An efficient kernel product for autodiff libraries

With applications to measure transport

Benjamin Charlier, Jean Feydy, Joan Alexis Glaunès, Alain Trouvé

November 13, 2017; GFSW03, Isaac Newton Institute

Overview

What is PyTorch? Facebook

Deep Learning only → Memory overflows

Overview

What is PyTorch? Facebook

Deep Learning only → Memory overflows

How do we fix it? + B. Charlier, J. Glaunès

`libkp` provides efficient CUDA routines,
wrapped in a `KernelProduct` operator.

What is PyTorch? Facebook

Deep Learning only $\rightarrow$ Memory overflows

How do we fix it? + B. Charlier, J. Glaunès

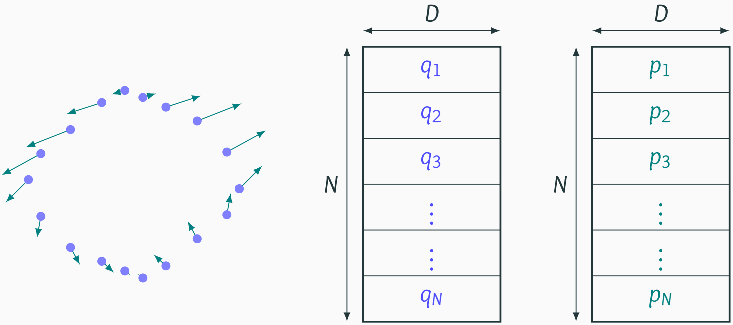
`libkp` provides efficient CUDA routines,

wrapped in a `KernelProduct` operator.

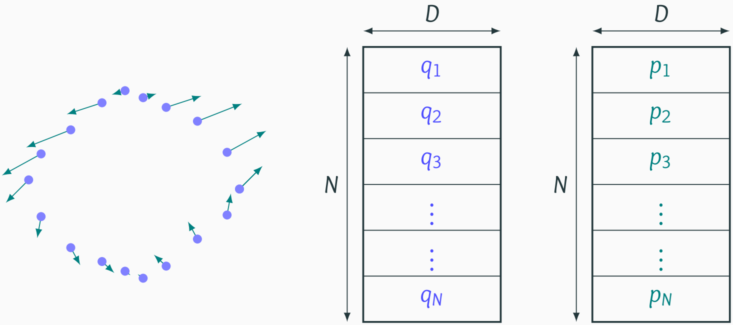
Where can this bring us? + A. Trouvé

Normalized Hamiltonian setting.

Shape analysis pipelines from a practical point of view



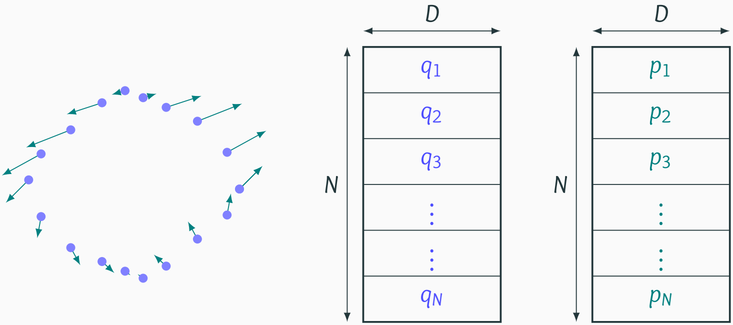
Shape analysis pipelines from a practical point of view



Algorithms typically rely on:

- $H(q, p) \;=\; \frac{1}{2}\langle p, K_q p\rangle_2 \;=\; \frac{1}{2}\sum_{i,j} k(q_i, q_j)\,\langle p_i, p_j\rangle_2$

Shape analysis pipelines from a practical point of view



Algorithms typically rely on:

- $H(q,p) \;=\; \frac{1}{2}\langle p, K_q p\rangle_2 \;=\; \frac{1}{2}\sum_{i,j} k(q_i,q_j)\,\langle p_i,p_j\rangle_2$
- $\nabla_q H, \nabla_p H$

How do we compute a gradient?

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

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

How do we compute a gradient?

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

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

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

How do we compute a gradient?

Let $(X, \langle \cdot , \cdot \rangle_X)$ and $(Y, \langle \cdot , \cdot \rangle_Y)$ be two Hilbert spaces.
Let $F : X \to Y$ be a smooth map. Then, we say that:

$(\mathrm{d}_x F)^*(x_0) : \alpha \in Y^* \to \beta \in X^*$ is the adjoint of the differential.

How do we compute a gradient?

Let $(X, \langle \cdot , \cdot \rangle_X)$ and $(Y, \langle \cdot , \cdot \rangle_Y)$ be two Hilbert spaces.
Let $F : X \to Y$ be a smooth map. Then, we say that:

$(\mathrm{d}_x F)^*(x_0) : \alpha \in Y^* \to \beta \in X^*$ is the adjoint of the differential.
$\partial_x F \quad (x_0) : a \in Y \to b \in X$ is the **gradient**.

How do we compute a gradient?

Let $(X, \langle \cdot, \cdot \rangle_X)$ and $(Y, \langle \cdot, \cdot \rangle_Y)$ be two Hilbert spaces.
Let $F : X \to Y$ be a smooth map. Then, we say that:

$(\mathrm{d}_x F)^*(x_0) : \alpha \in Y^* \to \beta \in X^*$ is the adjoint of the differential.
$\partial_x F \quad (x_0) : a \in Y \to b \in X$ is the **gradient**.

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

$$\partial_x F(x_0) \quad = \quad (\mathrm{d}_x F(x_0))^{\mathsf{T}} \quad = \quad \begin{pmatrix} \partial_{x^1} F(x_0) \\ \partial_{x^2} F(x_0) \\ \vdots \\ \partial_{x^n} F(x_0) \end{pmatrix}$$

What do you need to compute a gradient?

Backpropagating through a computational graph requires:

$$\begin{array}{rccl} F_i & : & E_{i-1} & \rightarrow & E_i \\ & & x & \mapsto & F_i(x) \end{array} \quad (1)$$

and

$$\begin{array}{rccl} \partial_x F_i & : & E_{i-1} \times E_i & \rightarrow & E_{i-1} \\ & & (x_0, a) & \mapsto & \partial_x F_i(x_0) \cdot a \end{array} \quad (2)$$

encoded as **computer programs.**

What do you need to compute a gradient?

Backpropagating through a computational graph requires:

$$F_i \quad : \quad \begin{array}{ccc} E_{i-1} & \rightarrow & E_i \\ x & \mapsto & F_i(x) \end{array} \tag{1}$$

and

$$\partial_x F_i \quad : \quad \begin{array}{ccc} E_{i-1} \times E_i & \rightarrow & E_{i-1} \\ (x_0, a) & \mapsto & \partial_x F_i(x_0) \cdot a \end{array} \tag{2}$$

encoded as **computer programs.**

This is what PyTorch is all about.

Computing the Hamiltonian

```
import torch          # GPU + autodiff library
# With PyTorch, using the GPU is that simple:
use_gpu  = torch.cuda.is_available()
dtype    = torch.cuda.FloatTensor if use_gpu \
           else torch.FloatTensor
```

Computing the Hamiltonian

```
import torch          # GPU + autodiff library
# With PyTorch, using the GPU is that simple:
use_gpu  = torch.cuda.is_available()
dtype    = torch.cuda.FloatTensor if use_gpu \
           else torch.FloatTensor
#
N = 1000; D = 3 ; # Clouds of 1,000 points in 3D
# Generate arbitrary arrays on the CPU or GPU:
q = torch.from_numpy( ... ).type(dtype).view(N,D)
p = torch.from_numpy( ... ).type(dtype).view(N,D)
s = torch.Tensor(  [2.5]  ).type(dtype)
```

Computing the Hamiltonian

```
import torch          # GPU + autodiff library
# With PyTorch, using the GPU is that simple:
use_gpu  = torch.cuda.is_available()
dtype    = torch.cuda.FloatTensor if use_gpu \
           else torch.FloatTensor
#
N = 1000; D = 3 ; # Clouds of 1,000 points in 3D
# Generate arbitrary arrays on the CPU or GPU:
q = torch.from_numpy( ... ).type(dtype).view(N,D)
p = torch.from_numpy( ... ).type(dtype).view(N,D)
s = torch.Tensor(  [2.5]  ).type(dtype)
#
# Wrap them into "autodiff" graph nodes. In this demo,
# we won't try to fine tune the deformation model, so
# we do not need any derivative with respect to s:
q = torch.autograd.Variable( q, requires_grad = True )
p = torch.autograd.Variable( p, requires_grad = True )
s = torch.autograd.Variable( s, requires_grad = False)
```

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.)
```

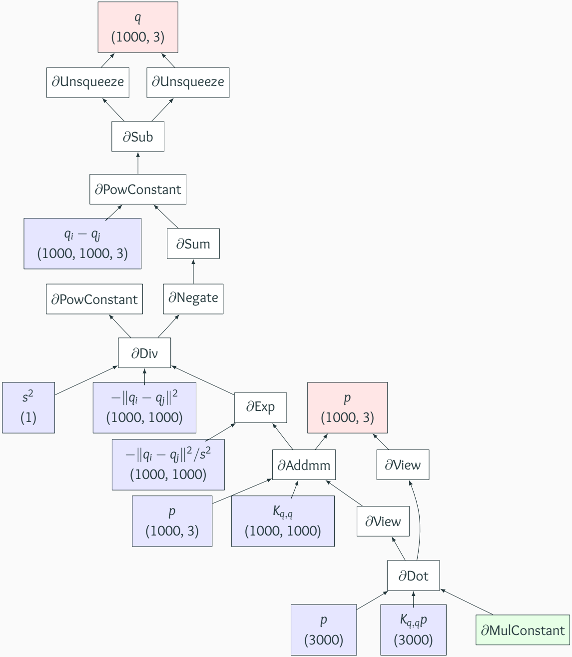
```
RuntimeError: cuda runtime error (2) : out of memory at
/opt/conda/.../THCStorage.cu:66
```

Computing the Hamiltonian

```
# 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.)
```

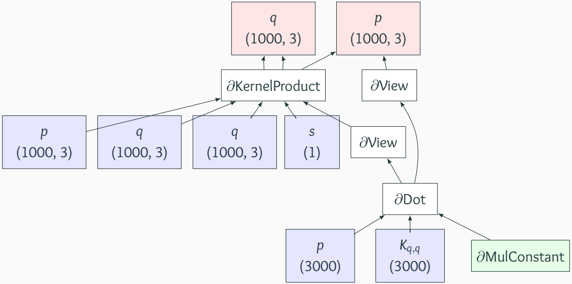
RuntimeError: cuda runtime error (2) : out of memory at /opt/conda/.../THCStorage.cu:66

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


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

```
# Compute the kernel convolution
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) )
```



How does one compute

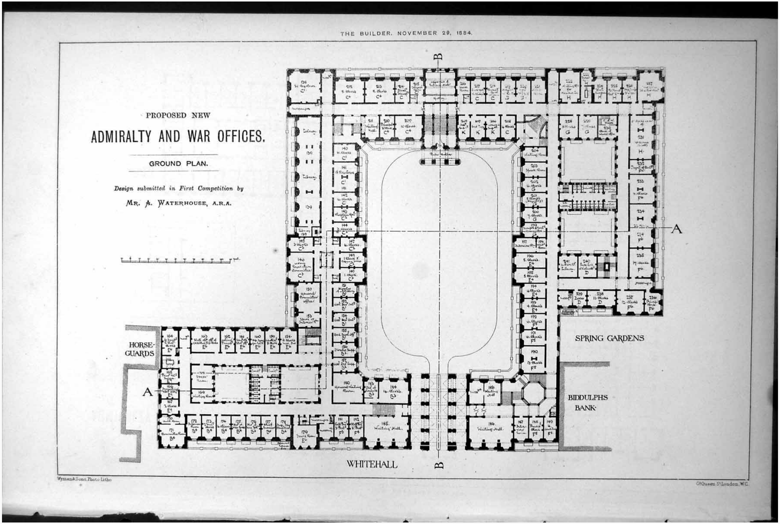
$$g_i = \sum_j k(x_i - y_j)\, b_j$$

on the GPU?

Memory management in CUDA

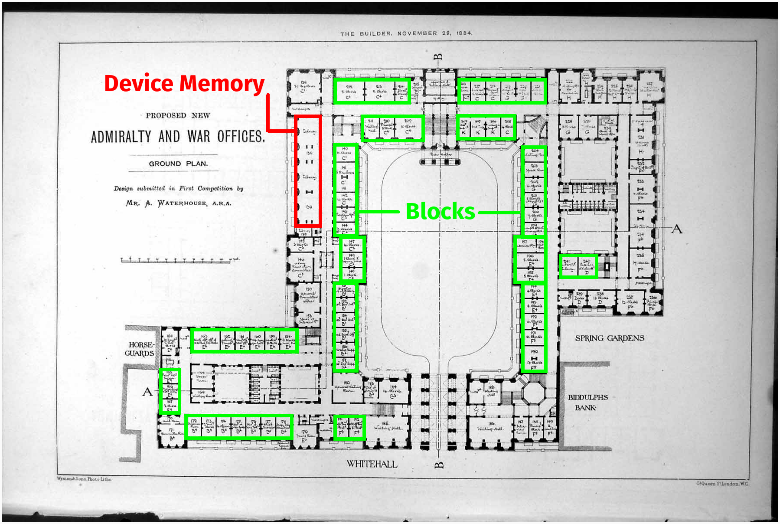
Leonhard Euler: the perfect XVIIIth century **CPU**.

Memory management in CUDA



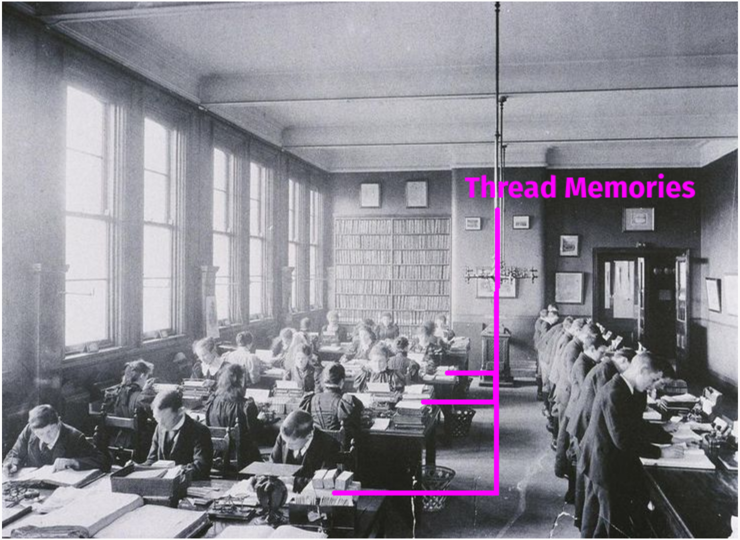
1884: a new age of **parallel computing**.

Memory management in CUDA



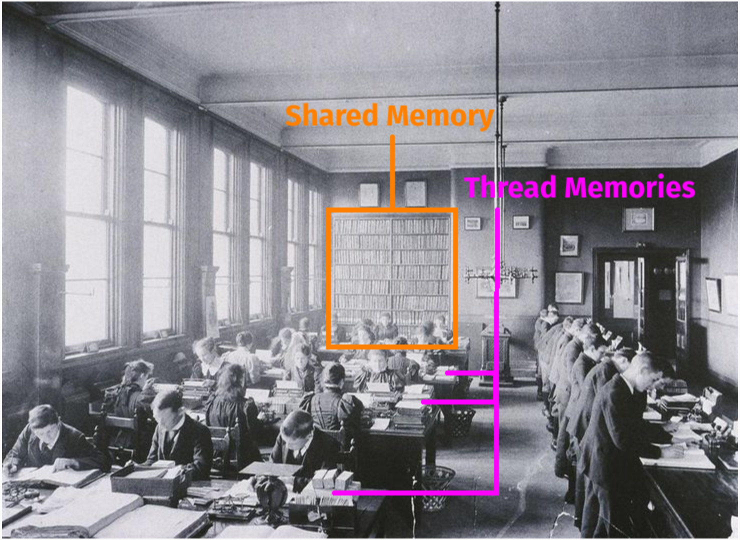
1884: a new age of **parallel computing**.

Memory management in CUDA



1884: inside a computing block.

Memory management in CUDA



1884: inside a computing block.

What is actually written

KernelProd CUDA program executed by a `Block`		
Input	:	in `GM` : `x`, `y`, `b`
		in `TM` : `BlockId`, `ThreadId`
Parameter	:	$k : x^2 \mapsto \exp(-\|x\|^2/\sigma^2)$, etc.
Output	:	$(g_i) = \sum_j k(x_i - y_j) \cdot b_j$

What is actually written

KernelProd CUDA program executed by a `Block`

Input : in `GM` : `x`, `y`, `b`

in `TM` : `BlockId`, `ThreadId`

Parameter : $k : x^2 \mapsto \exp(-\|x\|^2/\sigma^2)$, etc.

Output : $(g_i) = \sum_j k(x_i - y_j) \cdot b_j$

1: `i = BlockId` $\cdot$ BlockSize `+ ThreadId`

What is actually written

KernelProd CUDA program executed by a `Block`

Input : in `GM` : `x`, `y`, `b`

in `TM` : `BlockId`, `ThreadId`

Parameter : $k : x^2 \mapsto \exp(-\|x\|^2/\sigma^2)$, etc.

Output : $(g_i) = \sum_j k(x_i - y_j) \cdot b_j$

```
i = BlockId · BlockSize + ThreadId
g[i] = [0,...,0]; load x[i] in TM
```

What is actually written

KernelProd CUDA program executed by a `Block`

Input : in `GM` : `x`, `y`, `b`

in `TM` : `BlockId`, `ThreadId`

Parameter : $k : x^2 \mapsto \exp(-\|x\|^2/\sigma^2)$, etc.

Output : $(g_i) = \sum_j k(x_i - y_j) \cdot b_j$

```
i = BlockId · BlockSize + ThreadId
g[i] = [0,...,0]; load x[i] in TM
for (J=0; J<M; J+=BlockSize) do
```

What is actually written

KernelProd CUDA program executed by a `Block`

Input : in `GM` : `x`, `y`, `b`

in `TM` : `BlockId`, `ThreadId`

Parameter : $k : x^2 \mapsto \exp(-\|x\|^2/\sigma^2)$, etc.

Output : $(g_i) = \sum_j k(x_i - y_j) \cdot b_j$

`i = BlockId` · BlockSize + `ThreadId`

`g[i] = [0,...,0]`; load `x[i]` in `TM`

for `(J=0; J<M; J+=BlockSize)` **do**

  Load in parallel (`j` ∈ `[J,J+BlockSize[`) in `SM` : `y[j]`, `b[j]`

What is actually written

KernelProd CUDA program executed by a `Block`

Input : in `GM` : `x`, `y`, `b`

in `TM` : `BlockId`, `ThreadId`

Parameter : $k : x^2 \mapsto \exp(-\|x\|^2/\sigma^2)$, etc.

Output : $(g_i) = \sum_j k(x_i - y_j) \cdot b_j$

```
i = BlockId · BlockSize + ThreadId
g[i] = [0,...,0]; load x[i] in TM
for (J=0; J<M; J+=BlockSize) do
    Load in parallel (j ∈ [J,J+BlockSize[) in SM : y[j], b[j]
    for (j=J; j<J+BlockSize; j++) do
```

What is actually written

KernelProd CUDA program executed by a `Block`

Input : in `GM` : `x`, `y`, `b`

in `TM` : `BlockId`, `ThreadId`

Parameter : $k : x^2 \mapsto \exp(-\|x\|^2/\sigma^2)$, etc.

Output : $(g_i) = \sum_j k(x_i - y_j) \cdot b_j$

```
i = BlockId · BlockSize + ThreadId
g[i] = [0,...,0]; load x[i] in TM
for (J=0; J<M; J+=BlockSize) do
    Load in parallel (j ∈ [J,J+BlockSize[) in SM : y[j], b[j]
    for (j=J; j<J+BlockSize; j++) do
        r2     = sum( (x[i] - y[j] )**2 )
        g[i] += k(r2) * b[j]
```

What is actually written

KernelProd CUDA program executed by a `Block`

Input : in `GM` : `x`, `y`, `b`
in `TM` : `BlockId`, `ThreadId`

Parameter : $k : x^2 \mapsto \exp(-\|x\|^2/\sigma^2)$, etc.

Output : $(g_i) = \sum_j k(x_i - y_j) \cdot b_j$

```
i = BlockId · BlockSize + ThreadId
g[i] = [0,...,0]; load x[i] in TM
for (J=0; J<M; J+=BlockSize) do
    Load in parallel (j ∈ [J,J+BlockSize[) in SM : y[j], b[j]
    for (j=J; j<J+BlockSize; j++) do
        r2    = sum( (x[i] - y[j] )**2 )
        g[i] += k(r2) * b[j]
Push g[i] back in the GM
```

A flexible and scalable development framework

PyTorch + `libkp`:

- No need to write backwards anymore
- No more memory overflows

A flexible and scalable development framework

PyTorch + `libkp`:

- No need to write backwards anymore
- No more memory overflows

$\Longrightarrow$ Try out your ideas within a couple of hours!

Normalizing Hamiltonians to get mass awareness

Why is the LDDMM framework so popular?

In the computational sense, it is the cheapest way to build regularizing metrics on point clouds:

- Hamilton's theorem ($g_q \longrightarrow K_q$)
- The current availability of GPUs (parallelism)

Is LDDMM the missing link between Monge and Procustes?

If k is a smooth enough kernel function, it defines a RKHS norm

$$\|v\|_k^2 \;=\; \langle\, v, k^{(-1)} \star v \rangle \;=\; \int_{\mathbb{R}^d} \frac{1}{\widehat{k}(\omega)} |\widehat{v}(\omega)|^2 \, \mathrm{d}\omega, \tag{3}$$

$$\|p\|_k^{*2} \;=\; \langle\, p, k \star p \rangle \,. \tag{4}$$

Is LDDMM the missing link between Monge and Procustes?

If k is a smooth enough kernel function, it defines a RKHS norm

$$\|v\|_k^2 \;=\; \langle\, v, k^{(-1)} \star v \rangle \;=\; \int_{\mathbb{R}^d} \frac{1}{\widehat{k}(\omega)} |\widehat{v}(\omega)|^2 \, \mathrm{d}\omega, \tag{3}$$

$$\|p\|_k^{*2} \;=\; \langle\, p, k \star p \rangle \,. \tag{4}$$

The **Reduction Principle**:

$$(q_t, p_t) \longleftrightarrow \varphi_t \;\text{ where } \varphi_t \text{ is } k\text{-smooth} \tag{5}$$

Is LDDMM the missing link between Monge and Procustes?

If k is a smooth enough kernel function, it defines a RKHS norm

$$\|v\|_k^2 \;=\; \langle\, v, k^{(-1)} \star v \rangle \;=\; \int_{\mathbb{R}^d} \frac{1}{\widehat{k}(\omega)} |\widehat{v}(\omega)|^2 \,\mathrm{d}\omega, \tag{3}$$

$$\|p\|_k^{*2} \;=\; \langle\, p, k \star p \rangle\,. \tag{4}$$

The **Reduction Principle**:

$$(q_t, p_t) \longleftrightarrow \varphi_t \;\text{ where } \varphi_t \text{ is } k\text{-smooth} \tag{5}$$

On landmarks, one could be tempted to believe that:

$$\text{Wasserstein} \quad (\sigma = 0) \xrightarrow{\sigma++} \|\cdot\|_k \xrightarrow{\sigma++} (\sigma = \infty) \text{ Translations}$$

Recap of today's presentation

Contributions:

- Flexible and scalable development tools.

Recap of today's presentation

Contributions:

- Flexible and scalable development tools.
- Implement easily metrics which are not right-invariant.

Recap of today's presentation

Contributions:

- Flexible and scalable development tools.
- Implement easily metrics which are not right-invariant.

Schedule:

Today: Detailed PDF report + Git (Numpy, PyTorch, Matlab and R bindings), see `www.math.ens.fr/~feydy/research.html`

1st of Dec.: Full report on Arxiv.

1st of Jan. : `libkp` completed: Currents, Varifolds, etc.

1st of Apr.? Full Normalized Hamiltonians paper.

Thank you for your attention.