



# Seamless Kernel Operations on GPU without memory overflows

---

B. Charlier<sup>1</sup> J. Feydy<sup>2</sup> J. Glaunès<sup>3</sup> **G. Durif**<sup>1</sup> F.-D. Collin<sup>1</sup>

*July 7th 2020 – useR! 2020*

<sup>1</sup>IMAG - CNRS - Univ Montpellier, France, <sup>2</sup>Imperial College, London, UK,

<sup>3</sup>MAP5 - Univ Paris Descartes, France

<http://www.kernel-operations.io/>  
[ghislain.durif@umontpellier.fr](mailto:ghislain.durif@umontpellier.fr)

# Outline

1. Introduction
2. Kernel operations and reductions
3. Computation on GPU
4. Implementation
5. Using KeOps

# Introduction

---

# What is KeOps?

`http://www.kernel-operations.io/`

KeOps = “Kernel Operations”

 **RKeOps** 

RKeOps = R package interfacing KeOps library

# What KeOps can do?

Compute **generic reductions** of very large arrays

*e.g.* row-wise or column-wise matrix sum

$$\sum_{i=1}^M a_{ij} \quad \text{or} \quad \sum_{j=1}^N a_{ij}$$

(for some large matrix  $\mathbf{A} = [a_{ij}] \in \mathbb{R}^{M \times N}$ )

# What KeOps can do?

$$\begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & a_{ij} & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix}_{M \times N}$$

↓

$$\left[ \dots \dots \sum_{i=1}^M a_{ij} \dots \dots \right]_{1 \times N}$$

# What KeOps can do?

$$\begin{bmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & a_{ij} & \times & \times \\ \times & \times & \times & \times & \times \\ \times & \times & \times & \times & \times \end{bmatrix}_{M \times N} \rightarrow \begin{bmatrix} \vdots \\ \vdots \\ \sum_{j=1}^N a_{ij} \\ \vdots \\ \vdots \end{bmatrix}_{M \times 1}$$

# What KeOps can do?

Compute **kernel reduction**

$$\text{e.g. } \sum_{i=1}^M K(\mathbf{x}_i, \mathbf{y}_j) \quad \text{or} \quad \sum_{j=1}^N K(\mathbf{x}_i, \mathbf{y}_j)$$

and the associated **gradients**

(for a kernel function  $K$  and some data vectors  $\mathbf{x}_i, \mathbf{y}_j \in \mathbb{R}^D$ )

→ **Intuitively:**  $[K(\mathbf{x}_i, \mathbf{y}_j)] \in \mathbb{R}^{M \times N}$  = matrix whose elements are given by a **formula**



# What KeOps can do?

→ manage **large dimensions**

- even larger than GPU memory
- $M$  and  $N \approx 10^4, 10^5, 10^6$

→ **fast computation on GPU without memory overflow**

# Kernels in Statistics and Learning

- Kernel density estimation
- Classification/Regression: SVM, K-NN, etc...
- Kernel embeddings to compare distributions
- Interpolation and Kriging
- Optimal Transport

# Motivations

GPU user-friendly computing?

In R: only a few solution for specific tasks

See <https://CRAN.R-project.org/view=HighPerformanceComputing> (section GPUs)

# Motivations

Over the past 5 years: GPU computing development effort **oriented toward deep learning**

→ e.g. PyTorch or TensorFlow provide **GPU** implementation of common operations, together with **automatic differentiation**.

# Motivations

GPU computing can be used for **general purpose computations** and not only neural networks

→ Generic codes to use GPU computing require **low-level tools** (CUDA, OpenCL)

# Motivations

**Needs:** provide an effortless tool for GPU computing

**Application:** statistics, machine learning and more...

# Kernel operations and reductions

---

# Kernel operator

Considering some data vector  $\mathbf{x}_i$  and  $\mathbf{y}_j$  in  $\mathbb{R}^D$

(Intuitively)

a **kernel** function = an application  $K : \mathbb{R}^D \times \mathbb{R}^D \rightarrow \mathbb{R}$

$$(\mathbf{x}_i, \mathbf{y}_i) \mapsto K(\mathbf{x}_i, \mathbf{y}_i)$$

corresponding to a **scalar product** between  $\mathbf{x}_i$  and  $\mathbf{y}_j$  in a different space than usual  $\mathbb{R}^D$



# Kernel operator

Considering some data vector  $\mathbf{x}_i$  and  $\mathbf{y}_j$  in  $\mathbb{R}^D$

(Very intuitively)

a **kernel** function  $\approx$  “*similarity measure*”

between  $\mathbf{x}_i$  and  $\mathbf{y}_j$

(different from Euclidean distance)

# Example

Linear kernel

$$K(\mathbf{x}_i, \mathbf{y}_j) = \langle \mathbf{x}_i, \mathbf{y}_j \rangle = \mathbf{x}_i^T \mathbf{y}_j = \sum_{k=1}^D x_{ik} y_{jk}$$

Gaussian kernel

$$K(\mathbf{x}_i, \mathbf{y}_j) = \exp \left( -\frac{1}{2\sigma^2} \|\mathbf{x}_i - \mathbf{y}_j\|_2^2 \right)$$

# Kernel reduction

- **Row-wise or column-wise reduction** on the matrix  $\mathbf{K} = [K(\mathbf{x}_i, \mathbf{y}_j)] \in \mathbb{R}^{M \times N}$
- And more complex operations

Example: 
$$\sum_{i=1}^M K_1(\mathbf{x}_i, \mathbf{y}_j) K_2(\mathbf{u}_i, \mathbf{v}_j) \langle \boldsymbol{\alpha}_i; \boldsymbol{\beta}_j \rangle$$

for some kernel  $K_1$  and  $K_2$ , and some  $D$ -vectors  $(\mathbf{x}_i)_i, (\mathbf{u}_i)_i$   
 $(\boldsymbol{\alpha}_i)_i \in \mathbb{R}^{M \times D}$  and  $(\mathbf{y}_j)_j, (\mathbf{v}_j)_j, (\boldsymbol{\beta}_j)_j \in \mathbb{R}^{N \times D}$

# Kernel reduction

- **Row-wise or column-wise reduction** on the matrix  $\mathbf{K} = \left[ K(\mathbf{x}_i, \mathbf{y}_j) \right] \in \mathbb{R}^{M \times N}$
- And more complex operations

Example: 
$$\sum_{j=1}^N K_1(\mathbf{x}_i, \mathbf{y}_j) K_2(\mathbf{u}_i, \mathbf{v}_j) \langle \boldsymbol{\alpha}_i; \boldsymbol{\beta}_j \rangle$$

for some kernel  $K_1$  and  $K_2$ , and some  $D$ -vectors  $(\mathbf{x}_i)_i, (\mathbf{u}_i)_i$   
 $(\boldsymbol{\alpha}_i)_i \in \mathbb{R}^{M \times D}$  and  $(\mathbf{y}_j)_j, (\mathbf{v}_j)_j, (\boldsymbol{\beta}_j)_j \in \mathbb{R}^{N \times D}$

# Why GPU computing?

Matrix/kernel reduction = combination of generic matrix operations

→ GPU are good for matrix computations

# Computation on GPU

---

# GPUs

**PLUS:** thousands of computing units

→ fast with heavily parallelized computations

**MINUS:** relatively small memory

(compared to the number of computing units)

→ issue to process large data

# Challenge

Matrix  $\mathbf{K} = [K(\mathbf{x}_i, \mathbf{y}_j)] \in \mathbb{R}^{M \times N}$  is very large

$(M, N \approx 10^4, 10^5, 10^6)$

→ store it in memory? **NO!**

→ how to iterate through rows/columns?



# Memory management on GPU

Data initially stored on the host (in RAM)

→ should be transferred to the device (GPU) for computations (**bottleneck**)

Different kinds of memory inside the GPU

→ local (smaller) vs shared (bigger) memory

# Memory management on GPU

## Smart use of the shared memory

- less transfer between device and host
- key to provide an efficient code in term of computational time

## Tiling implementation

skip example

# Tiled implementation

Computations are divided into steps

Data are divided into blocks (called tiles)

Use of accumulators to combine intermediate results from each step on each block

# Tiled implementation

## Objective for massive parallel computing:

- Shared memory stores data commonly used by all threads during a computation step  
→ reduce transfers between host and GPU
- Size of data only used by a single thread during a step is reduced (in local memory)  
→ data locality

# Example: matrix product

$$\mathbf{A} = [a_{ij}] \in \mathbb{R}^{M \times N} \text{ and } \mathbf{B} = [b_{jk}] \in \mathbb{R}^{N \times P}$$

$$\mathbf{C} = \mathbf{A}\mathbf{B}$$

$$\mathbf{C} = [c_{ik}] \in \mathbb{R}^{M \times P} \text{ and } c_{ik} = \sum_j a_{ij} b_{jk} = \langle \mathbf{a}_{i\cdot}, \mathbf{b}_{\cdot k} \rangle$$

# Example: matrix product

$$\begin{bmatrix} \cdot & \cdot & \cdot \\ \hline & \mathbf{a}_{j\cdot} & \\ \hline \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \end{bmatrix}_{M \times N} \times \begin{bmatrix} \cdot & | & | & \cdot \\ \cdot & | & \mathbf{b}_{\cdot k} & \cdot \\ \cdot & | & | & \cdot \end{bmatrix}_{N \times P}$$

$$= \begin{bmatrix} \cdot & \cdot & \cdot \\ \cdot & \sum_j a_{ij} b_{jk} & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \end{bmatrix}_{M \times P}$$

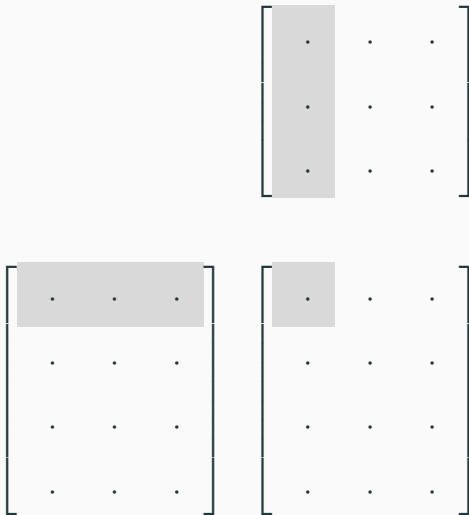
# Example: matrix product

$$\left[ \begin{array}{c|c|c} \cdot & & \cdot \\ \cdot & \mathbf{b}_{\cdot k} & \cdot \\ \cdot & & \cdot \end{array} \right]_{N \times P}$$

$$\left[ \begin{array}{c|c|c} \cdot & \cdot & \cdot \\ \hline & \mathbf{a}_i & \\ \hline \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \end{array} \right]_{M \times N}$$

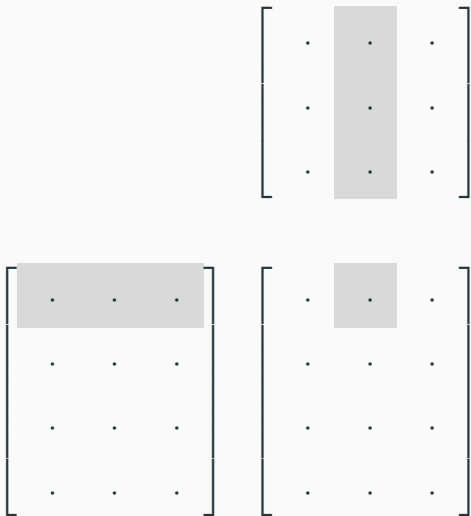
$$\left[ \begin{array}{c|c|c} \cdot & \cdot & \cdot \\ \cdot & \sum_j a_{ij} b_{jk} & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \end{array} \right]_{M \times P}$$

# Iterating through rows and columns

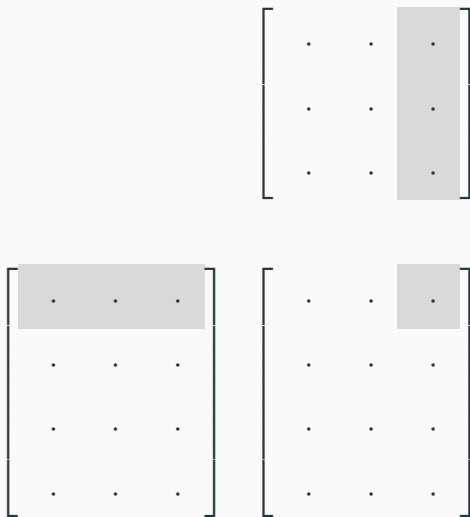




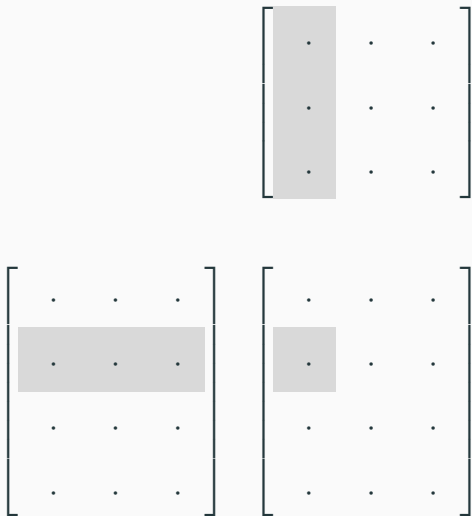
# Iterating through rows and columns



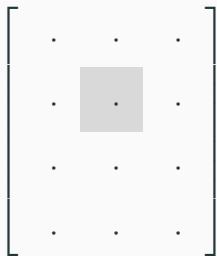
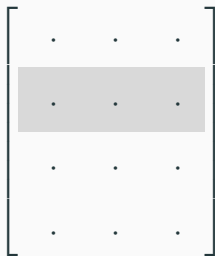
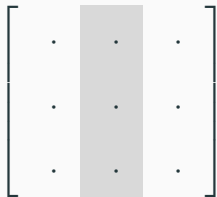
# Iterating through rows and columns



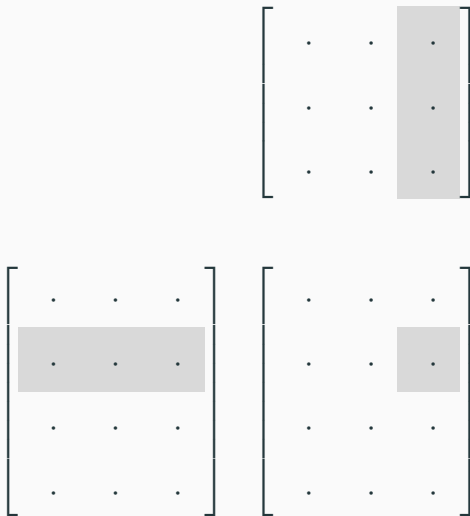
# Iterating through rows and columns



# Iterating through rows and columns



# Iterating through rows and columns



# Iterating through rows and columns

and so on...

# Parallel matrix product

Thread 1 and thread 2 **work in parallel**

$$\begin{bmatrix} \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \end{bmatrix}$$

$$\begin{array}{l} \text{thread 1 computes } T_1 \rightarrow \\ \text{thread 2 computes } T_2 \rightarrow \end{array} \begin{bmatrix} \square & \square & \square \\ \triangle & \triangle & \triangle \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \end{bmatrix} \begin{bmatrix} T_1 & \cdot & \cdot \\ T_2 & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \end{bmatrix}$$

 = shared data between threads



$\square$  = data only used by thread 1  
 $\triangle$  = data only used by thread 2

# Parallel matrix product

Thread 1 and thread 2 **work in parallel**



 = shared data between threads

 = data only used by thread 1  
 = data only used by thread 2



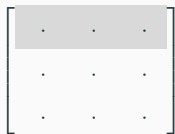
# Parallel matrix product

**Potential issues** if dimension  $N$  is large (nb. of columns in matrix **A**)

- parallel threads are asynchronous and have to wait each other before updating shared memory (= using next column of matrix **B**)
- rows of **A** are too large to fit into local memory used by each thread, hence numerous memory transfer

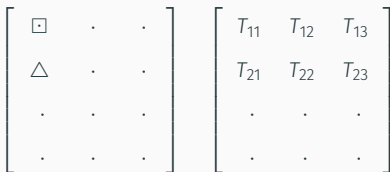
# Tiled implementation

Thread 1 and thread 2 **work in parallel**



thread 1 accumulates over  $T_{1k}$ 's  $\rightarrow$

thread 2 accumulates over  $T_{2k}$ 's  $\rightarrow$



 = shared "tile" between threads

$\square$  = "tile" only used by thread 1

$\triangle$  = "tile" only used by thread 2

# Tiled implementation

Thread 1 and thread 2 **work in parallel**

$$\begin{bmatrix} \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \end{bmatrix}$$

thread 1 accumulates over  $T_{1k}$ 's  $\rightarrow$

thread 2 accumulates over  $T_{2k}$ 's  $\rightarrow$

$$\begin{bmatrix} \cdot & \square & \cdot \\ \cdot & \triangle & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \end{bmatrix} \quad \begin{bmatrix} T_{11} & T_{12} & T_{13} \\ T_{21} & T_{22} & T_{23} \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \end{bmatrix}$$

 = shared "tile" between threads

$\square$  = "tile" only used by thread 1

$\triangle$  = "tile" only used by thread 2

# Tiled implementation

Thread 1 and thread 2 **work in parallel**



thread 1 accumulates over  $T_{1k}$ 's  $\rightarrow$

thread 2 accumulates over  $T_{2k}$ 's  $\rightarrow$

$$\begin{bmatrix} \cdot & \cdot & \square \\ \cdot & \cdot & \triangle \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \end{bmatrix} \quad \begin{bmatrix} T_{11} & T_{12} & T_{13} \\ T_{21} & T_{22} & T_{23} \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \end{bmatrix}$$

 = shared "tile" between threads

$\square$  = "tile" only used by thread 1

$\triangle$  = "tile" only used by thread 2

# Tiled implementation

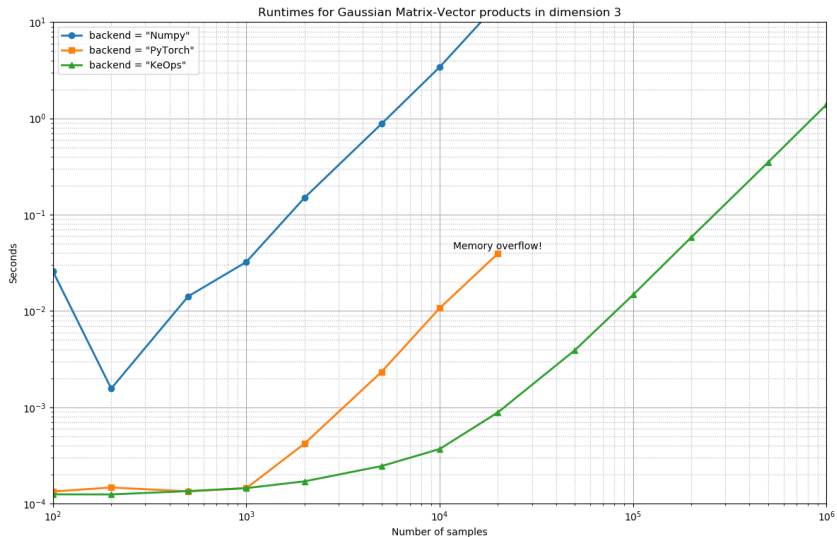
- Tasks (scanning rows  $\mathbf{a}_j$ .) divided into tiles
- All threads use the shared memory within a block
  - a single memory transfer of each tile in  $\mathbf{B}$  for all threads
- Accumulation (addition of the intermediate results) when scanning tiles across  $\mathbf{A}$

# Benchmark I

Runtime comparison for Gaussian matrix-vector product on GPU with different data size

- For small sample sizes (up to  $10^3$ ): similar performance for KeOps and PyTorch
- For larger sample sizes ( $> 10^3$ ): **KeOps outperforms PyTorch**
- Memory overflow with PyTorch on large sample
- **KeOps** able to process data **larger than GPU memory**

# Benchmark I



# Benchmark II

Runtime comparison between

- standard R
- RKeOps
- PyKeOps
- PyTorch



# Benchmark II

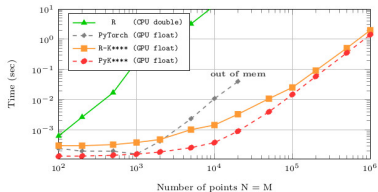
## Different tasks

- Matrix-vector product with Gaussian kernel
- Solving Gaussian kernel linear system
- 10-iterations of  $K$ -means algorithm
- Exact  $K$ -nearest neighbor search

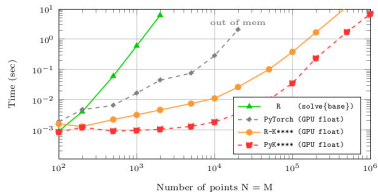
# Benchmark II

- *RKeOps* outperforms standard *R* and *PyTorch* with very large data sizes  
(except for *K*-means algorithm because of bad implementation)
- Memory overflow with *PyTorch* on large sample
- ***RKeOps*** able to process data **larger than GPU memory**

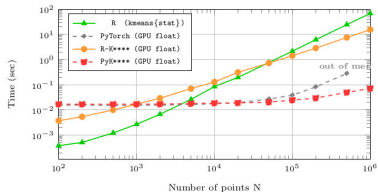
# Benchmark II



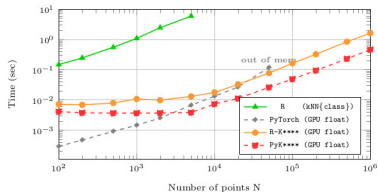
(a) Matrix-vector products with  $N$ -by- $N$  Gaussian kernel matrices built from point clouds in dimension  $D = 3$ .



(b) Solving an  $N$ -by- $N$  Gaussian kernel linear system with ridge regularization (constant diagonal weights).



(c) 10 iterations of K-means (Lloyd's algorithm) with  $N$  points in dimension  $D = 10$  and  $K = \lfloor \sqrt{N} \rfloor$  clusters.



(d) Exact ( $K = 10$ )-nearest neighbor search: 10k queries in dimension  $D = 100$  with a database of  $N$  samples.

# Implementation

---

# Coding generic formulas with KeOps

**Mathematical formula** with two vectors  $\mathbf{x}, \mathbf{y} \in \mathbb{R}^D$ :

$$(\mathbf{x}, \mathbf{y}) \mapsto \exp(\langle \mathbf{x}, \mathbf{y} \rangle)$$

→ what I want to compute

A formula  $F$  in KeOps is first **encoded as a string** using combinations of elementary operations

"Exp(Scalprod(x, y))"

→ what I have to write

# Under the hood

The formula  $F$  is expanded internally in the C++ code **using templates**:

$$F = \text{Exp} < \text{Scalprod} < X, Y > >$$

A formula is an instantiation of a variadic recursively defined templated class

→ KeOps compiles your operator on the fly to **compute on GPU**

# Combining elementary operations

KeOps proposes a wide range of elementary operations

- **Simple vector operations:** scalar product, norm, distance, normalization, vector/vector element-wise operation (+, -, \*, /), etc.
- **Elementary  $\mathbb{R} \rightarrow \mathbb{R}$  functions:** exp, log, inverse, abs, pow, sqrt, sin, cos, etc.

# Combining elementary operations

KeOps proposes a wide range of elementary operations

- **Simple matrix operations:** matrix product, etc.
- **Matrix reduction:** sum, min, max, argmin, argmax, etc.



# Combining elementary operations

KeOps proposes a wide range of elementary operations

→ a formula = **a combination of these operations**

# Using KeOps

---

- Complete documentation
- Installation instructions (available on CRAN)

```
install.packages("rkeops")
```

- Examples

# KeOps stack

<https://github.com/getkeops/keops>

Open source (MIT licence)

# Example in R: single Gaussian convolution

We want to compute

$$\gamma_i = \sum_{j=1}^N \exp\left(-s \|\mathbf{x}_i - \mathbf{y}_j\|_2^2\right) \mathbf{b}_j$$

with  $s \in \mathbb{R}$ ,

$$[\mathbf{x}_i]_{i=1,\dots,M} \in \mathbb{R}^{M \times 3}, [\mathbf{y}_j]_{j=1,\dots,N} \in \mathbb{R}^{N \times 3}$$

$$\text{and } [\mathbf{b}_j]_{j=1,\dots,N} \in \mathbb{R}^{N \times 6}$$

# Example in R: single Gaussian convolution

## Compilation on the fly of the operator

```
library(rkeops)

# implementation of a convolution with a Gaussian kernel
formula = "Sum_Reduction(Exp(-s * SqNorm2(x - y)) * b, 0)"

# definition of input arguments
args = c("x = Vi(3)",      # vector indexed by i (of dim 3)
         "y = Vj(3)",      # vector indexed by j (of dim 3)
         "b = Vj(6)",      # vector indexed by j (of dim 6)
         "s = Pm(1)")      # parameter (scalar)

# compilation
op <- keops_kernel(formula, args)
```

# Example in R: single Gaussian convolution

Some data

```
# data and parameter values
nx <- 100
ny <- 150
X <- matrix(runif(nx*3), nrow=nx) # matrix 100 x 3
Y <- matrix(runif(ny*3), nrow=ny) # matrix 150 x 3
B <- matrix(runif(ny*6), nrow=ny) # matrix 150 x 6
s <- 0.2
```

# Example in R: single Gaussian convolution

Run computations

```
# run computations on GPU (optional)
use_gpu()

# computation
# (order of input list similar to `args`)
res <- op(list(X, Y, B, s))
```



# Example in R: single Gaussian convolution

## Gradient

```
# compile gradient regarding 'x' variable  
grad_op <- keops_grad(op, var="x")
```

# Conclusion

---

# Take-home message: KeOps

Seamless Kernel Operations...

→ write formulas with simple matrix operations in R

...on GPU...

→ fast computations

...with auto-differentiation...

→ automatic gradient computation

...and without memory overflows

→ tiling implementation

# In the future?

## Lazy evaluation in R (example below with PyKeOps)

```
# Symbolic representation of data
x_i = LazyTensor( x[:,None,:] ) # shape (1e6, 1, 3)
y_j = LazyTensor( y[None,:,:] ) # shape ( 1, 2e6,3)

# Symbolic (1e6,2e6,1) matrix of squared distances
D_ij = ((x_i - y_j)**2).sum(dim=2)

# Symbolic (1e6,2e6,1) Gaussian kernel matrix
K_ij = (- D_ij).exp()

## Result (computations on GPU are done here)
a_i = K_ij.sum(dim=1) # shape (1e6, 1)
```

# Thank you for you attention

<https://arxiv.org/abs/2004.11127>  
(pending publication)

<http://www.kernel-operations.io/>

<https://github.com/getkeops/keops>