Espaces à noyau II

B. Charlier (Univ. Montpellier, IMAG & ICM) joint work with J. Feydy (ENS, Paris et CMLA & ENS Paris-Saclay) and Joan Glaunes (Univ. Paris Descartes, MAP5)

IMAG - Montpellier

26 mars 2018

1. Rappels et compléments de l'épisode précédent
   1.1. Application: spline
   1.2. Application: Distance entre mesures

2. Motivation et cadre général
   2.1. Motivations
   2.2. Cadre de calculs

3. KeOps et GPU
   3.1. Architecture
   3.2. Un exemple

4. Utiliser KeOps en pratique
   4.1. From sources
   4.2. Performances

5. Différentiation automatique

6. Pytorch bindings
   6.1. An example

Rappels

Definition (Noyau)

Soit X un ensemble non vide. Une fonction $k : X \times X \to \mathbb{R}$ est un noyau si il existe un espace de Hilbert H et une application $\phi : X \to H$ tels que

$$\langle \phi(x), \phi(y) \rangle = k(x, y), \quad \forall x, y \in X.$$

Definition (RKHS)

Soit H un $\mathbb{R}$-espace de Hilbert de fonctions définies sur $X \neq \emptyset$. On dit que $k : X \times X \to \mathbb{R}$ est un noyau auto-reproduisant et H est un RKHS si

1. $\forall x \in X$ on a $k(\cdot, x) \in H$
2. $\forall x \in X$ on a $\langle f, k(\cdot, x) \rangle_H = f(x)$

Exemple: $H^1(\mathbb{R})$, espace engendré par les RBF gaussien/Cauchy/Student...

Interpolation spline en 1d

Soit $(x_1, \cdots, x_N) \in \mathbb{R}$ et $(\lambda_1, \cdots, \lambda_N) \in \mathbb{R}$. On fixe un espace fonctionnel V et on cherche une solution de

$$\begin{cases} & \min_{f \in V} \|f\|_V \\ s.c. & f(x_i) = \lambda_i \end{cases} \tag{1}$$

Solution de l'interpolation spline en 1d

En fait, c'est bien moins compliqué qu'on peut ne le penser: la solution est à chercher dans un espace de dimension finie N. On pose :

$$V_0^{\perp} = \{K(\cdot, x_i), i = 1, \cdots, N\}$$

Proposition

Si il existe une solution $\hat{v} \in V$ de (1) alors:

1. $\hat{v} \in V_0^{\perp}$
2. *si $\hat{v} \in V_0^{\perp}$ est solution de (1) restreinte à V_0 (en changeant V par V_0) alors $\hat{v}$ est aussi une solution du problème (1) initial.*

Mesures discrètes

Soient $\alpha = \sum_{i=1}^{N} \alpha_i \delta_{x_i}$ et $\beta = \sum_{j=1}^{M} \beta_j \delta_{y_j}$ où

- les positions $x_1, \cdots, x_N, y_1, \cdots, y_M \in \mathbb{R}^2$
- les poids $\sum_i \alpha_i = \sum_j \beta_j = 1$.

Normes duales

- Variation totale:

$$d_{TV}(\alpha,\beta) = \sup_{\|f\|_\infty \le 1} \int f d\alpha - \int f d\beta = \sup_{\|f\|_\infty \le 1} \int f d(\alpha-\beta).$$

  Pb quand les supports sont disjoints.

- Normes à noyau: soit V un RKHS (+ hypothèses) de noyau k, on pose:

$$d_V(\alpha,\beta) = \sup_{\|f\|_V \le 1} \int f d(\alpha-\beta).$$

  Sur les mesures discrètes cela donne:

$$d_V(\alpha,\beta) = \left\langle \sum_i \alpha_i k(\cdot,x_i), \sum_j \beta_j k(\cdot,y_j) \right\rangle = \sum_i \sum_j \alpha_i \beta_j k(x_i,y_j)$$

Distance OT

Definition

$\mathrm{U}(\alpha,\beta) = \left\{\Pi \in \mathbb{R}^{NM} \mid \Pi\mathbb{1}_n = \alpha \text{ et } \Pi^t\mathbb{1}_M = \beta\right\}$

Etant donnée une fonction coût $c : \mathbb{R}^2 \times \mathbb{R}^2 \to \mathbb{R}^+$. La distance du transport optimale est

$$d_{OT}(\alpha,\beta) = \min_{\Pi \in U(\alpha,\beta)} \langle \Pi, C\rangle = \sum_i \sum_j \pi_{i,j} c(x_i, y_j)$$

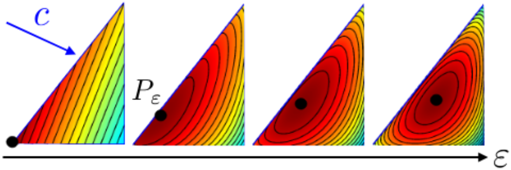
où $C = [c(x_i, y_j)]_{i=1,\dots,N, j=1,\dots,M}$
Programme linéaire d'optimisation.

OT et régularisation entropique

Régularise le problème

$$d_{OT}^{\varepsilon}(\alpha, \beta) = \langle \Pi^{\varepsilon}, C \rangle \text{ where } \Pi^{\varepsilon} = \min_{\Pi \in U(\alpha,\beta)} \langle \Pi, C \rangle - \varepsilon h(\Pi)$$

où $h(\Pi) = \sum_i \sum_j \pi_{i,j}(\log \pi_{i,j} - 1)$
Programme convexe:



La solution s'écrit $\Pi^{\varepsilon} = \mathrm{Diag}(u)[e^{-\varepsilon C}]\,\mathrm{Diag}(v)$

1. Rappels et compléments de l'épisode précédent
   1.1. Application: spline
   1.2. Application: Distance entre mesures

2. Motivation et cadre général
   2.1. Motivations
   2.2. Cadre de calculs

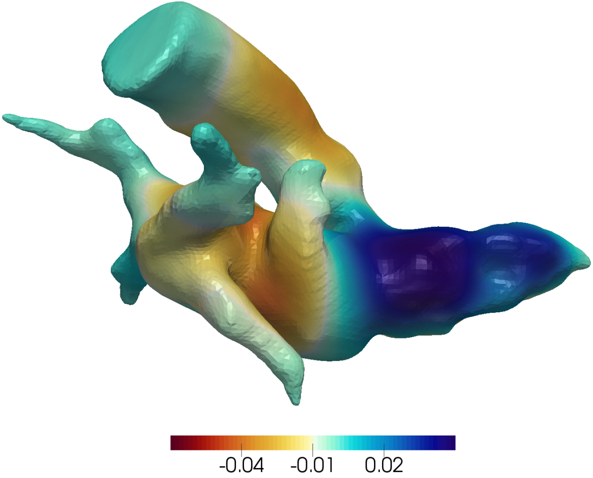
3. KeOps et GPU
   3.1. Architecture
   3.2. Un exemple

4. Utiliser KeOps en pratique
   4.1. From sources
   4.2. Performances

5. Différentiation automatique

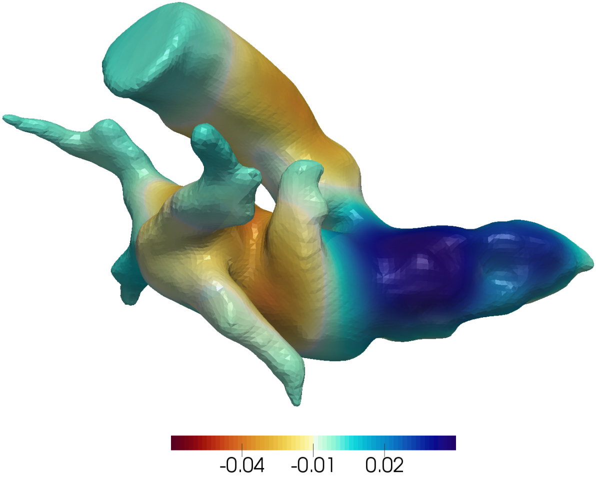
6. Pytorch bindings
   6.1. An example

Motivations: LDDMM



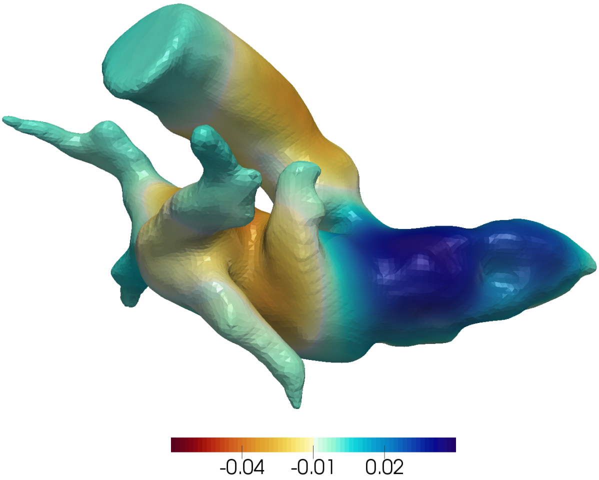
Data courtesy of C. Chnafa, S. Mendez, F. Nicoud (Université de Montpellier)

Motivations: LDDMM



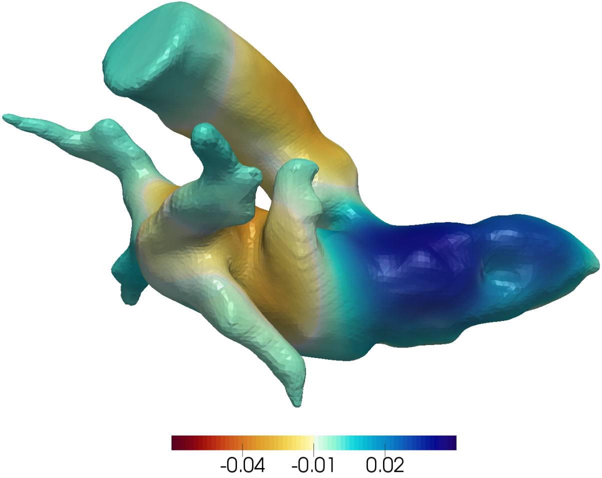
Data courtesy of C. Chnafa, S. Mendez, F. Nicoud (Université de Montpellier)

Motivations: LDDMM



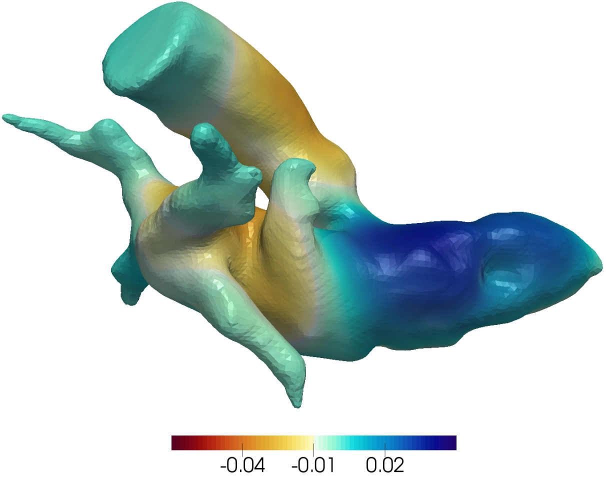
Data courtesy of C. Chnafa, S. Mendez, F. Nicoud (Université de Montpellier)

Motivations: LDDMM



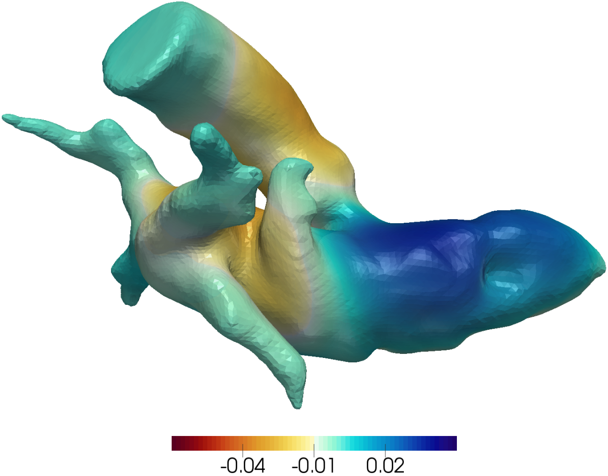
Data courtesy of C. Chnafa, S. Mendez, F. Nicoud (Université de Montpellier)

Motivations: LDDMM



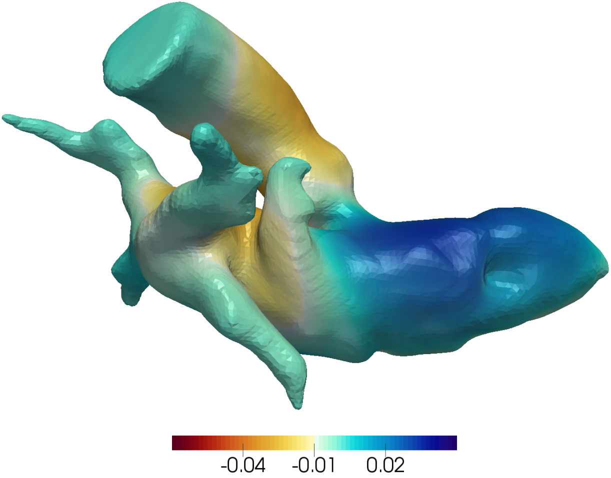
Data courtesy of C. Chnafa, S. Mendez, F. Nicoud (Université de Montpellier)

Motivations: LDDMM



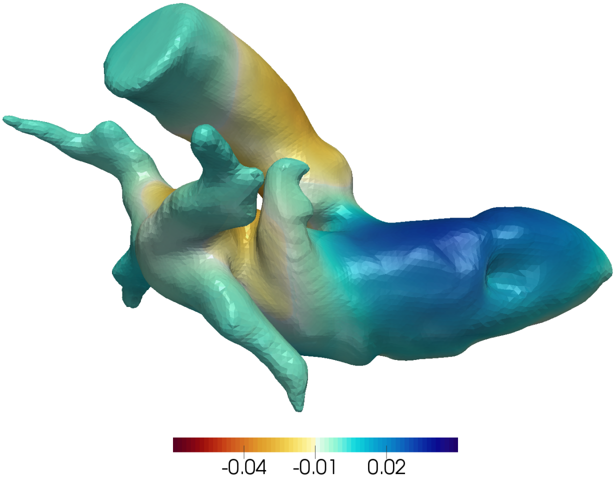
Data courtesy of C. Chnafa, S. Mendez, F. Nicoud (Université de Montpellier)

Motivations: LDDMM



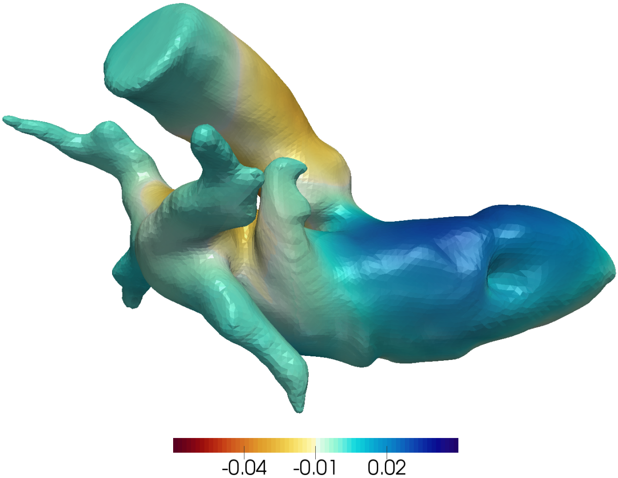
Data courtesy of C. Chnafa, S. Mendez, F. Nicoud (Université de Montpellier)

Motivations: LDDMM



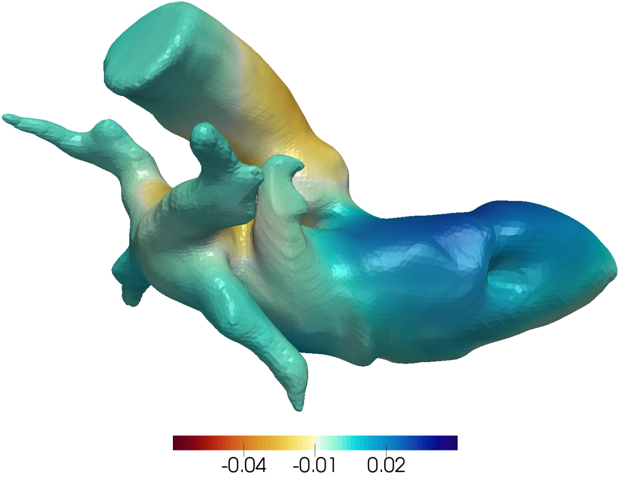
Data courtesy of C. Chnafa, S. Mendez, F. Nicoud (Université de Montpellier)

Motivations: LDDMM



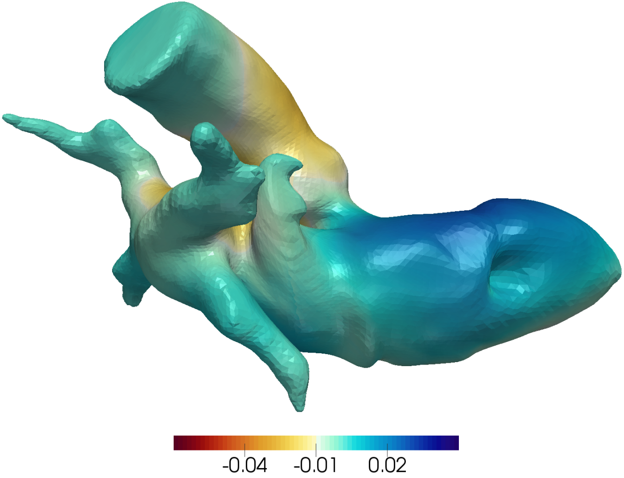
Data courtesy of C. Chnafa, S. Mendez, F. Nicoud (Université de Montpellier)

Motivations: LDDMM



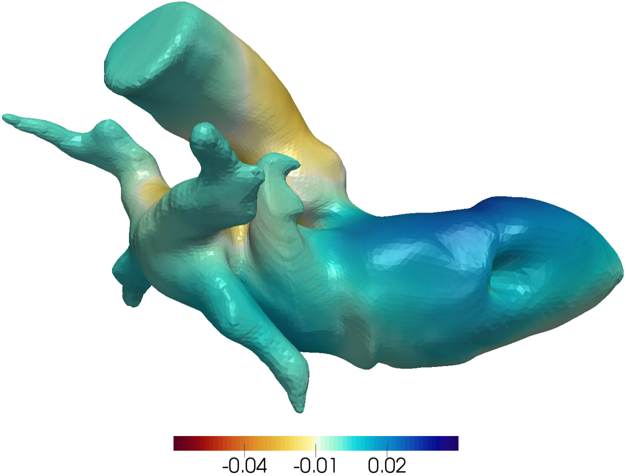
Data courtesy of C. Chnafa, S. Mendez, F. Nicoud (Université de Montpellier)

Motivations: LDDMM



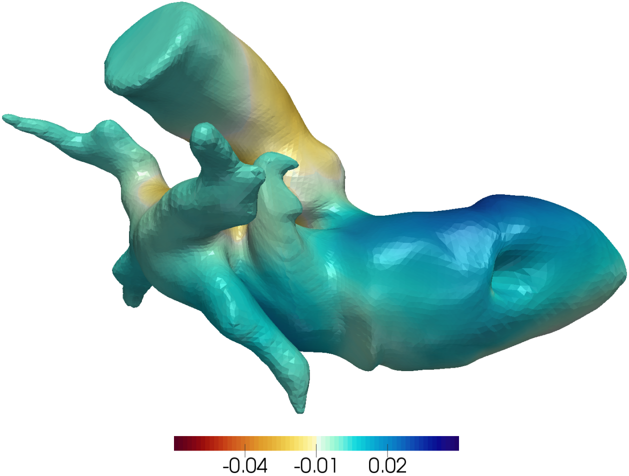
Data courtesy of C. Chnafa, S. Mendez, F. Nicoud (Université de Montpellier)

Motivations: LDDMM



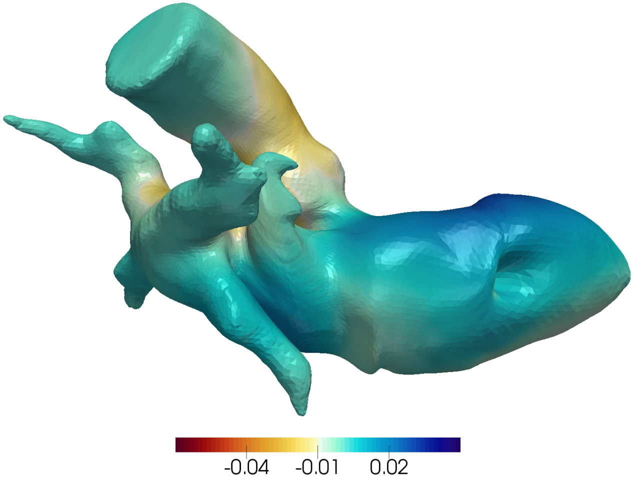
Data courtesy of C. Chnafa, S. Mendez, F. Nicoud (Université de Montpellier)

Motivations: LDDMM



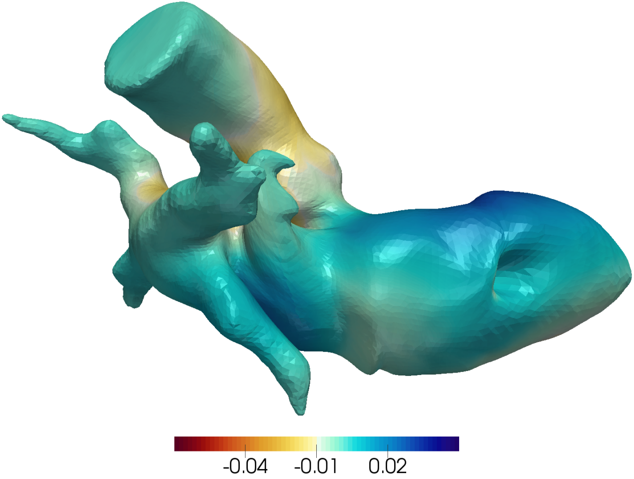
Data courtesy of C. Chnafa, S. Mendez, F. Nicoud (Université de Montpellier)

Motivations: LDDMM



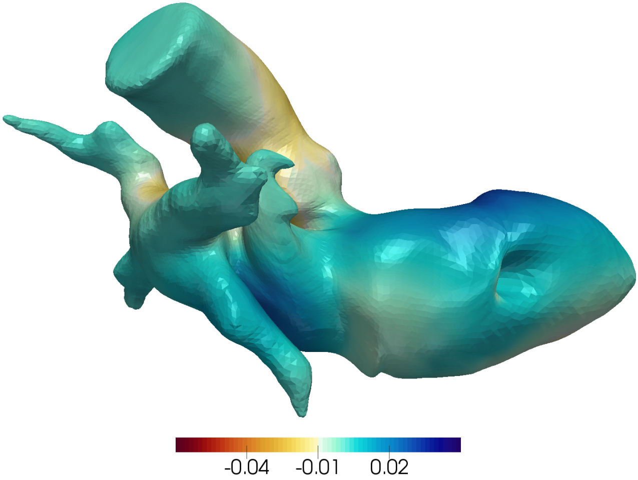
Data courtesy of C. Chnafa, S. Mendez, F. Nicoud (Université de Montpellier)

Motivations: LDDMM



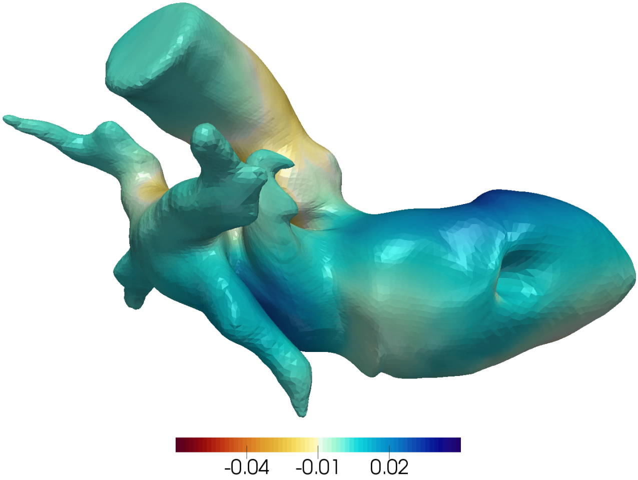
Data courtesy of C. Chnafa, S. Mendez, F. Nicoud (Université de Montpellier)

Motivations: LDDMM



Data courtesy of C. Chnafa, S. Mendez, F. Nicoud (Université de Montpellier)

Motivations: LDDMM



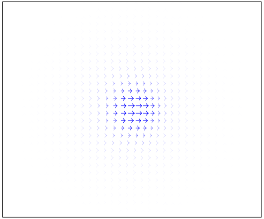
Data courtesy of C. Chnafa, S. Mendez, F. Nicoud (Université de Montpellier)

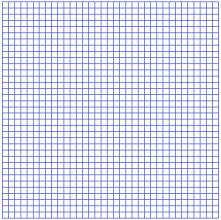
Motivation : LDDMM

Deformation = flow of time varying smooth vector field

- **Flow :** let $v = (v_t)_{t\in[0,1]} \in V$ be a time dependant vectors field of $\mathbb{R}^3$. Let $\phi : [0,1] \times \mathbb{R}^3 \to \mathbb{R}^3$:

$$\begin{cases} \dot{\phi}_t(x) = v_t(\phi_t(x)) \\ \phi_0(x) = x. \end{cases} \qquad t \in [0,1] \text{ and } x \in \mathbb{R}^3$$





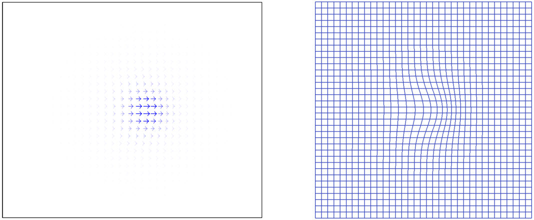
$t = 0$

Motivation : LDDMM

Deformation = flow of time varying smooth vector field

- **Flow :** let $v = (v_t)_{t\in[0,1]} \in V$ be a time dependant vectors field of $\mathbb{R}^3$. Let $\phi : [0,1] \times \mathbb{R}^3 \to \mathbb{R}^3$:

$$\begin{cases} \dot{\phi}_t(x) = v_t(\phi_t(x)) \\ \phi_0(x) = x. \end{cases} \qquad t \in [0,1] \text{ and } x \in \mathbb{R}^3$$



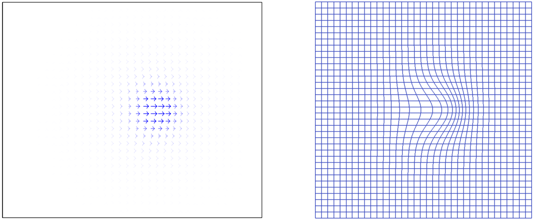
$t = 1/5$

Motivation : LDDMM

Deformation = flow of time varying smooth vector field

- **Flow :** let $v = (v_t)_{t \in [0,1]} \in V$ be a time dependant vectors field of $\mathbb{R}^3$. Let $\phi : [0,1] \times \mathbb{R}^3 \to \mathbb{R}^3$:

$$\begin{cases} \dot{\phi}_t(x) = v_t(\phi_t(x)) \\ \phi_0(x) = x. \end{cases} \qquad t \in [0,1] \text{ and } x \in \mathbb{R}^3$$



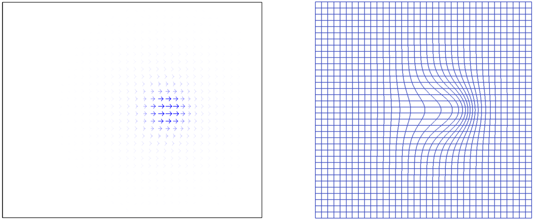
$t = 2/5$

Motivation : LDDMM

Deformation = flow of time varying smooth vector field

- **Flow :** let $v = (v_t)_{t\in[0,1]} \in V$ be a time dependant vectors field of $\mathbb{R}^3$. Let $\phi : [0,1] \times \mathbb{R}^3 \to \mathbb{R}^3$:

$$\begin{cases} \dot{\phi}_t(x) = v_t(\phi_t(x)) \\ \phi_0(x) = x. \end{cases} \qquad t \in [0,1] \text{ and } x \in \mathbb{R}^3$$



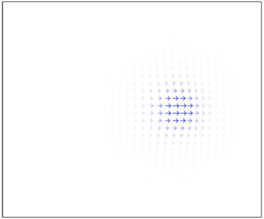
$t = 3/5$

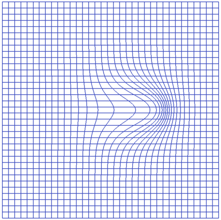
Motivation : LDDMM

Deformation = flow of time varying smooth vector field

- **Flow :** let $v = (v_t)_{t\in[0,1]} \in V$ be a time dependant vectors field of $\mathbb{R}^3$. Let $\phi : [0,1] \times \mathbb{R}^3 \to \mathbb{R}^3$:

$$\begin{cases} \dot{\phi}_t(x) = v_t(\phi_t(x)) \\ \phi_0(x) = x. \end{cases} \qquad t \in [0,1] \text{ and } x \in \mathbb{R}^3$$





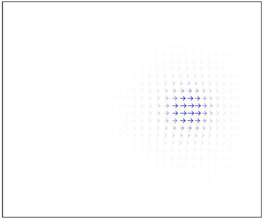
$t = 4/5$

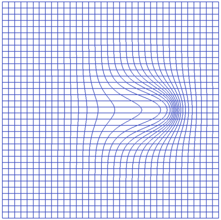
Motivation : LDDMM

Deformation = flow of time varying smooth vector field

- **Flow :** let $v = (v_t)_{t\in[0,1]} \in V$ be a time dependant vectors field of $\mathbb{R}^3$. Let $\phi : [0,1] \times \mathbb{R}^3 \to \mathbb{R}^3$:

$$\begin{cases} \dot{\phi}_t(x) = v_t(\phi_t(x)) \\ \phi_0(x) = x. \end{cases} \qquad t \in [0,1] \text{ and } x \in \mathbb{R}^3$$





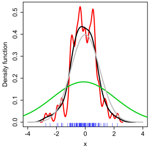
$t = 1$

Motivation : LDDMM

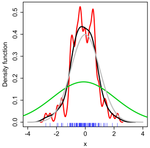
Algorithms typically rely on:

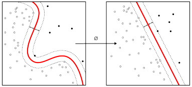
- $H(x,p) = \frac{1}{2}\left\langle p, K_{q,q}p\right\rangle_2 = \frac{1}{2}\sum_{i,j} k(q_i,q_j)\ \left\langle p_i,p_j\right\rangle_2 = \frac{1}{2}\sum_{i,j} \exp(\|q_i - q_j\|^2/\sigma^2)\ \left\langle p_i,p_j\right\rangle_2$
- $\nabla_q H$, $\nabla_p H$

Les espaces à noyaux en statistique et apprentissage

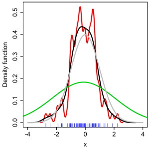
- Kernel density estimation : 

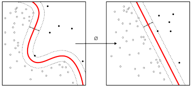
Les espaces à noyaux en statistique et apprentissage

- Kernel density estimation : 

- SVM : classification/Regression 

Les espaces à noyaux en statistique et apprentissage

- Kernel density estimation : 

- SVM : classification/Regression 

- Kernel embeddings to compare distribution :  $\sim P$  $\sim Q$

Un schéma général de calculs

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

- Multiplication matricielle :

$$\gamma = \left[\sum_{j} A_{i,j} \beta_j \right]_i$$

Un schéma général de calculs

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

- Multiplication matricielle :

$$\gamma = \left[\sum_j A_{i,j} \beta_j \right]_i$$

- Convolution simple :

$$\gamma = \left[\sum_j K_\sigma(x_i, y_j) \beta_j \right]_i$$

Un schéma général de calculs

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

- Multiplication matricielle :

$$\gamma = \left[\sum_j A_{i,j} \beta_j \right]_i$$

- Convolution simple :

$$\gamma = \left[\sum_j K_\sigma(x_i, y_j) \beta_j \right]_i$$

- Des opérations plus compliquées :

$$\gamma_i = \left[\sum_j K_1(x_i, y_j) K_2(u_i, v_j) \langle \alpha_i, \beta_j \rangle \right]$$

Un schéma général de calculs

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

- En général :

$$\gamma = \left[\sum_j F(\sigma_1, \cdots, \sigma_\ell, X_i^1, \cdots, X_i^k, Y_j^1, \cdots, Y_j^m) \right]_i$$

Un schéma général de calculs

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

- En général :

$$\gamma = \left[\sum_j F(\sigma_1, \cdots, \sigma_\ell, X_i^1, \cdots, X_i^k, Y_j^1, \cdots, Y_j^m) \right]_i$$

- ... encore plus général :

$$\gamma = \left[\mathop{\Huge\ast}_j F(\sigma_1, \cdots, \sigma_\ell, X_i^1, \cdots, X_i^k, Y_j^1, \cdots, Y_j^m) \right]_i$$

1. Rappels et compléments de l'épisode précédent
1.1. Application: spline
1.2. Application: Distance entre mesures

2. Motivation et cadre général
2.1. Motivations
2.2. Cadre de calculs

3. KeOps et GPU
3.1. Architecture
3.2. Un exemple

4. Utiliser KeOps en pratique
4.1. From sources
4.2. Performances

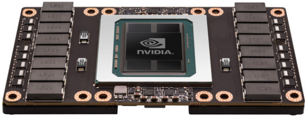
5. Différentiation automatique

6. Pytorch bindings
6.1. An example

(Nvidia) GPU Market

- Gamers : 800euros

- Scientific computing: $2000 - 5000$euros



- Under the hood :

Une architecture massivement parallèle

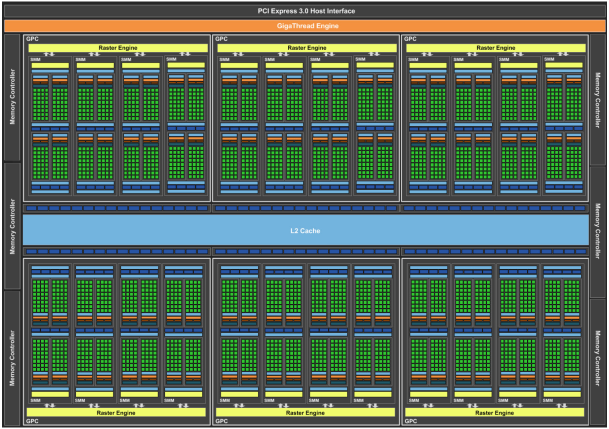


Figure: A GPU architecture is built around a scalable array of multithreaded *Streaming Multiprocessors* (*SM*s). In a SM, each single processor is called a *thread* and is able to execute an independent set of instructions.

MatMult : A first naive implementation

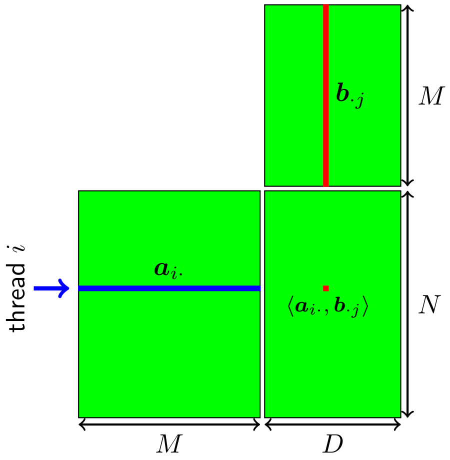


Figure: A matrix multiplication AB (where $A \in \mathbb{R}^{N\times M}$ and $B \in \mathbb{R}^{M\times D}$) is a set of ND scalar products. Each thread computes D scalar products.

MatMult : A first naive implementation

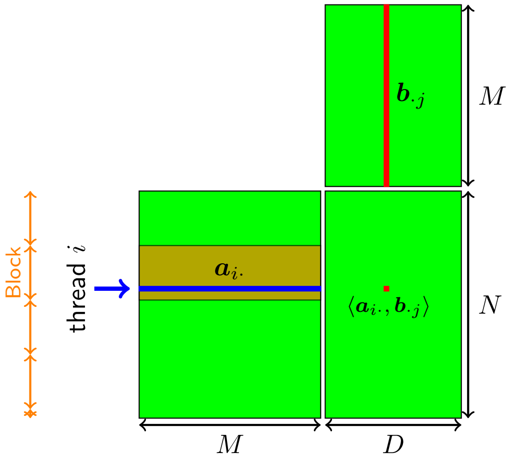
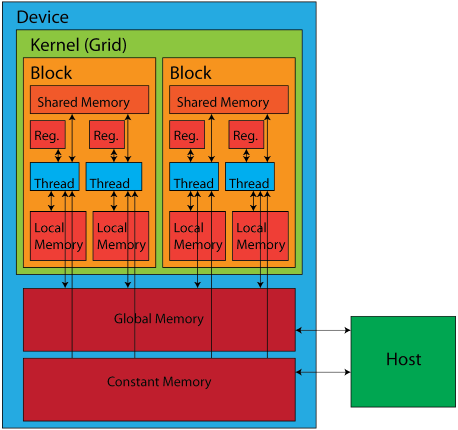


Figure: A matrix multiplication AB (where $A \in \mathbb{R}^{N \times M}$ and $B \in \mathbb{R}^{M \times D}$) is a set of ND scalar products. Each thread computes D scalar products.

Memory managment



- The data are initially stored on the host and should be transfer to the device to be treated (bottleneck).
- Different kinds of memory (métaphore culinaire?)
- A smart use of the shared memory is often the key to provide an

MatMult : Tiled implementation

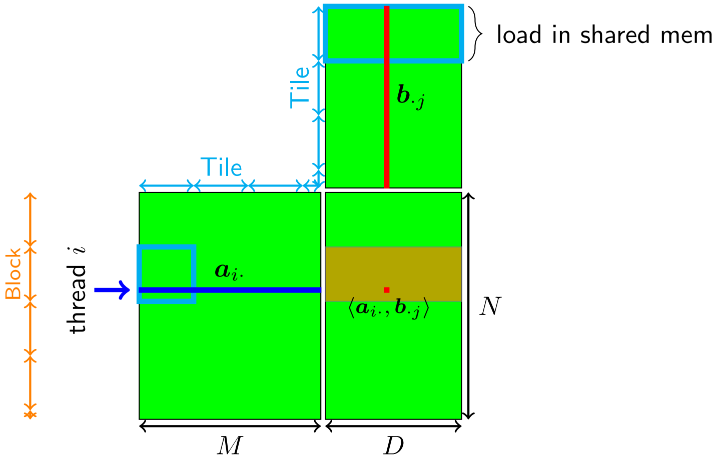


Figure: Divided job into tiles and use the shared memory whithin a block.

MatMult : Tiled implementation

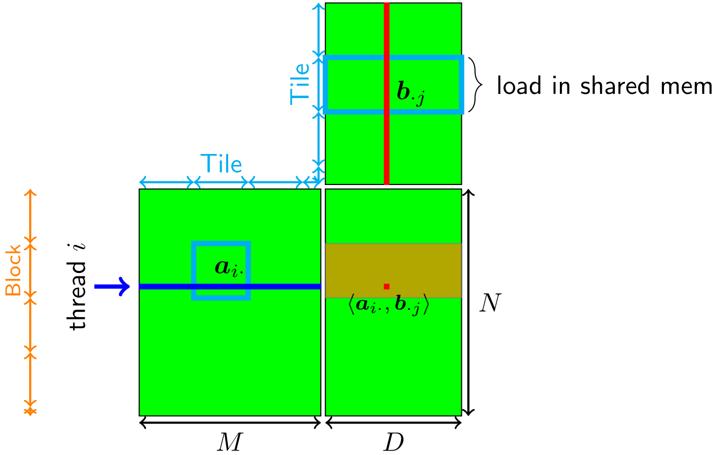


Figure: Divided job into tiles and use the shared memory whithin a block.

MatMult : Tiled implementation

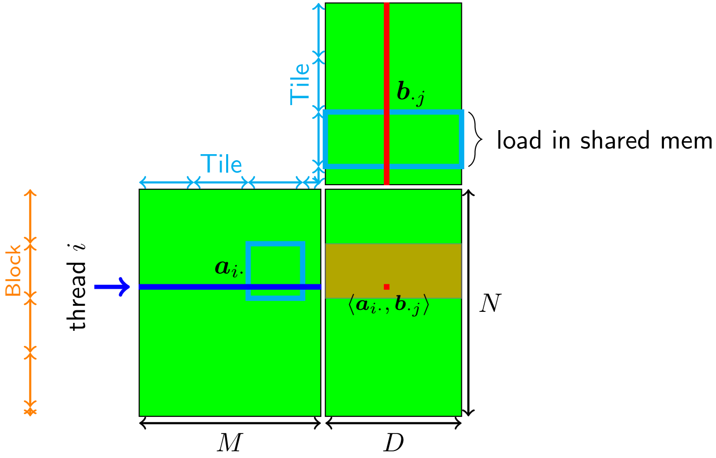


Figure: Divided job into tiles and use the shared memory whithin a block.

1. Rappels et compléments de l'épisode précédent
1.1. Application: spline
1.2. Application: Distance entre mesures

2. Motivation et cadre général
2.1. Motivations
2.2. Cadre de calculs

3. KeOps et GPU
3.1. Architecture
3.2. Un exemple

4. Utiliser KeOps en pratique
4.1. From sources
4.2. Performances

5. Différentiation automatique

6. Pytorch bindings
6.1. An example

Calculer une convolution gaussienne

Soit $s \in \mathbb{R}$, $x \in \mathbb{R}^{N\times 3}$, $y \in \mathbb{R}^{M\times 3}$, $b \in \mathbb{R}^{M\times 6}$ on cherche à calculer

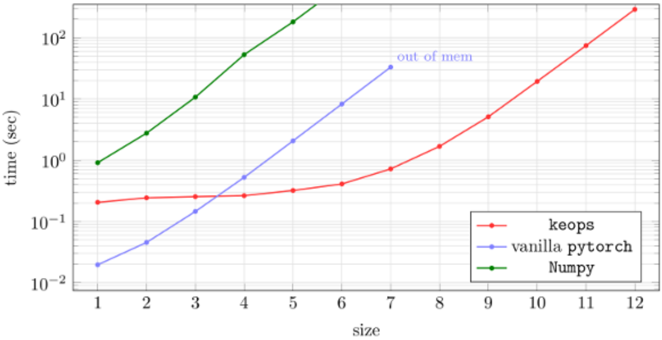
$$\gamma_j = \sum_j \exp(-s\|x_i - y_j\|^2) b_j.$$

- depuis python ou matlab

```
import pykeops
my_conv = Genred("Exp(-s*SqNorm2(x-y))*b",
                         "s = Pm(1)",
                         "x = Vx(3)",
                         "y = Vy(3)",
                         "b = Vy(6)")
gamma = my_conv(p,x,y,b) # calcule sur le GPU
```

- Formule possible : exp, log, $+$, $*$, $/$, $\min$, $\max$, $\mathrm{LogSumExp}$, Kinv, $\ldots$

Benchmark



Time needed to compute 200 Gaussian kernel products on a Tesla P100 GPU.
At size i we have $M = 200 \times 2^i$ and $N = 300 \times 2^i$.

1. Rappels et compléments de l'épisode précédent
1.1. Application: spline
1.2. Application: Distance entre mesures

2. Motivation et cadre général
2.1. Motivations
2.2. Cadre de calculs

3. KeOps et GPU
3.1. Architecture
3.2. Un exemple

4. Utiliser KeOps en pratique
4.1. From sources
4.2. Performances

5. Différentiation automatique

6. Pytorch bindings
6.1. An example

Différences finies?

Let $F : \mathbb{R}^n \to \mathbb{R}$ be a smooth function. Then:

$$\nabla F(x_0) \quad = \quad \begin{pmatrix} \partial_{x^1} F(x_0) \\ \partial_{x^2} F(x_0) \\ \vdots \\ \partial_{x^n} F(x_0) \end{pmatrix} \quad \simeq \quad \frac{1}{\delta t} \begin{pmatrix} F(x_0 + \delta t \cdot (1, 0, \dots, 0)) - F(x_0) \\ F(x_0 + \delta t \cdot (0, 1, \dots, 0)) - F(x_0) \\ \vdots \\ F(x_0 + \delta t \cdot (0, 0, \dots, 1)) - F(x_0) \end{pmatrix}.$$

Différences finies?

Let $F : \mathbb{R}^n \to \mathbb{R}$ be a smooth function. Then:

$$\nabla F(x_0) \quad = \quad \begin{pmatrix} \partial_{x^1} F(x_0) \\ \partial_{x^2} F(x_0) \\ \vdots \\ \partial_{x^n} F(x_0) \end{pmatrix} \quad \simeq \quad \frac{1}{\delta t} \begin{pmatrix} F(x_0 + \delta t \cdot (1, 0, \ldots, 0)) - F(x_0) \\ F(x_0 + \delta t \cdot (0, 1, \ldots, 0)) - F(x_0) \\ \vdots \\ F(x_0 + \delta t \cdot (0, 0, \ldots, 1)) - F(x_0) \end{pmatrix}.$$

$\Longrightarrow$ costs **(N+1) evaluations of** F, which is poor.

Reverse AD

Let $F : (X, \langle \cdot, \cdot \rangle_X) \to (Y, \langle \cdot, \cdot \rangle_Y)$ be a smooth map between be two Hilbert spaces.

Reverse AD

Let $F : (X, \langle \cdot, \cdot \rangle_X) \to (Y, \langle \cdot, \cdot \rangle_Y)$ be a smooth map between be two Hilbert spaces.

- the adjoint of the differential is $(\mathrm{d}_x F)^*(x_0) : \alpha \in Y^* \to \beta \in X^*$. Riesz representation theorem gives a map

$$\partial_x F\,(x_0) : a \in Y \to b \in X$$

called generalized **gradient**.

Reverse AD

Let $F : (X, \langle \cdot, \cdot \rangle_X) \to (Y, \langle \cdot, \cdot \rangle_Y)$ be a smooth map between be two Hilbert spaces.

- the adjoint of the differential is $(\mathrm{d}_x F)^*(x_0) : \alpha \in Y^* \to \beta \in X^*$. Riesz representation theorem gives a map

$$\partial_x F\,(x_0) : a \,\in Y \;\; \to b \,\in X$$

called generalized **gradient**.

- If $X = \mathbb{R}^n$, $Y = \mathbb{R}$ endowed with the Euclidean metric,

$$\mathcal{M}_{\partial_x F(x_0)} = \begin{pmatrix} \partial_{x^1} F(x_0) \\ \partial_{x^2} F(x_0) \\ \vdots \\ \partial_{x^n} F(x_0) \end{pmatrix}$$

Reverse AD = backpropagating = chain rules

$$\begin{array}{ccccccccccccccc}
\mathbb{R}^n & & E_0 & & E_1 & & E_2 & & \cdots & & E_p & & \mathbb{R} \\
\Cup & & \Cup & & \Cup & & \Cup & & & & \Cup & & \Cup \\
x_0 & \xrightarrow{\textbf{input}} & x_0 & \xrightarrow{F_1} & x_1 & \xrightarrow{F_2} & x_2 & \xrightarrow{\cdots} & \cdots & \xrightarrow{F_p} & x_p & \xrightarrow{\textbf{output}} & F(x_0)
\end{array}$$

Backpropagating through a computational graph requires:

$$\begin{array}{cccc} F_i & : & E_{i-1} & \to & E_i \\ & & x & \mapsto & F_i(x) \end{array} \qquad \text{(Forward)}$$

and

(Backward)

encoded as **computer programs**.

Reverse AD = backpropagating = chain rules

$$
\begin{tikzcd}
\mathbb{R}^n & E_0 & E_1 & E_2 & \cdots & E_p & \mathbb{R} \\
\cup & \cup & \cup & \cup & & \cup & \cup \\
x_0 \arrow[r, "\textbf{input}"] & x_0 \arrow[r, "F_1"] \arrow[dd] & x_1 \arrow[r, "F_2"] \arrow[dd] & x_2 \arrow[r, "\cdots"] \arrow[dd] & \cdots \arrow[r, "F_p"] \arrow[dd] & x_p \arrow[r, "\textbf{output}"] & F(x_0) \\
\\
\partial_x F(x_0)\cdot 1 & x_0^* \arrow[l, "\textbf{output}"'] & x_1^* \arrow[l, "\partial_x F_1"'] & x_2^* \arrow[l, "\partial_x F_2"'] & \cdots \arrow[l, "\cdots"'] & x_p^* \arrow[l, "\partial_x F_p"'] & 1 \arrow[l, "\textbf{input}"'] \\
\cap & \cap & \cap & \cap & & \cap & \cap \\
\mathbb{R}^n & E_0 & E_1 & E_2 & \cdots & E_p & \mathbb{R}
\end{tikzcd}
$$

Backpropagating through a computational graph requires:

$$
\begin{array}{cccc}
F_i & : & E_{i-1} & \to & E_i \\
 & & x & \mapsto & F_i(x)
\end{array}
\qquad \text{(Forward)}
$$

and

$$
\begin{array}{cccc}
\partial_x F_i & : & E_{i-1} \times E_i & \to & E_{i-1} \\
 & & (x_0, a) & \mapsto & \partial_x F_i(x_0) \cdot a
\end{array}
\qquad \text{(Backward)}
$$

encoded as **computer programs**.

Reverse AD = backpropagating = chain rules

$$
\begin{tikzcd}
\mathbb{R}^n & E_0 & E_1 & E_2 & \cdots & E_p & \mathbb{R} \\
x_0 \arrow[u, phantom, sloped, "\in"] \arrow[r, "\textbf{input}"] & x_0 \arrow[u, phantom, sloped, "\in"] \arrow[r, "F_1"] \arrow[d, blue] & x_1 \arrow[u, phantom, sloped, "\in"] \arrow[r, "F_2"] \arrow[d, blue] & x_2 \arrow[u, phantom, sloped, "\in"] \arrow[r, "\cdots"] \arrow[d, blue] & \cdots \arrow[r, "F_p"] \arrow[d, blue] & x_p \arrow[u, phantom, sloped, "\in"] \arrow[r, "\textbf{output}"] & F(x_0) \arrow[u, phantom, sloped, "\in"] \\
\partial_x F(x_0) \cdot 1 \arrow[d, phantom, sloped, "\in"] & x_0^* \arrow[l, red, "\textbf{output}"'] \arrow[d, phantom, sloped, "\in"] & x_1^* \arrow[l, "\partial_x F_1"'] \arrow[d, phantom, sloped, "\in"] & x_2^* \arrow[l, "\partial_x F_2"'] \arrow[d, phantom, sloped, "\in"] & \cdots \arrow[l, "\cdots"'] & x_p^* \arrow[l, "\partial_x F_p"'] \arrow[d, phantom, sloped, "\in"] & 1 \arrow[l, green, "\textbf{input}"'] \arrow[d, phantom, sloped, "\in"] \\
\mathbb{R}^n & E_0 & E_1 & E_2 & \cdots & E_p & \mathbb{R}
\end{tikzcd}
$$

1. Starting from $x_0 \in \mathbb{R}^n = E_0$, compute and **store in memory** the successive vectors $x_i \in E_i$. The last one, $x_p = F(x_0) \in \mathbb{R}$.
2. Starting from the canonical value of $x_p^* = 1 \in \mathbb{R}$, compute the successive *dual* vectors

$$x_i^* \;=\; \partial_x F_{i+1}(x_i) \cdot x_{i+1}^*. \tag{2}$$

The last one, $x_0^* = \nabla F(x_0) = \partial_x F(x_0) \cdot 1 \in \mathbb{R}^n$, is the gradient.

1. Rappels et compléments de l'épisode précédent
1.1. Application: spline
1.2. Application: Distance entre mesures

2. Motivation et cadre général
2.1. Motivations
2.2. Cadre de calculs

3. KeOps et GPU
3.1. Architecture
3.2. Un exemple

4. Utiliser KeOps en pratique
4.1. From sources
4.2. Performances

5. Différentiation automatique

6. Pytorch bindings
6.1. An example

Computing an Hamiltonian

Soit $\boldsymbol{p}, \boldsymbol{q} \in \mathbb{R}^{N \times D}$, on cherche à calculer

$$H(\boldsymbol{q}, \boldsymbol{p}) = \boldsymbol{p}^t K_{\boldsymbol{q},\boldsymbol{q}} \boldsymbol{p} = \sum_i \sum_j p_i^t \underbrace{K_s(q_i, q_j)}_{\exp(-\|q_i - q_j\|^2/s)} p_j$$

peut être interprété comme une energie cinétique ou une norme d'un RKHS.

```python
import torch
# Choose the storage place for our data : CPU (host) or GPU (cu) memory.
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

Computing an Hamiltonian

Soit $\boldsymbol{p}, \boldsymbol{q} \in \mathbb{R}^{N \times D}$, on cherche à calculer

$$H(\boldsymbol{q}, \boldsymbol{p}) = \boldsymbol{p}^t K_{\boldsymbol{q},\boldsymbol{q}} \boldsymbol{p} = \sum_i \sum_j p_i^t \underbrace{K_s(q_i, q_j)}_{\exp(-\|q_i - q_j\|^2/s)} p_j$$

peut être interprété comme une energie cinétique ou une norme d'un RKHS.

```
import torch
# Choose the storage place for our data : CPU (host) or GPU (cu) memory.
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
#
N = 1000; D = 3 ; # Clouds of 1,000 points in 3D
# Generate arbitrary arrays on the CPU or GPU:
q = torch.randn( N,D  , requires_grad=True , device=device)
p = torch.randn( N,D  , requires_grad=True , device=device)
s = torch.tensor([0.5], requires_grad=False, device=device)
```

Computing the Hamiltonian

```
# Actual computations.
q_i  = q.unsqueeze(1) # shape (N,D) -> (N,1,D)
q_j  = q.unsqueeze(0) # shape (N,D) -> (1,N,D)
```

Computing the Hamiltonian

```
# Actual computations.
q_i  = q.unsqueeze(1) # shape (N,D) -> (N,1,D)
q_j  = q.unsqueeze(0) # shape (N,D) -> (1,N,D)
sqd  = torch.sum( (q_i - q_j)**2 , 2 ) # |q_i-q_j|^2
```

Computing the Hamiltonian

```
# Actual computations.
q_i  = q.unsqueeze(1) # shape (N,D) -> (N,1,D)
q_j  = q.unsqueeze(0) # shape (N,D) -> (1,N,D)
sqd  = torch.sum( (q_i - q_j)**2 , 2 ) # |q_i-q_j|^2
K_qq = torch.exp( - sqd / (s**2) )     # Gaussian kernel
```

Computing the Hamiltonian

```
# Actual computations.
q_i  = q.unsqueeze(1) # shape (N,D) -> (N,1,D)
q_j  = q.unsqueeze(0) # shape (N,D) -> (1,N,D)
sqd  = torch.sum( (q_i - q_j)**2 , 2 ) # |q_i-q_j|^2
K_qq = torch.exp( - sqd / (s**2) )     # Gaussian kernel
v    = K_qq @ p # matrix mult. (N,N)@(N,D) = (N,D)
```

Computing the Hamiltonian

```
# Actual computations.
q_i  = q.unsqueeze(1) # shape (N,D) -> (N,1,D)
q_j  = q.unsqueeze(0) # shape (N,D) -> (1,N,D)
sqd  = torch.sum( (q_i - q_j)**2 , 2 ) # |q_i-q_j|^2
K_qq = torch.exp( - sqd / (s**2) )     # Gaussian kernel
v    = K_qq @ p # matrix mult. (N,N)@(N,D) = (N,D)
#
# Finally, compute the Hamiltonian H(q,p): .5*<p,v>
H    = .5 * torch.dot( p.view(-1), v.view(-1) )
```

Computing the Hamiltonian

```
# Actual computations.
q_i  = q.unsqueeze(1) # shape (N,D) -> (N,1,D)
q_j  = q.unsqueeze(0) # shape (N,D) -> (1,N,D)
sqd  = torch.sum( (q_i - q_j)**2 , 2 ) # |q_i-q_j|^2
K_qq = torch.exp( - sqd / (s**2) )     # Gaussian kernel
v    = K_qq @ p # matrix mult. (N,N)@(N,D) = (N,D)
#
# Finally, compute the Hamiltonian H(q,p): .5*<p,v>
H    = .5 * torch.dot( p.view(-1), v.view(-1) )
#
# Automatic differentiation is straitghtforward
[dq,dp] = torch.autograd.grad( H, [q,p], 1.)
```

Computing the Hamiltonian

```
# Actual computations.
q_i  = q.unsqueeze(1) # shape (N,D) -> (N,1,D)
q_j  = q.unsqueeze(0) # shape (N,D) -> (1,N,D)
sqd  = torch.sum( (q_i - q_j)**2 , 2 ) # |q_i-q_j|^2
K_qq = torch.exp( - sqd / (s**2) )     # Gaussian kernel
v    = K_qq @ p # matrix mult. (N,N)@(N,D) = (N,D)
#
# Finally, compute the Hamiltonian H(q,p): .5*<p,v>
H    = .5 * torch.dot( p.view(-1), v.view(-1) )
#
# Automatic differentiation is straitghtforward
[dq,dp] = torch.autograd.grad( H, [q,p], 1.)
```

```
RuntimeError: cuda runtime error (2) : out of memory at
/opt/conda/.../THCStorage.cu:66
```

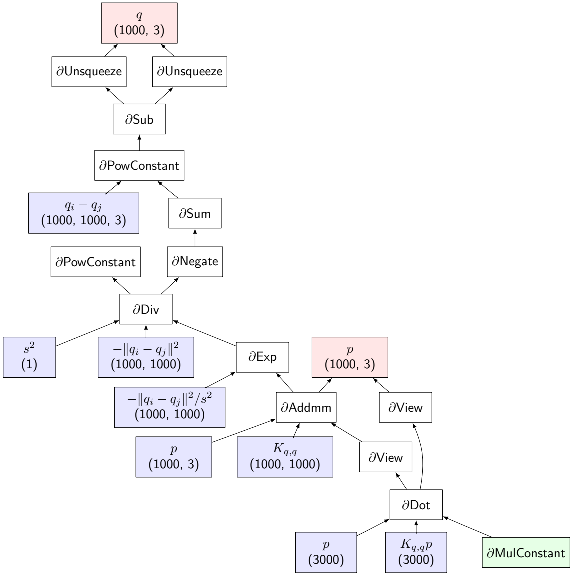
Computing the Hamiltonian

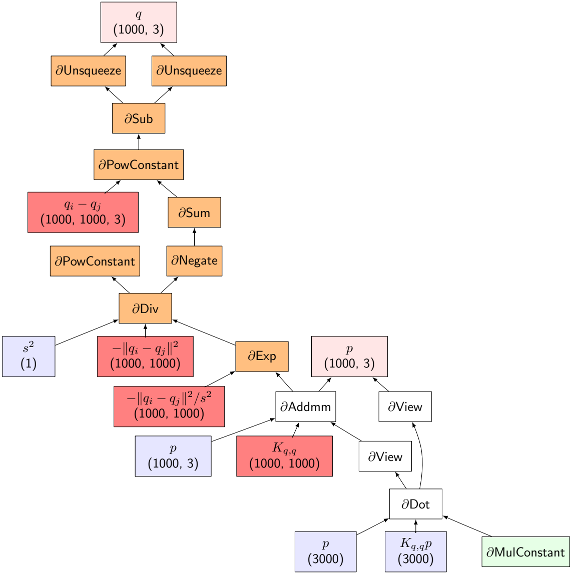
```python
# Actual computations.
q_i  = q.unsqueeze(1) # shape (N,D) -> (N,1,D)
q_j  = q.unsqueeze(0) # shape (N,D) -> (1,N,D)
sqd  = torch.sum( (q_i - q_j)**2 , 2 ) # |q_i-q_j|^2
K_qq = torch.exp( - sqd / (s**2) )     # Gaussian kernel
v    = K_qq @ p # matrix mult. (N,N)@(N,D) = (N,D)
#
# Finally, compute the Hamiltonian H(q,p): .5*<p,v>
H    = .5 * torch.dot( p.view(-1), v.view(-1) )
#
# Automatic differentiation is straitghtforward
[dq,dp] = torch.autograd.grad( H, [q,p], 1.)
```

RuntimeError: cuda runtime error (2) : out of memory at /opt/conda/.../THCStorage.cu:66

```python
# Display -- see next figure.
make_dot(H, {'q':q, 'p':p, 's':s}).render(view=True)
```

6 Pytorch bindings: An example





q
(1000, 3)
∂Unsqueeze
∂Unsqueeze
∂Sub
∂PowConstant
$q_i - q_j$
(1000, 1000, 3)
∂Sum
∂PowConstant
∂Negate
∂Div
s^2
(1)
$-\|q_i - q_j\|^2$
(1000, 1000)
∂Exp
p
(1000, 3)
$-\|q_i - q_j\|^2/s^2$
(1000, 1000)
∂Addmm
∂View
p
(1000, 3)
$K_{q,q}$
(1000, 1000)
∂View
∂Dot
p
(3000)
$K_{q,q}p$
(3000)
∂MulConstant

Our contribution

- Créer une classe python/pytorch contenant une méthode qui calcule F et autre méthode qui calcule sa dérivée.

Our contribution

- Créer une classe python/pytorch contenant une méthode qui calcule F et autre méthode qui calcule sa dérivée.

```
class GenericKernelProduct(torch.autograd.Function):
    ...
    def forward(ctx,..., formula, ..., *args):
        cuda_conv_generic(formula, result, *args,... ) # Inplace CUDA rout
        return result
    ...
    def backward(ctx, G):
        ...
        formula_g   = "Grad("+ formula +","+ var +","+ eta +")"
        ...
        grads.append( genconv( formula_g, ..., *args_g )) # generate CUDA
        return   (..., *grads )
```

- La différentiation automatique du code cuda est faite pendant la compilation.

Our contribution

- Créer une classe python/pytorch contenant une méthode qui calcule F et autre méthode qui calcule sa dérivée.

```
class GenericKernelProduct(torch.autograd.Function):
    ...
    def forward(ctx,..., formula, ..., *args):
        cuda_conv_generic(formula, result, *args,... ) # Inplace CUDA rout
        return result
    ...
    def backward(ctx, G):
        ...
        formula_g   = "Grad("+ formula +","+ var +","+ eta +")"
        ...
        grads.append( genconv( formula_g, ..., *args_g )) # generate CUDA
        return   (..., *grads )
```

- La différentiation automatique du code cuda est faite pendant la compilation.
- Cette classe est “appelable” par le différentiateur automatique de pytorch dans n'importe quel calcul.

KeOps rocks!

```
import pykeops
# Compute the kernel convolution with keops
kernelproduct = KernelProduct.apply
v = kernelproduct(s, q, q, p, "gaussian")
# Then, compute the Hamiltonian H(q,p): .5*<p,v>
H = .5 * torch.dot( p.view(-1), v.view(-1) )
#
# Automatic differentiation works
[dq,dp] = torch.autograd.grad( H, [q,p], 1.)
```

KeOps rocks!

```
import pykeops
# Compute the kernel convolution with keops
kernelproduct = KernelProduct.apply
v = kernelproduct(s, q, q, p, "gaussian")
# Then, compute the Hamiltonian H(q,p): .5*<p,v>
H = .5 * torch.dot( p.view(-1), v.view(-1) )
#
# Automatic differentiation works
[dq,dp] = torch.autograd.grad( H, [q,p], 1.)
```