RKeOps v2: Kernel operations with Symbolic Tensors on the GPU in R

Amélie Vernay[1,*] Benjamin Charlier[1,†] Ghislain Durif[2,‡] Chloe Serre-Combe[1,§]

[1] IMAG, Université de Montpellier, CNRS UMR 5149, Montpellier, France

[2] LBMC, ENS de Lyon, CNRS UMR 5239, Inserm U1293, Université Claude Bernard Lyon 1, Lyon, France

Abstract

We present `RKeOps` version 2, a binder in `R` for the `KeOps` library which implements "kernel operations on the GPU, with autodiff, without memory overflows". The `RKeOps` package allows the user to seamlessly perform fast and memory-efficient kernel computations, based on symbolic matrix operations, such as kernel matrix reductions, convolutions or nearest neighbor search, involving large datasets (up to 10^7 points), on CPU or GPU without any additional development cost. The main contribution of this work is to provide the `LazyTensor` abstraction directly in `R`, allowing to write tensor operations similarly to `R` native syntax for vector and matrix operations, greatly simplifying the user experience.

Keywords: Kernel operations, Matrix reduction, GPU, Symbolic matrix operations, Computational statistics

1 Introduction

In machine learning, statistics and many other fields, practical mathematical computations are performed through dedicated tensor libraries. The data structure behind tensors are usually dense arrays. Although versatile and well supported, this data structure can cause memory allocation issues especially on Graphical Processor Unit (GPU) that do not contains more than few dozens Giga Bytes of Random Access Memory (RAM). Thus, large arrays may not fit in memory or be impossible to process. `KeOps`[1] [1], short for Kernel Operations, is a `C++`/`Python`-based software library that allows to compute symbolic operations on very large arrays whose entries are given by a mathematical formula. These operations are efficiently performed, with a linear memory footprint on GPUs or CPUs. `KeOps` also supports automatic differentiation. The library can be used in `Python` (with `Numpy` or `PyTorch`) and `R` through high-level binders.

We focus here on `RKeOps`[2] version 2, the new `R` front-end for `KeOps`, now relying on `PyKeOps` — the `Python` binder for `KeOps` — through the `reticulate` R package [2]. It has been extended to directly support tensor operations through lazy evaluation. `LazyTensor` (presented below) may be understood as a convenient high level interface to perform computation on symbolic matrices. Instead of writing directly a formula, the user can now use standard `R` syntax with almost no boilerplate code.

*amelie.vernay@umontpellier.fr

†benjamin.charlier@umontpellier.fr

‡ghislain.durif@ens-lyon.fr

§chloe.serre-combe@umontpellier.fr

[1] https://github.com/getkeops/keops

[2] https://github.com/getkeops/keops/tree/main/rkeops

2 Kernel operations as reductions

As data scientists, we often seek to perform reductions (Sum, Max, Argmin, LogSumExp, *etc.*) of large tensors representing interactions between clouds of data points. Common examples of methods using such operations include temporal or spatial convolutions, Gaussian processes, kernel methods. The main drawback of these operations is their high computational complexity, both in time and memory storage requirement.

Consider the following example of a Gaussian convolution: given some source points $\mathbf{y}_1, \dots, \mathbf{y}_j, \dots, \mathbf{y}_N \in \mathbb{R}^D$ with associated weights $b_1, \dots, b_j, \dots, b_N \in \mathbb{R}$, and target points $\mathbf{x}_1, \dots, \mathbf{x}_i, \dots, \mathbf{x}_M \in \mathbb{R}^D$, we seek to evaluate the Gaussian kernel product

$$a_i \leftarrow \sum_{j=1}^{N} k(\mathbf{x}_i, \mathbf{y}_j)\, b_j, \quad i = 1, \dots, M \tag{1}$$

where $k(\mathbf{x}_i, \mathbf{y}_j) = \exp\left(-\|\mathbf{x}_i - \mathbf{y}_j\|_2^2\right)$ is a Gaussian kernel. A common practice is to compute all the elements $(k(\mathbf{x}_i, \mathbf{y}_j))_{i,j}$ and to store them in a dense $M \times N$ matrix — usually called the kernel matrix — encoded as an explicit array. The reduction is then performed as a matrix-vector product using some dedicated linear algebra routine. Unfortunately, with a quadratic $\mathcal{O}(MN)$ time complexity and memory usage, in high dimension it becomes impossible to store the kernel matrix and to perform the sum reduction step.

3 Symbolic evaluation with RKeOps

3.1 Symbolic evaluations

Most of the time, when considering reductions involving interaction steps, we are more interested in the evaluation of an operator on a given vector than in the explicit matrix evaluation of that operator. This is the idea behind symbolic evaluation: providing that the entries of a matrix can be expressed as a mathematical formula, intermediate computations can be written as a sequence of symbolic operations that are not directly evaluated. The real evaluation is only made for the final computation, avoiding the computation and storage of large matrices (of intermediate results).

3.2 The LazyTensor abstraction

`RKeOps` now provides a new data structure, called `LazyTensor`, to encode numerical arrays through the combination of a symbolic mathematical formula and a list of data arrays. The `LazyTensor` objects can be used to implement efficient algorithms on objects that are easy to define but impossible to store in memory, through an array-like interface in an `R`-friendly syntax. For example, let `x`, `y` be both `R matrix` encoding 3D point clouds with respectively $M = N = 10^6$ samples each, and let `b` be another `R matrix` storing a column vector of the same size 10^6. Then, one can easily compute the reduction given by Equation (1) with `RKeOps`:

```
library(rkeops)
x_i  <- LazyTensor(x, "i")           # symbolic variables wrapping 3D data x
y_j  <- LazyTensor(y, "j")           # symbolic variables wrapping 3D data y
b_j  <- LazyTensor(b, "j")           # symbolic variables wrapping 1D data b
K_ij <- exp(- sum((x_i - y_j)^2))    # symbolic Gaussian kernel (of dim M x N)
a_i  <- sum(K_ij * b_j, index = "j") # actual R matrix with the results
```

In the previous example, we first define symbolic objects representing an arbitrary row of `x` (resp. `y` and `b`), indexed by the letter `"i"` (resp. `"j"`). When `K_ij` is defined, **no actual computation is performed**: `RKeOps` builds a symbolic formula encoded as a character string added to the attributes of the `LazyTensor`. `K_ij` represents a $10^6 \times 10^6$ tensor but is never stored in memory. The sum reduction on the last line triggers the real computation. The corresponding `C++` or `Cuda` (for GPU[3] computing) code is silently compiled, executed and the result is stored in the new `R matrix` called `a_i`.

[3] KeOps works with *Nvidia* GPUs and `Cuda`.

The `LazyTensor` objects support a wide range of mathematical operations and reductions[4]. It is also possible to work with complex number and symbolic complex operation by defining `ComplexLazyTensor` objects. These are a generalization of the `LazyTensor` objects: the real and imaginary parts are dissociated and stored in two contiguous columns, but the initial inner dimension is preserved.

4 Examples

In this section, we give some code snippets to show the syntax of `RKeOps` on two simple, yet useful, examples.

4.1 Gaussian convolution

We give here a full version of the standard Gaussian convolution (defined in Equation 1) in $\mathbb{R}^{15}$ with bandwidth$= 1$ and weight vectors $\mathbf{b}_j$ also in $\mathbb{R}^{15}$, using 10^6 data points:

```
library(rkeops)
N <- 10^6; D <- 15               # dimensions
x <- matrix(rnorm(N*D), N, D)    # data: x of dim (N,D)
y <- matrix(rnorm(N*D), N, D)    # data: y of dim (N,D)
b <- matrix(rnorm(N*D), N, D)    # data: b of dim (N,D)
lambda <- 1.0                    # parameter: lambda

x_i <- Vi(x)                     # init x LazyTensor ("i" -> rows 1:N)
y_j <- Vj(y)                     # init y LazyTensor ("j" -> rows 1:N)
b_j <- Vj(b)                     # init b LazyTensor ("j" -> rows 1:N)
Pm_lambda <- Pm(lambda)          # init lambda LazyTensor (Parameter)

res <- sum(exp(-Pm_lambda * sqdist(x_i, y_j)) * b_j, 'j')
```

4.2 Nearest Neighbor search

We use the `LazyTensor` syntax to compute K-nearest-neighbor search between two clouds of 3-dimensional data points of size 10^6 using the ℓ_1 distance (with $K = 10$):

```
library(rkeops)
N <- 10^6; D <- 3                # dimensionss
x <- matrix(rnorm(N*D), N, D)    # data: x of dim (N,D)
y <- matrix(rnorm(N*D), N, D)    # data: y of dim (N,D)

x_i <- Vi(x)                     # init x LazyTensor ("i" -> rows 1:N)
y_j <- Vj(y)                     # init y LazyTensor ("j" -> rows 1:N)

K <- 10                          # number of nearest neighbor

res <- argKmin(sum(abs(x_i - y_j)), K, 'j')
```

In the previous code, each row in the `res` array (of dimension `N` $\times$ `K`) contains the row index (i.e. `"j"` index) of the K nearest neighbors in `y` for each row `"i"` in `x`.

5 Benchmark

We show here some benchmarking results about the computation of the Gaussian convolution (previously introduced in subsection 4.1) computed with `RKeOps` and computed using R base code. Averaged computation

[4]The full list may be found at https://www.kernel-operations.io/keops/api/math-operations.html

times across 100 repetitions are presented in Figure 1 for different numbers of points in the data:

$$N = 100, 1000, 5000, 10000, 50000, 100000, 200000.$$

`RKeOps` computations were made both on CPU and GPU, using 32bits (single precision) and 64bits (double precision) floating point numbers. CPU computations were made using 16 cores of an *Intel Xeon Gold 6142* processor, whereas GPU computations were made using an *Nvidia A10* GPU chip.

R base code computation times are between 10 and 1000 times slower than `RKeOps` computations (and not available for $N > 10000$ because too long). GPU computations are around 10 to 100 times faster than CPU computations with RKeOps. Single precision computing is faster on GPU compared to double precision computing but not on CPU[5].

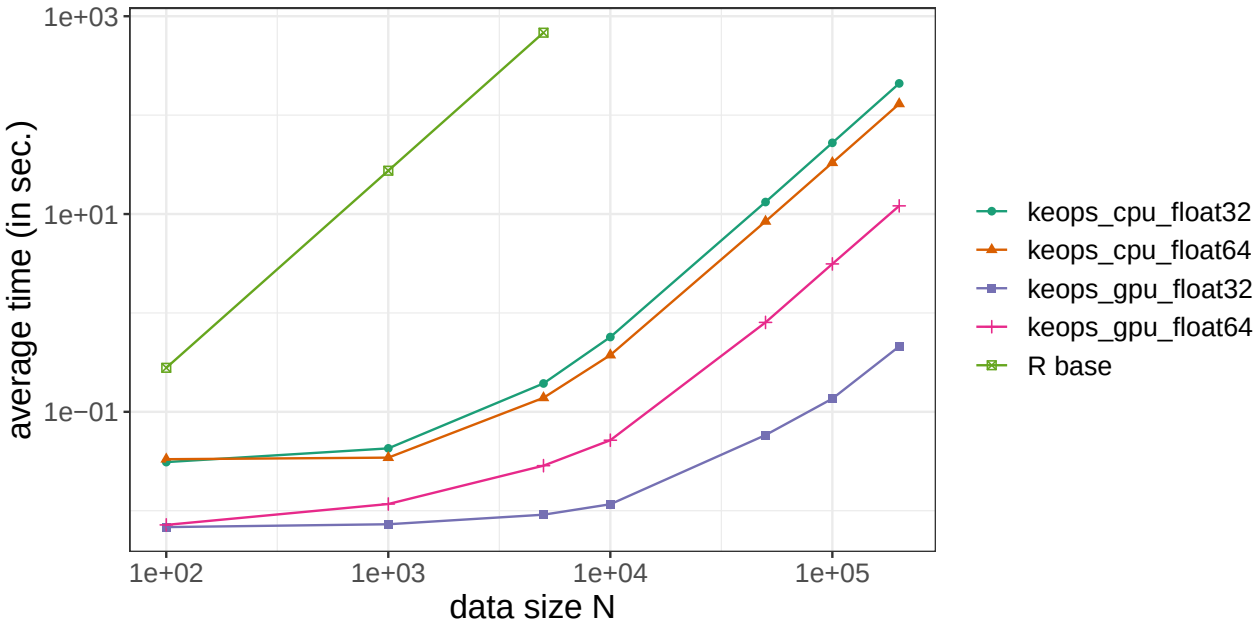

Figure 1: Gaussian convolution benchmark results.

Additional examples and benchmarks will be showcased during the presentation.

References

1. **Charlier**, **B**, **Feydy**, **J**, **Glaunès**, **J A**, **Collin**, **F-D**, and **Durif**, **G** 2021 Kernel operations on the GPU, with autodiff, without memory overflows. *Journal of Machine Learning Research*, 22(74): 1–6. URL http://jmlr.org/papers/v22/20-275.html

2. **Ushey**, **K**, **Allaire**, **J**, and **Tang**, **Y** 2023 *Reticulate: Interface to 'python'*.

[5]GPU cores are optimized to be faster with single-precision computing than with double-precision, which is not always the case for CPU cores.