgements

The three first authors are the project leaders: they contributed equally to the library and its documentation. The authors also thank Alain Trouvé, whose theoretical work in shape analysis was the first motivation for the development of the KeOps engine.

References

M. Abadi, A. Agarwal, et al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL https://www.tensorflow.org/.

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

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. URL http://github.com/google/jax.

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.

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.

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

E. Jones, T. Oliphant, P. Peterson, et al. SciPy: Open source scientific tools for Python, 2001. URL http://www.scipy.org/.

H. Nguyen. *Gpu Gems 3*. Addison-Wesley Professional, first edition, 2007. ISBN 9780321545428.

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. In *NIPS Autodiff Workshop*, 2017.

J. Ragan-Kelley, A. Adams, D. Sharlet, C. Barnes, S. Paris, M. Levoy, S. Amarasinghe, and F. Durand. Halide: Decoupling algorithms from schedules for high-performance image processing. *Commun. ACM*, 61(1):106–115, Dec. 2017. ISSN 0001-0782. doi: 10.1145/3150211. URL https://doi.org/10.1145/3150211.

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.