RKeOps

Kernel operations with Symbolic Tensors on the GPU in R

Ghislain Durif[2] & Benjamin Charlier[1] & Chloé Serre-Combe[1] & Amélie Vernay[1,*]

June 21st 2023 – Rencontres R 2023

[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*
[*] amelie.vernay@umontpellier.fr

A motivating example

For $i = 1, \ldots, M$ we want to compute the **reduction**

$$a_i = \sum_{j=1}^{N} K(\mathrm{x}_i, \mathrm{y}_j) b_j, \tag{1}$$

with

- source points $\mathrm{y}_1, \ldots, \mathrm{y}_N \in \mathbb{R}^D$ with associated weights $b_1, \ldots, b_N \in \mathbb{R}$
- target points $\mathrm{x}_1 \ldots, \mathrm{x}_M \in \mathbb{R}^D$
- a Gaussian kernel $K(\mathrm{x}_i, \mathrm{y}_j) = \exp\left(-\|\mathrm{x}_i - \mathrm{y}_j\|_2^2\right)$

A motivating example

For $i = 1, \ldots, M$ we want to compute the **reduction**

$$a_i = \sum_{j=1}^{N} K(\mathbf{x}_i, \mathbf{y}_j) b_j, \tag{1}$$

with

- source points $\mathbf{y}_1, \ldots, \mathbf{y}_N \in \mathbb{R}^D$ with associated weights $b_1, \ldots, b_N \in \mathbb{R}$
- target points $\mathbf{x}_1 \ldots, \mathbf{x}_M \in \mathbb{R}^D$
- a Gaussian kernel $K(\mathbf{x}_i, \mathbf{y}_j) = \exp\left(-\|\mathbf{x}_i - \mathbf{y}_j\|_2^2\right)$

Limitations of basic routines

- computation of all the elements $(K(\mathbf{x}_i, \mathbf{y}_j))_{i,j}$ $\rightarrow \mathcal{O}(MN)$ time complexity
- storage as a dense $M \times N$ matrix $\rightarrow \mathcal{O}(MN)$ memory usage

A motivating example

For $i = 1, \ldots, M$ we want to compute the **reduction**

$$a_i = \sum_{j=1}^{N} K(\mathrm{x}_i, \mathrm{y}_j) b_j, \tag{1}$$

with

- source points $\mathrm{y}_1, \ldots, \mathrm{y}_N \in \mathbb{R}^D$ with associated weights $b_1, \ldots, b_N \in \mathbb{R}$
- target points $\mathrm{x}_1 \ldots, \mathrm{x}_M \in \mathbb{R}^D$
- a Gaussian kernel $K(\mathrm{x}_i, \mathrm{y}_j) = \exp\left(-\|\mathrm{x}_i - \mathrm{y}_j\|_2^2\right)$

Limitations of basic routines

- computation of all the elements $(K(\mathrm{x}_i, \mathrm{y}_j))_{i,j}$ $\rightarrow \mathcal{O}(MN)$ time complexity
- storage as a dense $M \times N$ matrix $\rightarrow \mathcal{O}(MN)$ memory usage

$\rightarrow$ Impossible in high dimension! $\leftarrow$

KeOps: Kernel Operations

RKeOps: Kernel Operations in R

RKeOps: Kernel Operations in R

- Perform **fast reductions** of very large arrays (M, $N \simeq 10^6$),

RKeOps: Kernel Operations in R

- Perform **fast reductions** of very large arrays ($M, N \simeq 10^6$),
- with **effortless computation on GPU** without memory overflow,

RKeOps: Kernel Operations in R

- Perform **fast reductions** of very large arrays ($M, N \simeq 10^6$),
- with **effortless computation on GPU** without memory overflow,
- with **automatic differentiation** up to arbitrary orders.

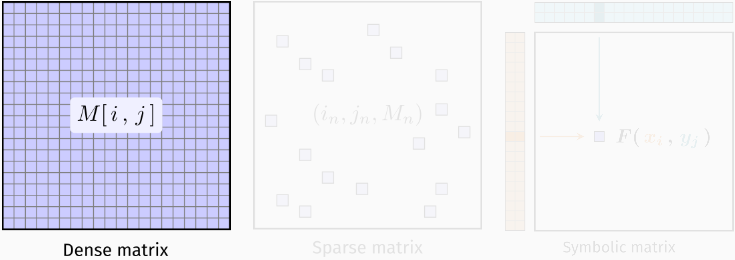
RKeOps: Kernel Operations in R

- Perform **fast reductions** of very large arrays ($M, N \simeq 10^6$),
- with **effortless computation on GPU** without memory overflow,
- with **automatic differentiation** up to arbitrary orders.

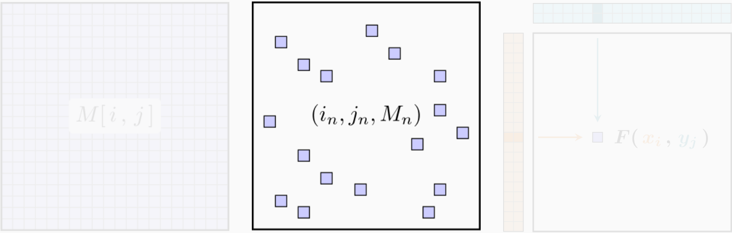
In short, RKeOps offers gains on runtime and memory usage by performing on-the-fly compilation with **symbolic matrices**.

$\rightarrow$ https://www.kernel-operations.io $\leftarrow$

Symbolic evaluation with RKeOps LazyTensors

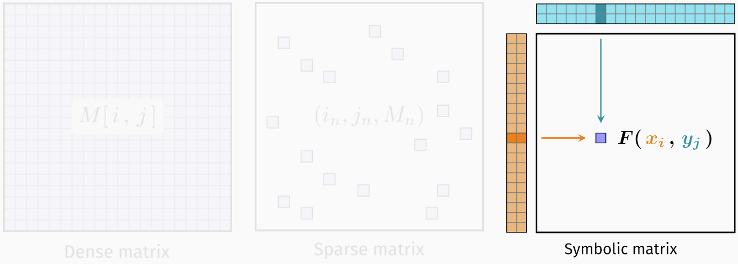


Symbolic evaluation with RKeOps `LazyTensors`



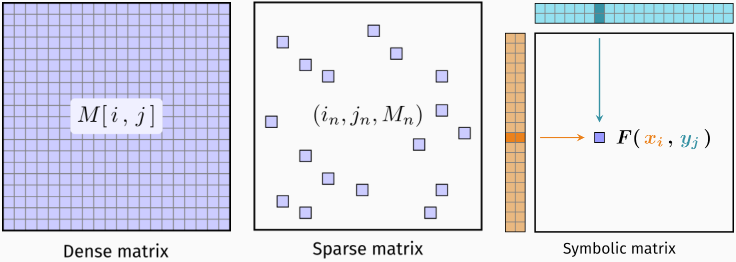
Dense matrix — Sparse matrix — Symbolic matrix

Symbolic evaluation with RKeOps LazyTensors



Symbolic matrices: Matrices whose entries are given by a mathematical formula.

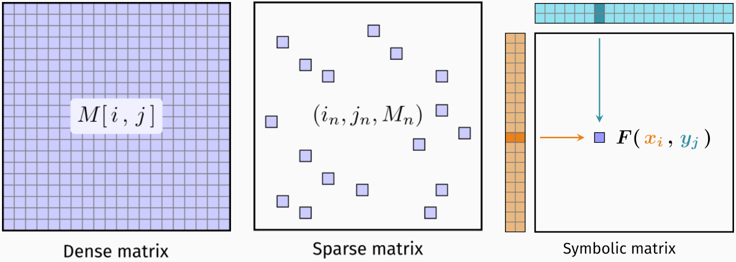
Symbolic evaluation with RKeOps LazyTensors



Symbolic matrices: Matrices whose entries are given by a mathematical formula.

`RKeOps LazyTensors`: Wrappers around R data arrays that embody symbolic matrices.

Symbolic evaluation with RKeOps LazyTensors



Symbolic matrices: Matrices whose entries are given by a mathematical formula.

`RKeOps` `LazyTensors`: Wrappers around R data arrays that embody symbolic matrices.

1. import `RKeOps`
2. create `LazyTensors` with your data
3. perform *any* kinds of reduction using friendly R native syntax

Symbolic evaluation with RKeOps LazyTensors

Let us perform the Gaussian reduction (1) with RKeOps: $\left[\sum_{j=1}^{N} k(\mathbf{x}_i, \mathbf{y}_j) b_j\right]_{i=1}^{M}$

```r
# Create large point clouds
N <- 10^5; D <- 15
x <- matrix(rnorm(N*D), N, D)
y <- matrix(rnorm(N*D), N, D)
b <- matrix(rnorm(N*D), N, D)
# Turn dense arrays into symbolic matrices
x_i  <- LazyTensor(x, "i")
y_j  <- LazyTensor(y, "j")
b_j  <- LazyTensor(b, "j")
K_ij <- exp( -sum((x_i - y_j)^2) )    # symbolic N×N Gaussian kernel
# Call sum() reduction to trigger the computation
a_i  <- sum(K_ij * b_j, index = "j")
dim(a_i); class(a_i)
# [1] 100000     15
# [1] "matrix" "array"
```

Total running time of the script: 4.373 sec.[1]

1. using 16 cores of an *Intel Xeon Gold 6142* processor

Symbolic evaluation with RKeOps LazyTensors

Let us perform the Gaussian reduction (1) with RKeOps: $\left[\sum_{j=1}^{N} k(\mathbf{x}_i, \mathbf{y}_j) b_j\right]_{i=1}^{M}$

```
# Create large point clouds
N <- 10^5; D <- 15
x <- matrix(rnorm(N*D), N, D)
y <- matrix(rnorm(N*D), N, D)
b <- matrix(rnorm(N*D), N, D)
# Turn dense arrays into symbolic matrices
x_i  <- LazyTensor(x, "i")
y_j  <- LazyTensor(y, "j")
b_j  <- LazyTensor(b, "j")
K_ij <- exp( -sum((x_i - y_j)^2) )    # symbolic N×N Gaussian kernel
# Call sum() reduction to trigger the computation
a_i  <- sum(K_ij * b_j, index = "j")
dim(a_i); class(a_i)
# [1] 100000     15
# [1] "matrix" "array"
```

Total running time of the script: 4.373 sec.[1]

1. using 16 cores of an *Intel Xeon Gold 6142* processor

Symbolic evaluation with RKeOps LazyTensors

Let us perform the Gaussian reduction (1) with RKeOps: $\left[\sum_{j=1}^{N} k(x_i, y_j) b_j\right]_{i=1}^{M}$

```r
# Create large point clouds
N <- 10^5; D <- 15
x <- matrix(rnorm(N*D), N, D)
y <- matrix(rnorm(N*D), N, D)
b <- matrix(rnorm(N*D), N, D)
# Turn dense arrays into symbolic matrices
x_i  <- LazyTensor(x, "i")
y_j  <- LazyTensor(y, "j")
b_j  <- LazyTensor(b, "j")
K_ij <- exp( -sum((x_i - y_j)^2) )    # symbolic N×N Gaussian kernel
# Call sum() reduction to trigger the computation
a_i  <- sum(K_ij * b_j, index = "j")
dim(a_i); class(a_i)
# [1] 100000     15
# [1] "matrix" "array"
```

Total running time of the script: 4.373 sec.[1]

1. using 16 cores of an *Intel Xeon Gold 6142* processor

Symbolic evaluation with RKeOps LazyTensors

Let us perform the Gaussian reduction (1) with RKeOps: $\left[\sum_{j=1}^{N} k(\mathbf{x}_i, \mathbf{y}_j) b_j\right]_{i=1}^{M}$

```
# Create large point clouds
N <- 10^5; D <- 15
x <- matrix(rnorm(N*D), N, D)
y <- matrix(rnorm(N*D), N, D)
b <- matrix(rnorm(N*D), N, D)
# Turn dense arrays into symbolic matrices
x_i  <- LazyTensor(x, "i")
y_j  <- LazyTensor(y, "j")
b_j  <- LazyTensor(b, "j")
K_ij <- exp( -sum((x_i - y_j)^2) )    # symbolic N×N Gaussian kernel
# Call sum() reduction to trigger the computation
a_i  <- sum(K_ij * b_j, index = "j")
dim(a_i); class(a_i)
# [1] 100000     15
# [1] "matrix" "array"
```

Total running time of the script: 4.373 sec.[1]

1. using 16 cores of an *Intel Xeon Gold 6142* processor

Generic reduction with RKeOps

`RKeOps` supports all kinds of reduction. For $1 \leq i \leq M$, compute

$$\left[\text{Reduction}_{j=1,\dots,M} F(\mathrm{p}^1, \mathrm{p}^2, \dots, \mathrm{x}_i^1, \mathrm{x}_i^2, \dots, \mathrm{y}_j^1, \mathrm{y}_j^2, \dots)\right]_{i=1,\dots,M}$$

where

- "Reduction" can be any reduction over a dimension (Sum, Max, ArgMax, LogSumExp...)
- F is a vector-valued formula
- $\mathrm{x}_i^1, \mathrm{x}_i^2, \dots$ are vector variables indexed by i
- $\mathrm{y}_j^1, \mathrm{y}_j^2, \dots$ are vector variables indexed by j
- $\mathrm{p}^1, \mathrm{p}^2, \dots$ are vector parameter fixed across indices

2. https://github.com/getkeops/keops/blob/main/rkeops/vignettes/LazyTensor_rkeops.Rmd

Generic reduction with RKeOps

`RKeOps` supports all kinds of reduction. For $1 \leq i \leq M$, compute

$$\left[\text{Reduction}_{j=1,\ldots,M} F(\mathrm{p}^1, \mathrm{p}^2, \ldots, \mathrm{x}_i^1, \mathrm{x}_i^2, \ldots, \mathrm{y}_j^1, \mathrm{y}_j^2, \ldots)\right]_{i=1,\ldots,M}$$

where

- "Reduction" can be any reduction over a dimension (Sum, Max, ArgMax, LogSumExp...)
- F is a vector-valued formula
- $\mathrm{x}_i^1, \mathrm{x}_i^2, \ldots$ are vector variables indexed by i
- $\mathrm{y}_j^1, \mathrm{y}_j^2, \ldots$ are vector variables indexed by j
- $\mathrm{p}^1, \mathrm{p}^2, \ldots$ are vector parameter fixed across indices

The full range of reductions and operations provided by `RKeOps` is available in the vignette [2].

2. https://github.com/getkeops/keops/blob/main/rkeops/vignettes/LazyTensor_rkeops.Rmd

Generic reduction with RKeOps

`RKeOps` supports all kinds of reduction. For $1 \leq i \leq M$, compute

$$\left[\text{Reduction}_{j=1,\ldots,M} F(\mathrm{p}^1, \mathrm{p}^2, \ldots, \mathrm{x}_i^1, \mathrm{x}_i^2, \ldots, \mathrm{y}_j^1, \mathrm{y}_j^2, \ldots)\right]_{i=1,\ldots,M}$$

where

- "Reduction" can be any reduction over a dimension (Sum, Max, ArgMax, LogSumExp...)
- F is a vector-valued formula
- $\mathrm{x}_i^1, \mathrm{x}_i^2, \ldots$ are vector variables indexed by i
- $\mathrm{y}_j^1, \mathrm{y}_j^2, \ldots$ are vector variables indexed by j
- $\mathrm{p}^1, \mathrm{p}^2, \ldots$ are vector parameter fixed across indices

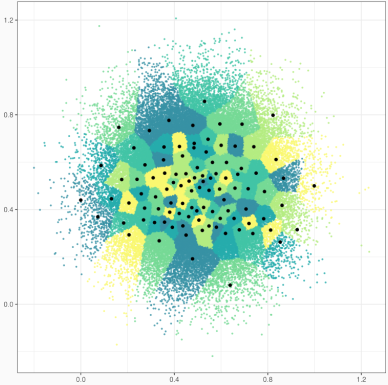
The full range of reductions and operations provided by `RKeOps` is available in the vignette [2].

Note: `RKeOps` also supports reductions on complex data!

2. https://github.com/getkeops/keops/blob/main/rkeops/vignettes/LazyTensor_rkeops.Rmd

Example I - K-means clustering

At each iteration, compute $\operatorname{argmin}_{j=1,\ldots,K} \|\mathbf{x}_i - \mathbf{c}_j\|$ for $i = 1, \ldots, N$, where $\mathbf{c}_j$ is the centroid of cluster j.



Example of 50-means clustering with $N = 10^5$ points in $\mathbb{R}^2$ and the Euclidean distance.

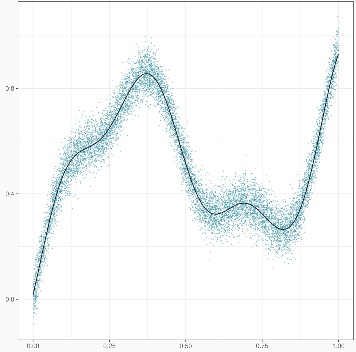
Time for 10 iterations on CPU: 2.224 sec. (0.222 sec. per iteration)

Example II - Kernel interpolation

For $\lambda \in \mathbb{R}_+$, let us solve a linear system of the form

$$\mathrm{a}^* = \operatorname*{argmin}_{\mathrm{a}} \frac{1}{2}\langle \mathrm{a}, (\lambda \mathrm{Id} + \mathrm{K})\mathrm{a}\rangle - \langle \mathrm{a}, \mathrm{b}\rangle = (\lambda \mathrm{Id} + \mathrm{K})^{-1}\mathrm{b}$$

where K is a symmetric, positive definite linear operator defined with a symbolic formula.



Example of 1D interpolation with a Gaussian kernel matrix on $N = 10^4$ points.

Time to perform the interpolation with a precision of 10^{-6} on CPU: 13 sec.

AutoDiff with RKeOps

What if we need the gradient of $\mathbf{a} = (a_1, \dots, a_M)$, say with respect to $\mathbf{y}$?

$$\partial_{\mathbf{y}} \mathbf{a} = \partial_{\mathbf{y}} (a_i)_{i=1,\dots,M} = \partial_{\mathbf{y}} \left(\sum_{j=1}^{N} k(\mathbf{x}_i, \mathbf{y}_j) b_j \right)_{i=1,\dots,M}$$

`RKeOps LazyTensors` $\rightarrow$ do not support automatic differentiation **(yet!)**

`KeOps` $\rightarrow$ provides an autodiff engine for formulae wrapped in `Grad( )`:

```
# Define a formula with a gradient
formula_grad <- 'Grad(Sum_Reduction(Exp(-SqDist(x,y)) * b , 0), y, e)'
variables <- c('x = Vi(15)', 'y = Vj(15)', 'b = Vj(15)', 'e = Vi(15)')
# Compile the corresponding operator
gaussian_kernel_grad <- keops_kernel(formula_grad, variables)
# Declare a new tensor used as the input of the gradient operator
e <- matrix(rnorm(N*D), N, D)
# Computation
res <- gaussian_kernel_grad(list(x, y, b, e))
```

Total running time of the script: 11.975 sec.[3]

3. using 16 cores of an *Intel Xeon Gold 6142* processor

AutoDiff with RKeOps

What if we need the gradient of $\mathbf{a} = (a_1, \dots, a_M)$, say with respect to $\mathbf{y}$?

$$\partial_{\mathbf{y}} \mathbf{a} = \partial_{\mathbf{y}} (a_i)_{i=1,\dots,M} = \partial_{\mathbf{y}} \left(\sum_{j=1}^{N} k(\mathbf{x}_i, \mathbf{y}_j) b_j \right)_{i=1,\dots,M}$$

`RKeOps LazyTensors` $\to$ do not support automatic differentiation **(yet!)**

`KeOps` $\to$ provides an autodiff engine for formulae wrapped in `Grad( )`:

```
# Define a formula with a gradient
formula_grad <- 'Grad(Sum_Reduction(Exp(-SqDist(x,y)) * b , 0), y, e)'
variables <- c('x = Vi(15)', 'y = Vj(15)', 'b = Vj(15)', 'e = Vi(15)')
# Compile the corresponding operator
gaussian_kernel_grad <- keops_kernel(formula_grad, variables)
# Declare a new tensor used as the input of the gradient operator
e <- matrix(rnorm(N*D), N, D)
# Computation
res <- gaussian_kernel_grad(list(x, y, b, e))
```

Total running time of the script: 11.975 sec.[3]

3. using 16 cores of an *Intel Xeon Gold 6142* processor

AutoDiff with RKeOps

What if we need the gradient of $\mathbf{a} = (a_1, \ldots, a_M)$, say with respect to $\mathbf{y}$?

$$\partial_{\mathbf{y}}\mathbf{a} = \partial_{\mathbf{y}}(a_i)_{i=1,\ldots,M} = \partial_{\mathbf{y}}\left(\sum_{j=1}^{N} k(\mathbf{x}_i, \mathbf{y}_j) b_j\right)_{i=1,\ldots,M}$$

`RKeOps LazyTensors` $\rightarrow$ do not support automatic differentiation **(yet!)**

`KeOps` $\rightarrow$ provides an autodiff engine for formulae wrapped in `Grad( )`:

```
# Define a formula with a gradient
formula_grad <- 'Grad(Sum_Reduction(Exp(-SqDist(x,y)) * b , 0), y, e)'
variables <- c('x = Vi(15)', 'y = Vj(15)', 'b = Vj(15)', 'e = Vi(15)')
# Compile the corresponding operator
gaussian_kernel_grad <- keops_kernel(formula_grad, variables)
# Declare a new tensor used as the input of the gradient operator
e <- matrix(rnorm(N*D), N, D)
# Computation
res <- gaussian_kernel_grad(list(x, y, b, e))
```

Total running time of the script: 11.975 sec.[3]

3. using 16 cores of an *Intel Xeon Gold 6142* processor

AutoDiff with RKeOps

What if we need the gradient of $\mathbf{a} = (a_1, \ldots, a_M)$, say with respect to $\mathbf{y}$?

$$\partial_{\mathbf{y}} \mathbf{a} = \partial_{\mathbf{y}} (a_i)_{i=1,\ldots,M} = \partial_{\mathbf{y}} \left(\sum_{j=1}^{N} k(\mathbf{x}_i, \mathbf{y}_j) b_j \right)_{i=1,\ldots,M}$$

`RKeOps LazyTensors` $\rightarrow$ do not support automatic differentiation **(yet!)**

`KeOps` $\rightarrow$ provides an autodiff engine for formulae wrapped in `Grad( )`:

```
# Define a formula with a gradient
formula_grad <- 'Grad(Sum_Reduction(Exp(-SqDist(x,y)) * b , 0), y, e)'
variables <- c('x = Vi(15)', 'y = Vj(15)', 'b = Vj(15)', 'e = Vi(15)')
# Compile the corresponding operator
gaussian_kernel_grad <- keops_kernel(formula_grad, variables)
# Declare a new tensor used as the input of the gradient operator
e <- matrix(rnorm(N*D), N, D)
# Computation
res <- gaussian_kernel_grad(list(x, y, b, e))
```

Total running time of the script: 11.975 sec.[3]

3. using 16 cores of an *Intel Xeon Gold 6142* processor

AutoDiff with RKeOps

What if we need the gradient of $\mathbf{a} = (a_1, \ldots, a_M)$, say with respect to $\mathbf{y}$?

$$\partial_{\mathbf{y}}\mathbf{a} = \partial_{\mathbf{y}}(a_i)_{i=1,\ldots,M} = \partial_{\mathbf{y}}\left(\sum_{j=1}^{N} k(\mathbf{x}_i, \mathbf{y}_j) b_j\right)_{i=1,\ldots,M}$$

`RKeOps LazyTensors` $\to$ do not support automatic differentiation **(yet!)**

`KeOps` $\to$ provides an autodiff engine for formulae wrapped in `Grad( )`:

```
# Define a formula with a gradient
formula_grad <- 'Grad(Sum_Reduction(Exp(-SqDist(x,y)) * b , 0), y, e)'
variables <- c('x = Vi(15)', 'y = Vj(15)', 'b = Vj(15)', 'e = Vi(15)')
# Compile the corresponding operator
gaussian_kernel_grad <- keops_kernel(formula_grad, variables)
# Declare a new tensor used as the input of the gradient operator
e <- matrix(rnorm(N*D), N, D)
# Computation
res <- gaussian_kernel_grad(list(x, y, b, e))
```

Total running time of the script: 11.975 sec.[3]

3. using 16 cores of an *Intel Xeon Gold 6142* processor

AutoDiff with RKeOps

What if we need the gradient of $\mathbf{a} = (a_1, \ldots, a_M)$, say with respect to $\mathbf{y}$?

$$\partial_{\mathbf{y}} \mathbf{a} = \partial_{\mathbf{y}} (a_i)_{i=1,\ldots,M} = \partial_{\mathbf{y}} \left(\sum_{j=1}^{N} k(\mathbf{x}_i, \mathbf{y}_j) b_j \right)_{i=1,\ldots,M}$$

`RKeOps LazyTensors` $\rightarrow$ do not support automatic differentiation **(yet!)**

`KeOps` $\rightarrow$ provides an autodiff engine for formulae wrapped in `Grad( )`:

```
# Define a formula with a gradient
formula_grad <- 'Grad(Sum_Reduction(Exp(-SqDist(x,y)) * b , 0), y, e)'
variables <- c('x = Vi(15)', 'y = Vj(15)', 'b = Vj(15)', 'e = Vi(15)')
# Compile the corresponding operator
gaussian_kernel_grad <- keops_kernel(formula_grad, variables)
# Declare a new tensor used as the input of the gradient operator
e <- matrix(rnorm(N*D), N, D)
# Computation
res <- gaussian_kernel_grad(list(x, y, b, e))
```

Total running time of the script: 11.975 sec.[3]

3. using 16 cores of an *Intel Xeon Gold 6142* processor

What if we need the gradient of $\mathbf{a} = (a_1, \ldots, a_M)$, say with respect to $\mathbf{y}$?

$$\partial_{\mathbf{y}} \mathbf{a} = \partial_{\mathbf{y}} (a_i)_{i=1,\ldots,M} = \partial_{\mathbf{y}} \left(\sum_{j=1}^{N} k(\mathbf{x}_i, \mathbf{y}_j) b_j \right)_{i=1,\ldots,M}$$

`RKeOps LazyTensors` $\rightarrow$ do not support automatic differentiation **(yet!)**

`KeOps` $\rightarrow$ provides an autodiff engine for formulae wrapped in `Grad( )`:

```
# Define a formula with a gradient
formula_grad <- 'Grad(Sum_Reduction(Exp(-SqDist(x,y)) * b , 0), y, e)'
variables <- c('x = Vi(15)', 'y = Vj(15)', 'b = Vj(15)', 'e = Vi(15)')
# Compile the corresponding operator
gaussian_kernel_grad <- keops_kernel(formula_grad, variables)
# Declare a new tensor used as the input of the gradient operator
e <- matrix(rnorm(N*D), N, D)
# Computation
res <- gaussian_kernel_grad(list(x, y, b, e))
```

Total running time of the script: 11.975 sec.[3]

3. using 16 cores of an *Intel Xeon Gold 6142* processor

AutoDiff with RKeOps

What if we need the gradient of $\mathbf{a} = (a_1, \dots, a_M)$, say with respect to $\mathbf{y}$?

$$\partial_{\mathbf{y}} \mathbf{a} = \partial_{\mathbf{y}} (a_i)_{i=1,\dots,M} = \partial_{\mathbf{y}} \left(\sum_{j=1}^{N} k(\mathbf{x}_i, \mathbf{y}_j) b_j \right)_{i=1,\dots,M}$$

`RKeOps LazyTensors` $\to$ do not support automatic differentiation **(yet!)**

`KeOps` $\to$ provides an autodiff engine for formulae wrapped in `Grad( )`:

```
# Define a formula with a gradient
formula_grad <- 'Grad(Sum_Reduction(Exp(-SqDist(x,y)) * b , 0), y, e)'
variables <- c('x = Vi(15)', 'y = Vj(15)', 'b = Vj(15)', 'e = Vi(15)')
# Compile the corresponding operator
gaussian_kernel_grad <- keops_kernel(formula_grad, variables)
# Declare a new tensor used as the input of the gradient operator
e <- matrix(rnorm(N*D), N, D)
# Computation
res <- gaussian_kernel_grad(list(x, y, b, e))
```

Total running time of the script: 11.975 sec.[3]

3. using 16 cores of an *Intel Xeon Gold 6142* processor

AutoDiff with RKeOps

What if we need the gradient of $\mathbf{a} = (a_1, \ldots, a_M)$, say with respect to $\mathbf{y}$?

$$\partial_{\mathbf{y}} \mathbf{a} = \partial_{\mathbf{y}} (a_i)_{i=1,\ldots,M} = \partial_{\mathbf{y}} \left(\sum_{j=1}^{N} k(\mathbf{x}_i, \mathbf{y}_j) b_j \right)_{i=1,\ldots,M}$$

`RKeOps LazyTensors` $\rightarrow$ do not support automatic differentiation **(yet!)**

`KeOps` $\rightarrow$ provides an autodiff engine for formulae wrapped in `Grad( )`:

```
# Define a formula with a gradient
formula_grad <- 'Grad(Sum_Reduction(Exp(-SqDist(x,y)) * b , 0), y, e)'
variables <- c('x = Vi(15)', 'y = Vj(15)', 'b = Vj(15)', 'e = Vi(15)')
# Compile the corresponding operator
gaussian_kernel_grad <- keops_kernel(formula_grad, variables)
# Declare a new tensor used as the input of the gradient operator
e <- matrix(rnorm(N*D), N, D)
# Computation
res <- gaussian_kernel_grad(list(x, y, b, e))
```

Total running time of the script: 11.975 sec.[3]

3. using 16 cores of an *Intel Xeon Gold 6142* processor

What if we need the gradient of $\mathbf{a} = (a_1, \ldots, a_M)$, say with respect to $\mathbf{y}$?

$$\partial_{\mathbf{y}} \mathbf{a} = \partial_{\mathbf{y}} (a_i)_{i=1,\ldots,M} = \partial_{\mathbf{y}} \left(\sum_{j=1}^{N} k(\mathbf{x}_i, \mathbf{y}_j) b_j \right)_{i=1,\ldots,M}$$

`RKeOps LazyTensors` $\rightarrow$ do not support automatic differentiation **(yet!)**

`KeOps` $\rightarrow$ provides an autodiff engine for formulae wrapped in `Grad( )`:

```
# Define a formula with a gradient
formula_grad <- 'Grad(Sum_Reduction(Exp(-SqDist(x,y)) * b , 0), y, e)'
variables <- c('x = Vi(15)', 'y = Vj(15)', 'b = Vj(15)', 'e = Vi(15)')
# Compile the corresponding operator
gaussian_kernel_grad <- keops_kernel(formula_grad, variables)
# Declare a new tensor used as the input of the gradient operator
e <- matrix(rnorm(N*D), N, D)
# Computation
res <- gaussian_kernel_grad(list(x, y, b, e))
```

Total running time of the script: 11.975 sec.[3]

3. using 16 cores of an *Intel Xeon Gold 6142* processor

Technical specifications

`KeOps`[4] Core library

- written in `C++/Python`
- dependencies: a `C++` compiler (`g++`, `clang`) and `nvrtc` headers provided by CUDA for GPU computing

4. Benjamin Charlier et al. “Kernel Operations on the GPU, with Autodiff, without Memory Overflows”. In: *Journal of Machine Learning Research* 22.74 (2021), pp. 1–6. URL: `http://jmlr.org/papers/v22/20-275.html`.
5. Kevin Ushey, JJ Allaire, and Yuan Tang. *reticulate: Interface to 'Python'*. https://rstudio.github.io/reticulate/, https://github.com/rstudio/reticulate. 2023.

Technical specifications

`KeOps`[4] Core library

- written in `C++/Python`
- dependencies: a `C++` compiler (`g++`, `clang`) and `nvrtc` headers provided by `CUDA` for GPU computing

`PyKeOps` `Python` binders for `KeOps` (both `NumPy` and `PyTorch`)

4. Benjamin Charlier et al. “Kernel Operations on the GPU, with Autodiff, without Memory Overflows”. In: *Journal of Machine Learning Research* 22.74 (2021), pp. 1–6. URL: `http://jmlr.org/papers/v22/20-275.html`.
5. Kevin Ushey, JJ Allaire, and Yuan Tang. *reticulate: Interface to ’Python’*. https://rstudio.github.io/reticulate/, https://github.com/rstudio/reticulate. 2023.

Technical specifications

KeOps[4] Core library

- written in C++/Python
- dependencies: a C++ compiler (g++, clang) and nvrtc headers provided by CUDA for GPU computing

PyKeOps Python binders for KeOps (both NumPy and PyTorch)

RKeOps R binder for KeOps

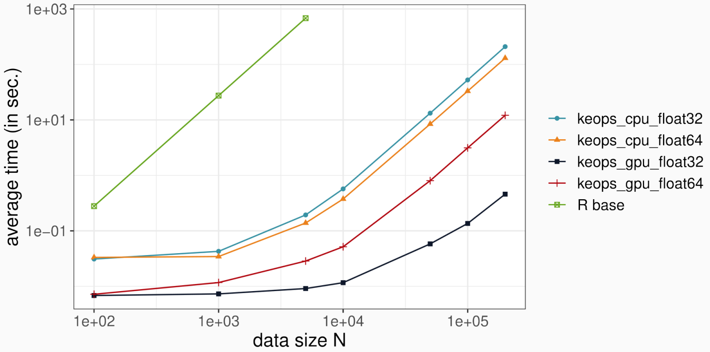
- since v.2.0, directly uses PyKeOps through reticulate[5]
- GPU computing directly inside R: just type rkeops_use_gpu()!
- soon on the CRAN, already available on github:

```
install.packages("remotes")
remotes::install_github("getkeops/keops", subdir = "rkeops")
```

4. Benjamin Charlier et al. "Kernel Operations on the GPU, with Autodiff, without Memory Overflows". In: *Journal of Machine Learning Research* 22.74 (2021), pp. 1–6. URL: http://jmlr.org/papers/v22/20-275.html.
5. Kevin Ushey, JJ Allaire, and Yuan Tang. *reticulate: Interface to 'Python'*. https://rstudio.github.io/reticulate/, https://github.com/rstudio/reticulate. 2023.

Benchmark I - Gaussian convolution

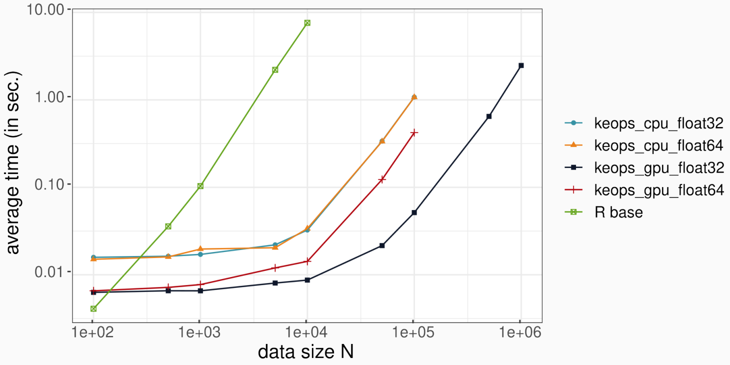
Runtimes [6] for Gaussian convolution with N samples in $\mathbb{R}^{15}$.



6. CPU: 16 cores of an *Intel Xeon Gold 6142* processor. GPU: *Nvidia A10*.

Benchmark II - KNN-search

Runtimes[7] for 10-Nearest Neighbors search with N samples in $\mathbb{R}^3$.



7. CPU: 16 cores of an *Intel Xeon Gold 6142* processor. GPU: *Nvidia A10*.

Take home message

RKeOps: Fast kernel operations on GPU without memory overflow and with automatic differenciation, directly inside R

- full documentation, tutorials, examples, benchmarks and more at https://www.kernel-operations.io
- active development and open contributions at https://github.com/getkeops/keops/blob/main/rkeops/
- current install process:

```
install.packages("remotes")
remotes::install_github("getkeops/keops", subdir = "rkeops")
```

- available on CRAN soon!

Thank you for your attention! Questions?

[1] Benjamin Charlier et al. “Kernel Operations on the GPU, with Autodiff, without Memory Overflows”. In: *Journal of Machine Learning Research* 22.74 (2021), pp. 1–6. URL: `http://jmlr.org/papers/v22/20-275.html`.

[2] Kevin Ushey, JJ Allaire, and Yuan Tang. *reticulate: Interface to 'Python'*. https://rstudio.github.io/reticulate/, https://github.com/rstudio/reticulate. 2023.