Kernel Operations on GPU, without memory overflows 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

Editor: ***

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 major 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 engines and GPU backends for tensor computations. Today, thanks to 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 two frameworks keep a focus on tensor-like arrays, with a strong support of convolution and linear algebra routines. They are perfectly suited to the design of convolutional neural networks but also have clear limitations: the *transfer* and *storage* of large quadratic buffers may prevent users of **kernel methods** or **distance matrices** to scale up their method to large data sets.

To work around this problem, a common option is to wrap a custom `Python` operator around a handcrafted piece of `C++/CUDA` code. This method can lead to optimal runtimes, but is reserved to advanced coders and lacks flexibility. In order to lower the barrier of entry to state-of-the-art performances, domain-specific compilers have thus been proposed for image processing and deep learning: see for instance the `Halide` (Ragan-Kelley et al., 2017), `TVM` (Chen et al., 2018) or `Tiramisu` (Baghdadi et al., 2019) libraries. These frameworks are ambitious, but none of them really suits the needs of theorists in the machine learning community: they generally do not support optimal schemes for kernel methods or require an understanding of low-level parallel computing to be setup.

`KeOps` is all about filling this gap to relieve mathematicians from the burden of low-level programming. It is fast, transparent to use and targets a single yet powerful abstraction: semi-symbolic arrays whose entries are given by a mathematical formula. It can be called from the major scripted languages used in the scientific community: `Python` (`Numpy` and `PyTorch`), `Matlab`, and `GNU R`. In this short 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 "`Genred`" engine for *Generic Reductions*. Let us assume that we have at hand:

1. **parameters**: a collection $p^1, p^2, \dots, p^P$ of vectors;
2. ***i*-variables**: a collection $x_i^1, x_i^2, \dots, x_i^X$ of vector sequences, indexed by $i \in [\![1, \mathrm{M}]\!]$;
3. ***j*-variables**: a collection $y_j^1, y_j^2, \dots, y_j^Y$ of vector sequences, indexed by $j \in [\![1, \mathrm{N}]\!]$;
4. a vector-valued **symbolic formula** $F(p^1, \dots, p^P, x_i^1, \dots, x_i^X, y_j^1, \dots, y_j^Y)$;
5. a **reduction** operation such as a sum, max, argmin, log-sum-exp, etc.

Then, a single call to the `Genred` engine allows users to evaluate the expression

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

efficiently, with a linear memory footprint on GPUs. 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 allows users to specify computations along the lines of Eq. (1) with a tensor-like interface. For instance, we can specify a Gaussian matrix-vector product with:

```
from pykeops.torch import LazyTensor  # Wrapper for PyTorch Tensors
q_i  = LazyTensor(q[:, None, :])  # (N,D) Tensor -> (N,1,D) Symbolic Tensor
q_j  = LazyTensor(q[None, :, :])  # (N,D) Tensor -> (1,N,D) Symbolic Tensor
```

```
4 D_ij = ((q_i - q_j) ** 2).sum(dim=2)  # Symbolic squared distances matrix
5 K_ij = (- D_ij / (2 * s ** 2)).exp()  # Symbolic Gaussian kernel matrix
6 v    = K_ij @ p  # Genuine torch Tensor. (N,N) @ (N,D) = (N,D)
```

In the script above, no computation is performed at lines 4 and 5: lazily, the `KeOps` engine simply builds up a symbolic formula F encoded as a string attribute of the `LazyTensor` `K_ij`. The only "genuine" computation here is a call to the `Genred` engine for the virtual matrix-vector product of line 6: no intermediate N-by-N buffer is created in memory. Note that variable types (i-, j-variable or parameter) are inferred from the shapes of the input tensors at lines 2 and 3.

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()`, `.argKmin(K=...)`, etc. This allows users to scale up to large data sets without having to make any kind of approximation: we refer to our tutorials on K-means clustering, K-Nearest Neighbours classification, spectral analysis or Gaussian mixture model estimation.

Backpropagation. Crucially, `KeOps` supports automatic differentiation through a `Grad` operator that can be used on any formula F, recursively if needed. This mechanism is fully integrated with the `torch.autograd` engine: users can seamlessly "backprop" through `KeOps` calls using the usual `torch.autograd.grad()` and `.backward()` methods.

3. Internal Engine

Efficient GPU schemes. The `KeOps` `CUDA` routines are based on parallel implementations of a tiled Map-Reduce scheme: we split reductions into medium-sized blocks to best leverage the **shared memory** of GPU devices. This technique is well-known in GPU programming and detailed in `CUDA` reference guides (Nguyen, 2007, Chapter 31). The originality of `KeOps` resides in its genericity with respect to the formula F, its math-friendly interface and its support of automatic differentiation.

Building formulas, automatic chain-rule. Internally, `KeOps` encodes formulas as recursively templated `C++` classes: each elementary vector-valued operation is defined in a templated `KeOps` class, that specifies an evaluation code to inline in the `CUDA` scheme and a symbolic expression for its gradient. As an example, the element-wise, vector-valued exponential function is encoded with:

```
1 template < class F >
2 struct Exp : UnaryOp<Exp, F> {
3   // dimension of the output: Exp(F) has the same dimension as F
4   static const int DIM = F::DIM
5   // actual computation, to be inlined inside the Cuda device routine
6   static DEVICE INLINE void Operation(TYPE *out, TYPE *in) {
7     #pragma unroll
```

```
    for (int k=0; k<DIM; k++) { out[k] = exp(in[k]); }
  }
  // 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) in math notations
  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 as a sum reduction of the formula `Scal< Exp<Minus<SqDist<X,Y>>>, B>` where `X`, `Y` and `B` are special classes that represent data loaders. This templated engine has two main advantages: first, the code for evaluating the full formula F is built up at compile time, allowing the compiler to optimize the resulting code; second, the chain rule derivation for automatic differentiation can be performed at compile time, using the recursive template mechanics.

4. Performances, Interface with High-Level Libraries

We compare the performances of several computing libraries on a simple benchmark: a Gaussian kernel matrix-vector product with a growing number of points 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.

	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++

`KeOps` is highly competitive for kernel-related computations, with a transparent syntax. Going further, it can be neatly interfaced with the iterative linear solvers of the `Scipy` (Jones et al., 2001) and `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).

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/`.

R. Baghdadi, J. Ray, M. B. Romdhane, E. Del Sozzo, A. Akkas, Y. Zhang, P. Suriana, S. Kamil, and S. Amarasinghe. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO 2019, page 193–205. IEEE Press, 2019. ISBN 9781728114361.

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.

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`.