KeOps

Seamless Kernel Operations on GPU, with auto-differentiation and without memory overflows

---

B. Charlier (IMAG - Univ Montpellier)
J. Feydy (Imperial College, London)

G. Durif (CNRS - IMAG - Univ Montpellier)
J. Glaunès (MAP5 - Univ Paris Descartes)

October 9th 2019 – *Journées Calcul et Données* (JCAD), Toulouse

`http://www.kernel-operations.io/`
`ghislain.durif@umontpellier.fr`

Outline

Introduction

What KeOps can do?

- Compute generic reductions of very large arrays/matrices

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

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

- Compute kernel dot products and the associated gradients

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

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

What KeOps can do?

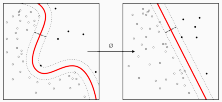
- Compute generic reductions of very large arrays/matrices
- Compute kernel dot products and the associated gradients

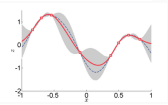
  → large dimensions M and N ($\approx 10^4$ ou 10^6)

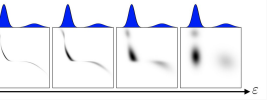
  → fast computation on GPU without memory overflow

Kernel spaces in statistics and Learning

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





Motivations

- GPU user-friendly computing: development effort oriented for deep learning

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

- GPU computing can be used for general purpose computations, not only neural networks

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

- Needs: provide an effortless tool for GPU computing (application: statistics, machine learning and more)

Matrix reduction and kernel operations

Matrix reduction

- Simple row or column-wise matrix reduction

$$\left[\sum_{i=1}^{M} a_{ij}\right]_{j=1,\dots,N} \in \mathbb{R}^N \quad \text{or} \quad \left[\sum_{j=1}^{N} a_{ij}\right]_{i=1,\dots,M} \in \mathbb{R}^M$$

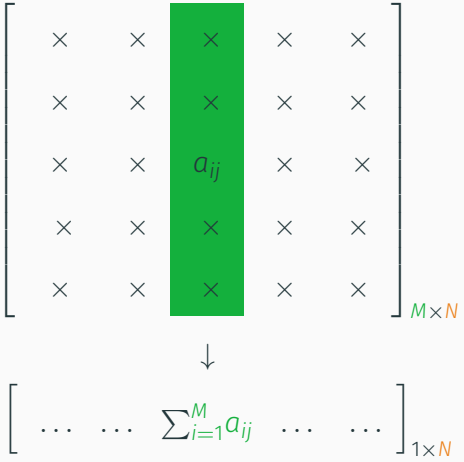
  for a matrix $\mathbf{A} = [a_{ij}] \in \mathbb{R}^{M \times N}$

- Vector/matrix or matrix/matrix product

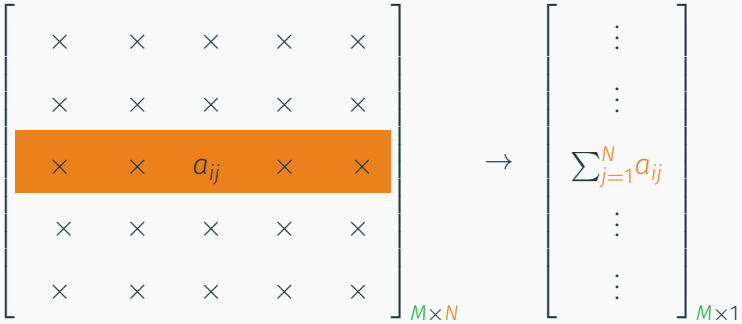
$$\left[\sum_{i=1}^{M} a_{ij}\, \beta_j\right]_{j=1,\dots,N} \in \mathbb{R}^N \quad \text{or} \quad \left[\sum_{j=1}^{N} a_{ij}\, \beta_j\right]_{i=1,\dots,M} \in \mathbb{R}^M$$

  for a matrix $\mathbf{A} = [a_{ij}] \in \mathbb{R}^{M \times N}$ and a vector $\boldsymbol{\beta} = [\beta_j] \in \mathbb{R}^N$

Matrix reduction



Matrix reduction



Kernel operator

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

- (Intuitevely) a kernel function is an application $K : \mathbb{R}^D \times \mathbb{R}^D \to \mathbb{R}$

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

  corresponding to a scalar product between $\mathbf{x}_i$ and $\mathbf{y}_j$ in a different space

- Example

$$\text{Linear kernel:} \quad K(\mathbf{x}_i, \mathbf{y}_j) = \left\langle \mathbf{x}_i \, , \, \mathbf{y}_j \right\rangle = \mathbf{x}_i^T \mathbf{y}_j = \sum_{k=1}^{D} x_{ik} \, y_{jk}$$

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

Kernel reduction

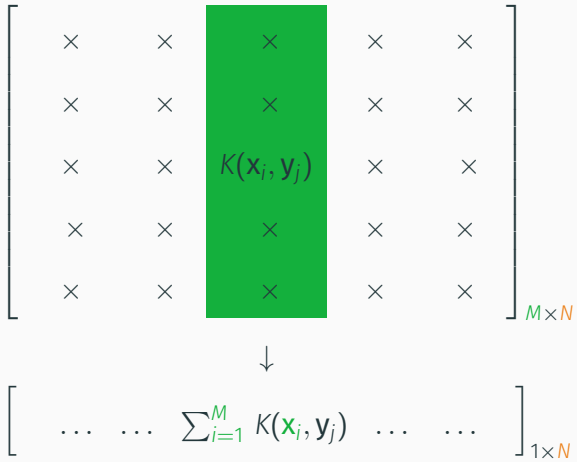
- Convolution-like operation

$$\left[\sum_{i=1}^{M} K(\mathbf{x}_i, \mathbf{y}_j)\,\beta_j\right]_{j=1,\dots,N} \in \mathbb{R}^N \quad \text{or} \quad \left[\sum_{j=1}^{N} K(\mathbf{x}_i, \mathbf{y}_j)\,\beta_j\right]_{i=1,\dots,M} \in \mathbb{R}^M$$

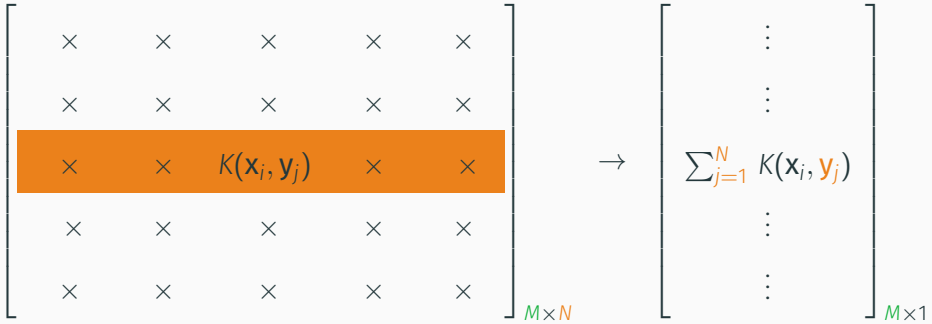
for some D-vectors $(\mathbf{x}_i)_{i=1,\dots,M} \in \mathbb{R}^{M\times D}$, $(\mathbf{y}_j)_{j=1,\dots,N} \in \mathbb{R}^{N\times D}$ and $\boldsymbol{\beta} = [\beta_j] \in \mathbb{R}^N$

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

Kernel reduction



Kernel reduction



Kernel reduction

- More complex operation

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

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 $(\mathbf{y}_j)_j, (\mathbf{v}_j)_j, (\boldsymbol{\beta}_j)_j \in \mathbb{R}^{N \times D}$

Generic reduction in KeOps

$1 \leq i \leq N$ et $1 \leq j \leq M$ avec $N, M \approx 10^4$ ou 10^6

- A generic case:

$$\left[\sum_j F(\sigma_1, \cdots, \sigma_\ell, X_i^1, \cdots, X_i^k, Y_j^1, \cdots, Y_j^m) \right]_{i=1,\ldots,M} \in \mathbb{R}^M$$

- ...an even more generic case:

$$\left[\bigstar_j F(\sigma_1, \cdots, \sigma_\ell, X_i^1, \cdots, X_i^k, Y_j^1, \cdots, Y_j^m) \right]_{i=1,\ldots,M} \in \mathbb{R}^M$$

where $\bigstar$ can be any reduction (sum, max, min, etc.) over a dimension

Why GPU computing

- Matrix/kernel reduction = combination of generic matrix operations
- GPU are good for matrix computations
- **Problem:** the matrix $\mathbf{K} = \left[K(\mathbf{x}_i, \mathbf{y}_j)\right] \in \mathbb{R}^{M \times N}$ is very large ($M, N \approx 10^4$ ou 10^6)

  $\rightarrow$ how to store it in memory

  $\rightarrow$ how to iterate through rows/columns to do computations

Computation on GPU

The GPU Market by Nvidia

Target:

- Gamers: 1000 euros

- Scientific computing: 3000 – 9000 euros

Under the hood: similar chipsets with few enhancements (ECC, float64,...)

GPU = massively parallel architecture

Nividia GTX XXXX architecture

A GPU architecture

→ scalable array of multithreaded *Streaming Multiprocessors* (*SM*s)

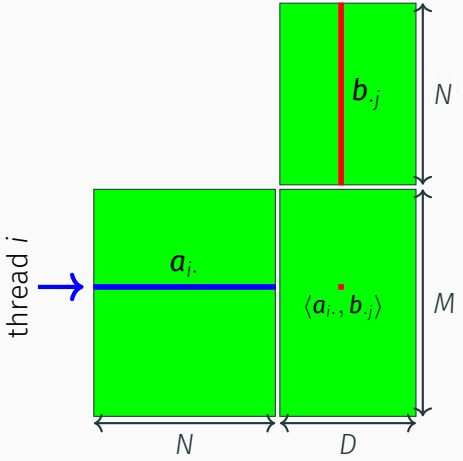
→ each single processor (called a *thread*) is able to execute an independent set of instructions.

1000's of cores inside a single GPU

(multi-CPU architecure = at best 10's – 100's of cores)

MatMult: A first naive implementation

$\mathbf{A} \in \mathbb{R}^{M \times N}$ and $\mathbf{B} \in \mathbb{R}^{N \times D}$



A matrix multiplication

$$\mathbf{A}\,\mathbf{B} = \left[\sum_k a_{ik}\, b_{kj}\right]_{M \times D}$$

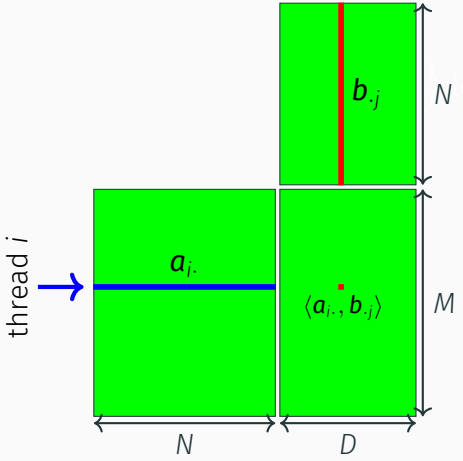
$\rightarrow$ a set of $N \times D$ scalar products

Parallel computing

- each thread computes D scalar products, i.e. $\langle \mathbf{a}_{i\cdot}, \mathbf{b}_{\cdot j} \rangle$ for all j

MatMult: A first naive implementation

$\mathbf{A} \in \mathbb{R}^{M \times N}$ and $\mathbf{B} \in \mathbb{R}^{N \times D}$



Thread i needs to access

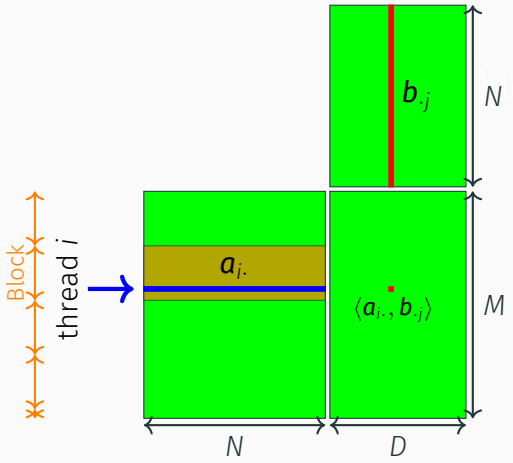
- row $\mathbf{a}_{i\cdot} \in \mathbb{R}^N$
- all columns $(\mathbf{b}_{\cdot j})_{j=1,\dots,D}$ i.e. the full matrix $\mathbf{B}$

$\rightarrow$ potential memory overflow

$\rightarrow$ no mutualisation of memory access between threads

MatMult: A first naive implementation

$\mathbf{A} \in \mathbb{R}^{M \times N}$ and $\mathbf{B} \in \mathbb{R}^{N \times D}$

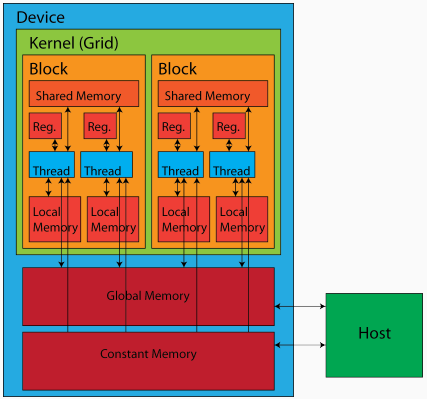


Assign a block of rows i to a thread

- mutualise the memory access to each $\mathbf{B}_{.j}$ to compute all rows i in the block

$\rightarrow$ each thread still requires to access the full matrix $\mathbf{B}$ to finish the computations for a row i

Memory management on GPU



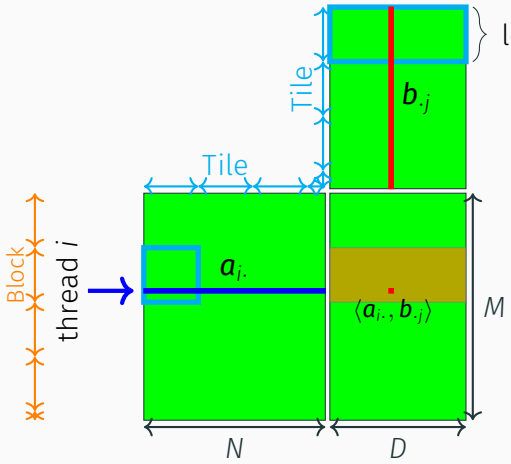
- Data initially stored on the host (in RAM)
  → should be transfer to the device (GPU) to be treated (bottleneck)
- Different kinds of memory
  → local vs shared memory

Smart use of the shared memory

→ less transfer between device and host

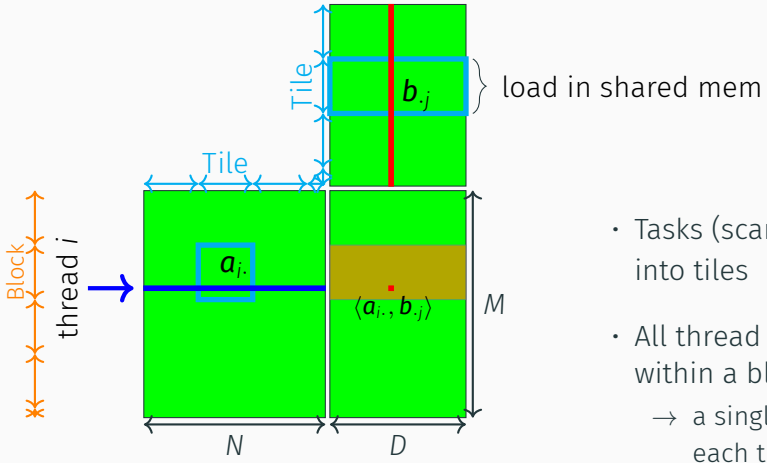
→ key to provide an efficient code in term of computational time

MatMult: Tiled implementation (decomposition with block sub-matrix product)



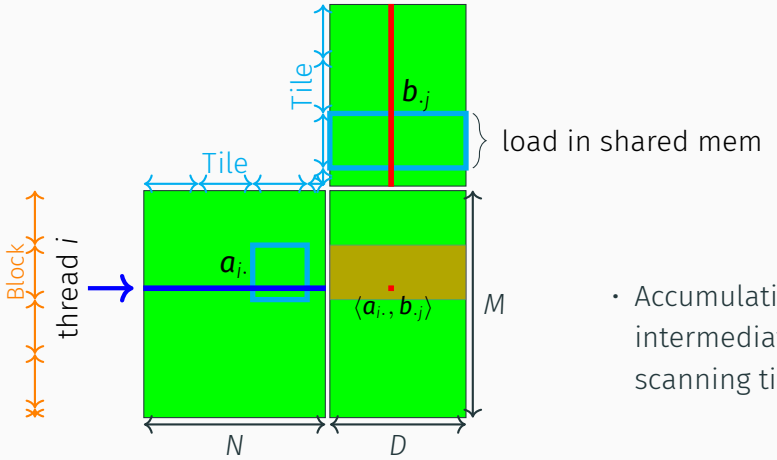
- Tasks (scanning rows $\mathbf{a}_i$) divided into tiles
- All thread use the shared memory within a block
  - $\rightarrow$ a single memory transfered of each tile in $\mathbf{B}$ for all threads

MatMult: Tiled implementation (decomposition with block sub-matrix product)



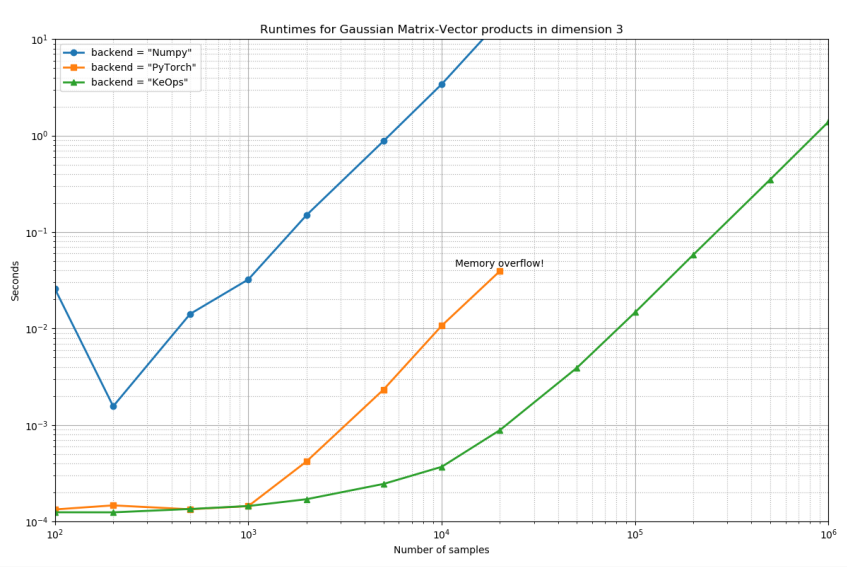
- Tasks (scanning rows $\mathbf{a}_i$) divided into tiles
- All thread use the shared memory within a block
  - $\rightarrow$ a single memory transfered of each tile in $\mathbf{B}$ for all threads

MatMult: Tiled implementation (decomposition with block sub-matrix product)

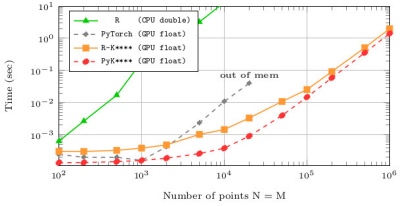


- Accumulation (addition of the intermediate results) when scanning tiles across **A**

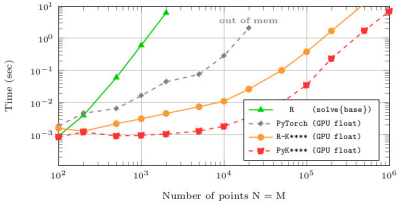
Benchmark I


Runtimes for Gaussian Matrix-Vector products in dimension 3
backend = "Numpy"
backend = "PyTorch"
backend = "KeOps"
Memory overflow!
Seconds
Number of samples
10^{1}
10^{0}
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^{2}
10^{3}
10^{4}
10^{5}
10^{6}

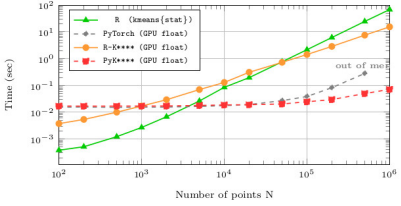
Benchmark II



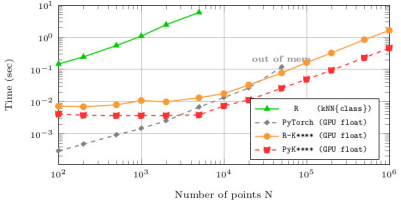
(a) Matrix-vector products with N-by-N Gaussian kernel matrices built from point clouds in dimension D = 3.



(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 $\mathbf{x}, \mathbf{y} \in \mathbb{R}^D$:

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

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

```
"Exp(Scalprod(x,y))"
```

- Then it is expanded internally in the C++ code using templates:

```
F=Exp<Scalprod<X,Y>>
```

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

→ KeOps is able to generate shared objects that compute on a GPU (compilation on the fly)

Coding generic formulas with KeOps

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

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

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

```
"Exp(Scalprod(x,y))"
```

- Then it is expanded internally in the C++ code using templates:

```
F=Exp<Scalprod<X,Y>>
```

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

→ KeOps is able to generate shared objects that compute on a GPU (compilation on the fly)

Coding generic formulas with KeOps

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

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

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

```
"Exp(Scalprod(x,y))"
```

- Then it is expanded internally in the C++ code using templates:

```
F=Exp<Scalprod<X,Y>>
```

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

→ KeOps is able to generate shared objects that compute on a GPU (compilation on the fly)

Coding generic formulas with KeOps

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

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

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

```
"Exp(Scalprod(x,y))"
```

- Then it is expanded internally in the C++ code using templates:

```
F=Exp<Scalprod<X,Y>>
```

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

→ KeOps is able to generate shared objects that compute on a GPU (compilation on the fly)

Coding generic formulas with KeOps

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

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

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

```
"Exp(Scalprod(x,y))"
```

- Then it is expanded internally in the C++ code using templates:

```
F=Exp<Scalprod<X,Y>>
```

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

→ KeOps is able to generate shared objects that compute on a GPU (compilation on the fly)

Under the hood: C++ encoding of maths operations in KeOps I

A typical KeOps (unary) operation is a `struct` that looks like:

```
template<class F>
struct Exp : UnaryOp<Exp, F> {
///////////////// dimension of the output
  static const int DIM = F::DIM;
```

Implementation of the operation:

```
///////////////// inlined function in the final cuda code
  static DEVICE INLINE void Operation(TYPE *out, TYPE *in) {
#pragma unroll
    for (int k = 0; k < DIM; k++) {  out[k] = exp(in[k]); }
   }
```

Under the hood: C++ encoding of maths operations in KeOps I

Gradient computation:

```
////////////////  Autodiff!
   template<class V, class GRADIN>
   using DiffT = typename F::template DiffT<V, Mult<Exp<F>, GRADIN>>;
 };
```

String encoding:

```
////////////////  Macro providing high level syntax
 #define Exp(f) Exp<decltype(f)>()
```

Under the hood: C++ encoding of maths operations in KeOps II

A typical KeOps (binary) operation is a `struct` that looks like:

```
template < class FA, class FB >
struct Add : BinaryOp< Add, FA, FB > {
///////////////// dimension of the output ... and (compile time) checks
  static const int DIM = FA::DIM; // Output dim = FA::DIM = FB::DIM
  static_assert(DIM == FB::DIM, "Dimensions must be the same for Add");
```

Implementation of the operation:

```
///////////////// inlined function in the final cuda code
  static DEVICE INLINE void Operation(TYPE *out, TYPE *inA, TYPE *inB) {
  for (int k = 0; k < DIM; k++) {out[k] = inA[k] + inB[k];}
  }
```

Under the hood: C++ encoding of maths operations in KeOps II

Gradient computation:

```
////////////////  Autodiff!
  template < class V, class GRADIN >
  using DiffT = Add< typename FA::template DiffT< V, GRADIN >,
                     typename FB::template DiffT< V, GRADIN > >;
};
```

Simplification rule:

```
////////////////  Simplification rules: e.g.
template < class F >
struct Add_Alias0< F, F > { using type = Scal< IntConstant< 2 >, F >; };
```

Combining elementary operations

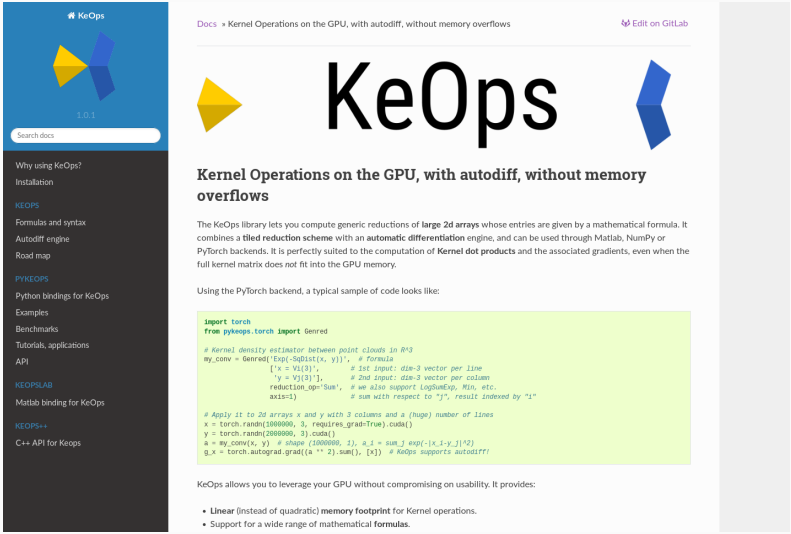
KeOps proposes a wide range of elementary operations

- **Simple vector operations:** scalar product, norm, distance, normalization, vector/vector element-wise operation (`+`,`-`,`*`,`/`), etc.
- **Elementary $\mathbb{R} \to \mathbb{R}$ functions:** exp, log, inverse, abs, pow, sqrt, sin, cos, etc.
- **Simple matrix operations:** matrix product, tensor product (in Python), etc.
- **Matrix reduction:** sum, min, max, argmin, argmax, etc.

$\to$ a formula = a combination of these operations

Using KeOps

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



- doc
- install instructions
- examples

KeOps stack

- Dependencies: `Cmake` ($\geq$3.10), `C++` compiler[1] (g++ $\geq$ 7 or clang) or `cuda` compiler (nvcc $\geq$10) and CUDA libs (for GPU computing)

- Open source (MIT licence): `github.com/getkeops/keops`

- Continuous integration (tested on linux distros and MacOs): Jenkins at `ci.inria.fr`

- Sphinx based documentation on `http://www.kernel-operations.io/`

[1] for CPU computing

KeOps user interface

- PyKeOps: `Python` (`numpy` and `pytorch`)

- KeOpsLab: `Matlab`

- RKeOps: `R` (beta version)

- `C++` API

Example in Python: single Gaussian convolution

We want to compute

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

with $s \in \mathbb{R}$, $[\mathbf{x}_i]_{i=1,\dots,M} \in \mathbb{R}^{M \times 3}$, $[\mathbf{y}_j]_{j=1,\dots,N} \in \mathbb{R}^{N \times 3}$ and $[\mathbf{b}_j]_{j=1,\dots,N} \in \mathbb{R}^{N \times 6}$

Example in Python: single Gaussian convolution

From Python using Numpy (similar in R or Matlab)

```
from pykeops.numpy import Genred

## compilation on the fly (user-friendly syntax)
my_conv = Genred(
            formula="Sum_Reduction(Exp(-s * SqNorm2(x - y)) * b, 0)",
            aliases=["s = Pm(1)",       # parameter (scalar)
                     "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)
            dtype='float32')

# assuming s, x, y and b are Numpy arrays (data and parameter values)

## compute directly on the GPU
gamma = my_conv(s, x, y, b)
```

Example in Python (LazyTensor)

Mathematical formula (standard Gaussian kernel)

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

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

Example in Python (LazyTensor)

Create two arrays with 3 columns and a (huge) number of lines, on the GPU

```
import torch
x = torch.randn(1000000, 3, requires_grad=True).cuda()
y = torch.randn(2000000, 3).cuda()
```

Given the same data tensors `x` and `y`. Use a decorator to turn tensors into KeOps symbolic variables:

```
from pykeops.torch import LazyTensor
x_i = LazyTensor( x[:,None,:] )  # x_i.shape = (1e6, 1, 3)
y_j = LazyTensor( y[None,:,:] )  # y_j.shape = ( 1, 2e6,3)
```

Example in Python (LazyTensor)

```
## Perform symbolic large-scale computations

# 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()

## Get the result (computations on GPU are done here)
a_i = K_ij.sum(dim=1)  # Genuine torch.cuda.FloatTensor
# a_i.shape = (1e6, 1)

## KeOps supports autograd!
g_x = torch.autograd.grad((a_i ** 2).sum(), [x])
```

Example in R

```
formula = "Sum_Reduction(Exp(lambda*SqNorm2(x-y))*beta, 1)"
args = c("x=Vi(3)", "y=Vj(3)", "beta=Vj(3)", "lambda=Pm(1)")

op <- keops_kernel(formula, args) # compilation

# data and paramters
nx = 1000
ny = 1500
x <- matrix(runif(nx*3), ncol=nx)
y <- matrix(runif(ny*3), ncol=ny)
beta <- matrix(runif(ny*3), ncol=ny)
lambda <- as.matrix(5)

# computation
res <- op(args=list(x, y, beta, lambda), nx=ncol(x), ny=ncol(y))
```

Example in R

- beta version
- Gradient computation not available for the moment
- Specific branch `rkeops`

```
git clone https://github.com/getkeops/keops
git checkout rkeops
```

- See `rkeops/REAMD.md` for install instructions

More features (not presented today)

- `PyKeOps (Numpy)`, `KeOpsLab`: formula gradient computation
- `PyKeOps (PyTorch)`: automatic differentiation engine (compatible with PyTorch autograd)
- In the near future
  - gradient computation and lazy evaluation in `Rkeops`
  - possible to add new generic operations upon request (responsive user support via Github issues)
  - and more...

Conclusion

Take-home message

KeOps:

Seamless Kernel Operations...

→ write formulas with simple matrix operations (Python, Matlab, R)

...on GPU...

→ fast computations

...with auto-differentiation...

→ automatic gradient computation

...and without memory overflows

→ implementation with tiling for efficient memory usage on GPU

Thank you for you attention

Questions?

http://www.kernel-operations.io/keops/index.html

https://github.com/getkeops/keops