Abstractions performantes pour cartes graphiques

Séminaire Compilation - 2014

Mathias Bourgoin

Emmanuel Chailloux et Jean-Luc Lamotte

02.07.2014

Carte graphique

Propriétés d'une carte graphique dédiée

- Plusieurs multi-processeurs
- Une mémoire dédiée
- Connectée à un hôte par un bus PCI-Express
- Les données sont transférées entre les mémoires de l'hôte et de la carte graphique
- Une programmation particulière et complexe

Matériel actuel

	CPU	GPU
# cores	4–16	300–2000
Mémoire max	32GB	6GB
GFLOPS SP	200	1000–4000
GFLOPS DP	100	100–1000

Addition de vecteurs

```
__kernel void vec_add(__global const double * a,
                      __global const double * b,
                      __global double * c, int N)
{
    int nIndex = get_global_id(0);
    if (nIndex >= N)
        return;
    c[nIndex] = a[nIndex] + b[nIndex];
}
```

La programmation GPGPU en pratique

Programme hôte : Un petit exemple en C

```
// create OpenCL device & context
cl_context hContext;
hContext = clCreateContextFromType(0, ↩
      CL_DEVICE_TYPE_GPU,
                                   0, 0, 0);
// query all devices available to the context
size_t nContextDescriptorSize;
clGetContextInfo(hContext, CL_CONTEXT_DEVICES,
                 0, 0, &nContextDescriptorSize);
cl_device_id * aDevices = malloc(↩
      nContextDescriptorSize);
clGetContextInfo(hContext, CL_CONTEXT_DEVICES,
                 nContextDescriptorSize, aDevices, 0)↩
                     ;
// create a command queue for first device the ↩
      context reported
cl_command_queue hCmdQueue;
hCmdQueue = clCreateCommandQueue(hContext, aDevices↩
      [0], 0, 0);
// create & compile program
cl_program hProgram;
hProgram = clCreateProgramWithSource(hContext, 1,
                                     sProgramSource, ↩
                                         0, 0);
clBuildProgram(hProgram, 0, 0, 0, 0, 0);

// create kernel
cl_kernel hKernel;
hKernel = clCreateKernel(hProgram, ""vec_add, 0);

// allocate device memory
cl_mem hDeviceMemA, hDeviceMemB, hDeviceMemC;
hDeviceMemA = clCreateBuffer(hContext,
                 CL_MEM_READ_ONLY | ↩
                       CL_MEM_COPY_HOST_PTR,
                 cnDimension * sizeof(cl_double),
                 pA,
                 0);
hDeviceMemB = clCreateBuffer(hContext,
                 CL_MEM_READ_ONLY | ↩
                       CL_MEM_COPY_HOST_PTR,
                 cnDimension * sizeof(cl_double),
                 pA,
                 0);
hDeviceMemC = clCreateBuffer(hContext,
                 CL_MEM_WRITE_ONLY,
                 cnDimension * sizeof(cl_double),
                 0, 0);
// setup parameter values
clSetKernelArg(hKernel, 0, sizeof(cl_mem), (void *)&↩
      hDeviceMemA);
clSetKernelArg(hKernel, 1, sizeof(cl_mem), (void *)&↩
      hDeviceMemB);
clSetKernelArg(hKernel, 2, sizeof(cl_mem), (void *)&↩
      hDeviceMemC);
// execute kernel
clEnqueueNDRangeKernel(hCmdQueue, hKernel, 1, 0,
                       &cnDimension, 0, 0, 0, 0);
// copy results from device back to host
clEnqueueReadBuffer(hContext, hDeviceMemC, CL_TRUE, ↩
      0,
                    cnDimension * sizeof(cl_double),
                    pC, 0, 0, 0);
clReleaseMemObj(hDeviceMemA);
clReleaseMemObj(hDeviceMemB);
clReleaseMemObj(hDeviceMemC);
```

Motivations

Quelques problèmes

- outils complexes
- systèmes incompatibles entre eux
- langages/bibliothèques verbeux
- systèmes de bas niveau
- gestion explicite des dispositifs et de la mémoire
- compilation souvent dynamique

- difficile à mettre au point
- difficile à *debugger*
- beaucoup d'efforts pour obtenir de bonnes performances

Motivations

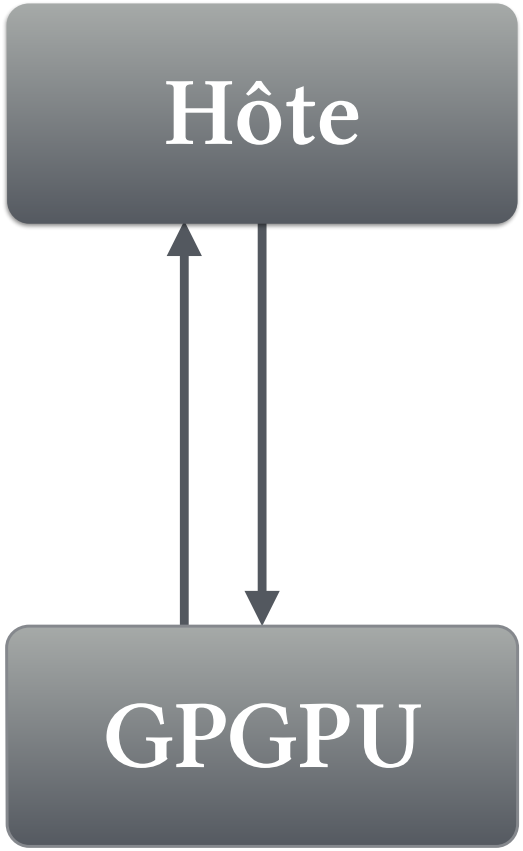
Quelles solutions ?

- un outil/langage simple et expressif
- compatible avec tous les systèmes GPGPU
- adapté aux langages de haut niveau
- qui abstrait dispositifs et transferts mémoires
- qui permet la composition des calculs
- compilation et typage statique
- plus simple à *debugger*
- plus simple à mettre au point

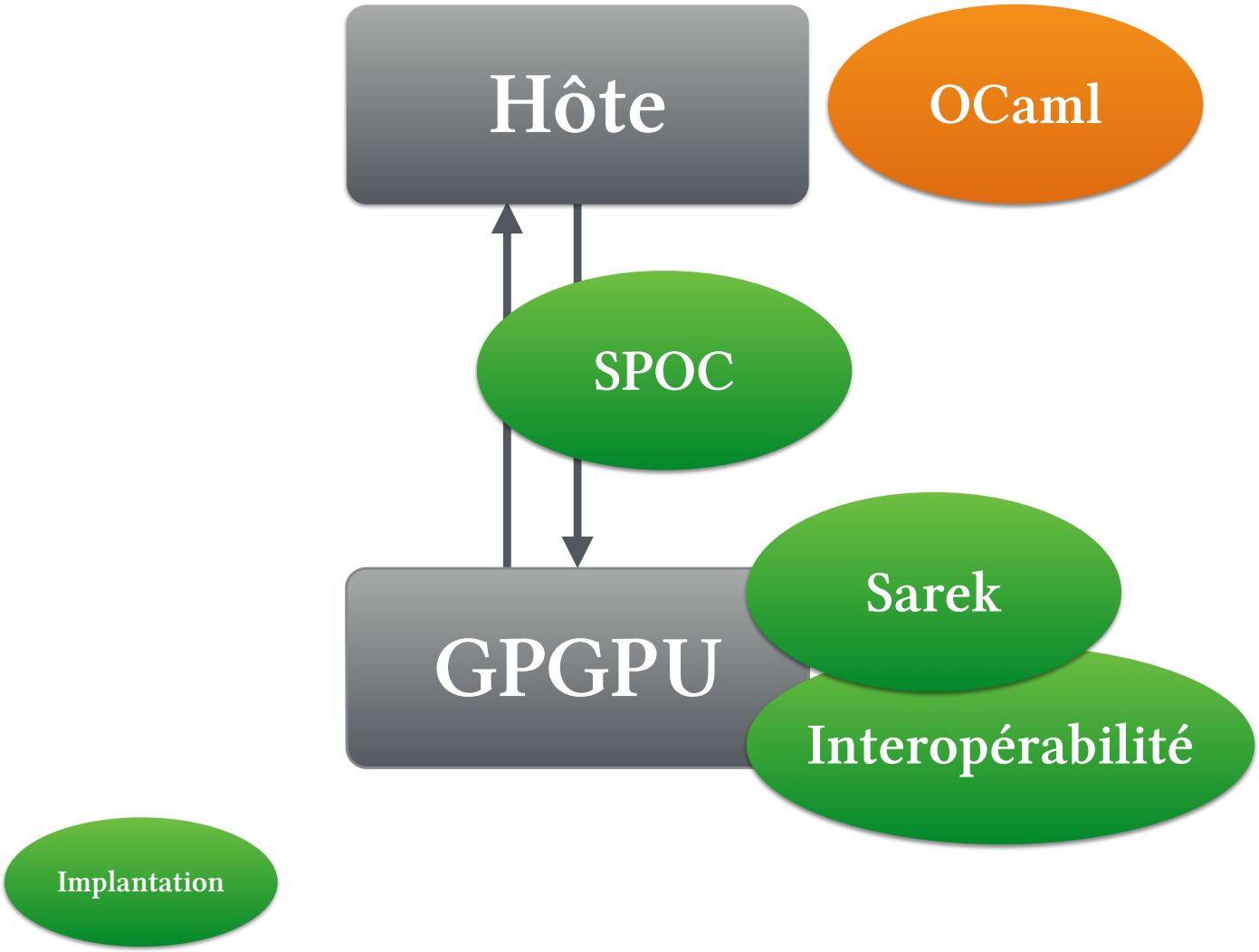
Quelles contraintes ?

- Maintenir un haut niveau de performances
- Utilisable dans un environnement très hétérogène

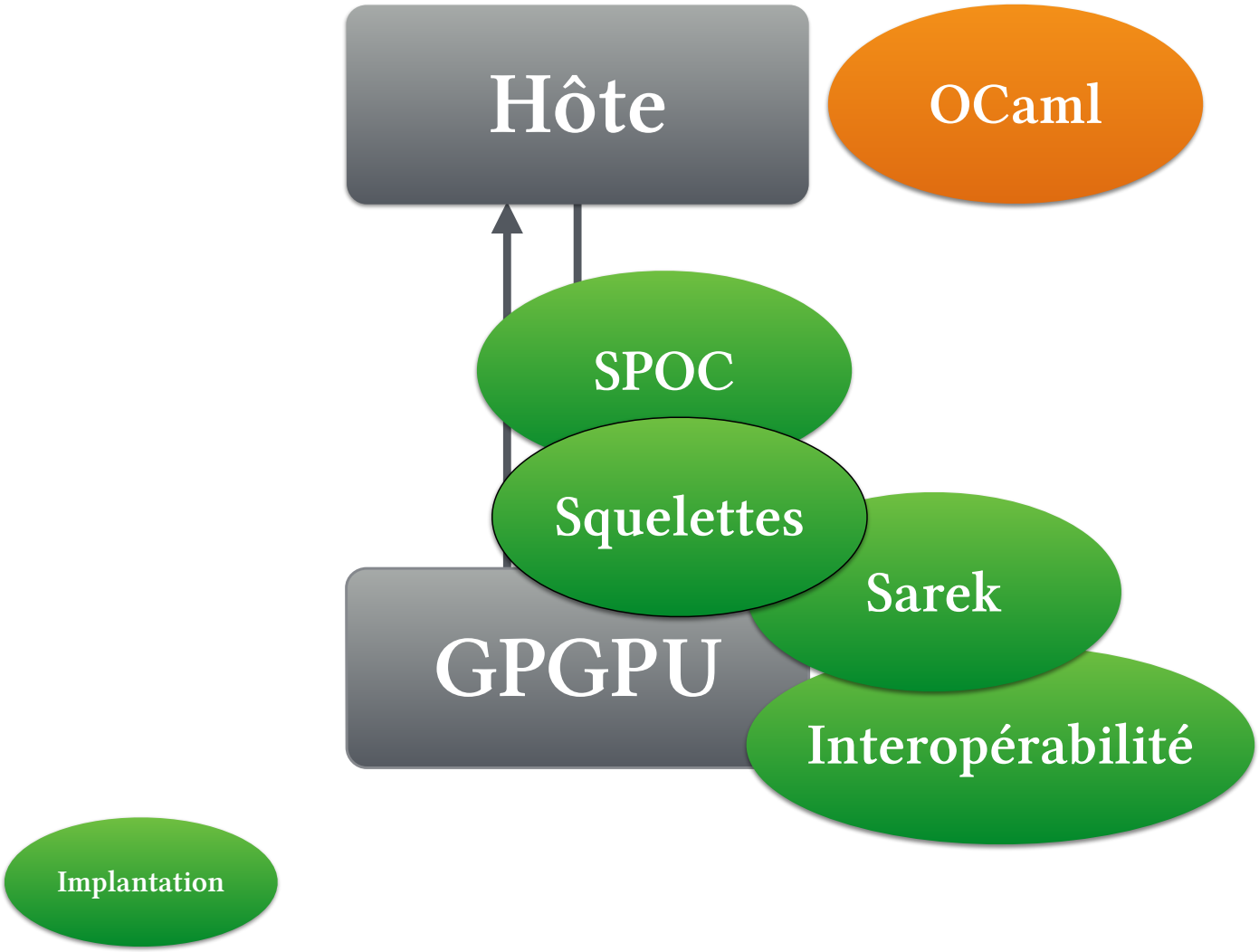
Vue d'ensemble



Vue d'ensemble



Vue d'ensemble



Principaux objectifs

- Permettre l'utilisation des systèmes Cuda/OpenCL avec OCaml
- Abstraire ces deux systèmes
- Abstraire les transferts mémoires
- Utiliser le typage statique pour vérifier les noyaux de calcul
- Proposer des abstractions pour la programmation GPGPU
- Conserver de hautes performances

Solution côté hôte : une bibliothèque pour OCaml

Un aperçu de SPOC

Abstraire les systèmes

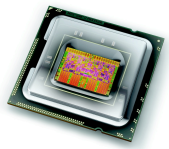
- Unification des deux API (Cuda/OpenCL), **liaison dynamique**.
- Solution portable, multi-GPGPU, hétérogène

Abstraire les transferts

Les vecteurs se déplacent automatiquement entre le CPU et les GPGPU

- Transferts à la demande
- **Allocation/Libération automatique** de l'espace mémoire utilisé par les vecteurs (sur l'hôte mais aussi sur les dispositifs)
- Un échec lors d'une allocation sur un GPGPU déclenche une collection

Un petit exemple



CPU RAM

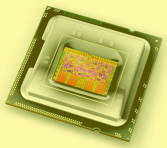
GPU0 RAM

GPU1 RAM

Exemple

```
let dev = Devices.init ()
let n = 1_000_000
let v1 = Vector.create Vector.float64 n
let v2 = Vector.create Vector.float64 n
let v3 = Vector.create Vector.float64 n

let k = vector_add (v1, v2, v3, n)
let block = {blockX = 1024; blockY = 1; blockZ = 1}
let grid={gridX=(n+1024-1)/1024; gridY=1; gridZ=1}

let main () =
  random_fill v1;
  random_fill v2;
  Kernel.run k (block,grid) dev.(0);
  for i = 0 to Vector.length v3 - 1 do
   Printf.printf "res[%d] = %f ; " i v3.[<i>]
  done;
```

Un petit exemple



v1
v2
v3
CPU RAM

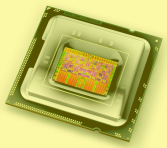
GPU0 RAM

GPU1 RAM

Exemple

```
let dev = Devices.init ()
let n = 1_000_000
let v1 = Vector.create Vector.float64 n
let v2 = Vector.create Vector.float64 n
let v3 = Vector.create Vector.float64 n

let k = vector_add (v1, v2, v3, n)
let block = {blockX = 1024; blockY = 1; blockZ = 1}
let grid={gridX=(n+1024-1)/1024; gridY=1; gridZ=1}

let main () =
  random_fill v1;
  random_fill v2;
  Kernel.run k (block,grid) dev.(0);
  for i = 0 to Vector.length v3 - 1 do
   Printf.printf "res[%d] = %f ; " i v3.[<i>]
  done;
```

Un petit exemple



v1
v2
v3
CPU RAM

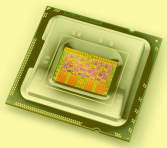
GPU0 RAM

GPU1 RAM

Exemple

```
let dev = Devices.init ()
let n = 1_000_000
let v1 = Vector.create Vector.float64 n
let v2 = Vector.create Vector.float64 n
let v3 = Vector.create Vector.float64 n

let k = vector_add (v1, v2, v3, n)
let block = {blockX = 1024; blockY = 1; blockZ = 1}
let grid={gridX=(n+1024-1)/1024; gridY=1; gridZ=1}

let main () =
  random_fill v1;
  random_fill v2;
  Kernel.run k (block,grid) dev.(0);
  for i = 0 to Vector.length v3 - 1 do
   Printf.printf "res[%d] = %f ; " i v3.[<i>]
  done;
```

Un petit exemple



v1
v2
v3
CPU RAM

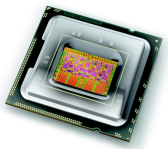
GPU0 RAM

GPU1 RAM

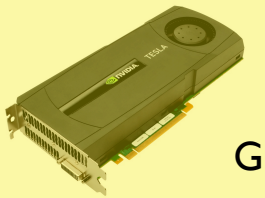
Exemple

```
let dev = Devices.init ()
let n = 1_000_000
let v1 = Vector.create Vector.float64 n
let v2 = Vector.create Vector.float64 n
let v3 = Vector.create Vector.float64 n

let k = vector_add (v1, v2, v3, n)
let block = {blockX = 1024; blockY = 1; blockZ = 1}
let grid={gridX=(n+1024-1)/1024; gridY=1; gridZ=1}

let main () =
  random_fill v1;
  random_fill v2;
  Kernel.run k (block,grid) dev.(0);
  for i = 0 to Vector.length v3 - 1 do
   Printf.printf "res[%d] = %f ; " i v3.[<i>]
  done;
```

Un petit exemple



CPU RAM



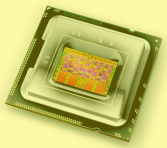
v1
v2
v3
GPU0 RAM

GPU1 RAM

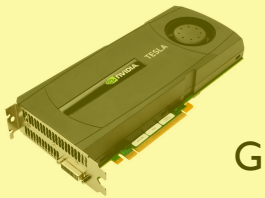
Exemple

```
let dev = Devices.init ()
let n = 1_000_000
let v1 = Vector.create Vector.float64 n
let v2 = Vector.create Vector.float64 n
let v3 = Vector.create Vector.float64 n

let k = vector_add (v1, v2, v3, n)
let block = {blockX = 1024; blockY = 1; blockZ = 1}
let grid={gridX=(n+1024-1)/1024; gridY=1; gridZ=1}

let main () =
  random_fill v1;
  random_fill v2;
  Kernel.run k (block,grid) dev.(0);
  for i = 0 to Vector.length v3 - 1 do
   Printf.printf "res[%d] = %f ; " i v3.[<i>]
  done;
```

Un petit exemple



v3
CPU RAM



v1
v2
GPU0 RAM

GPU1 RAM

Exemple

```
let dev = Devices.init ()
let n = 1_000_000
let v1 = Vector.create Vector.float64 n
let v2 = Vector.create Vector.float64 n
let v3 = Vector.create Vector.float64 n

let k = vector_add (v1, v2, v3, n)
let block = {blockX = 1024; blockY = 1; blockZ = 1}
let grid={gridX=(n+1024-1)/1024; gridY=1; gridZ=1}

let main () =
  random_fill v1;
  random_fill v2;
  Kernel.run k (block,grid) dev.(0);
  for i = 0 to Vector.length v3 - 1 do
   Printf.printf "res[%d] = %f ; " i v3.[<i>]
  done;
```

Comment exprimer les noyaux ?

Propriétés recherchées

- Simples à exprimer
- Aux performances prédictibles
- Facilement extensibles
- Compatibles avec les bibliothèques haute performance existantes
- Optimisables
- Plus sûrs qu'avec les solutions classiques

Deux solutions

Un DSL pour OCaml : Sarek

- Simple à exprimer
- Transformation simple depuis OCaml
- Plus sûr

Interoperabilité avec les noyaux Cuda/OpenCL

- Optimisations supplémentaires
- Compatible avec les bibliothèques actuelles
- Moins sûr

Sarek : Stream ARchitecture using Extensible Kernels

Addition de vecteurs en Sarek

```
let vec_add = kern a b c n ->
  let open Std in
  let idx = global_thread_id in
  if idx < n then
    c.[<idx>] <- a.[<idx>] + b.[<idx>]
```

Addition de vecteurs en OpenCL

```
__kernel void vec_add(__global const double * a,
                      __global const double * b,
                      __global double * c, int N)
{
    int nIndex = get_global_id(0);
    if (nIndex >= N)
        return;
    c[nIndex] = a[nIndex] + b[nIndex];
}
```

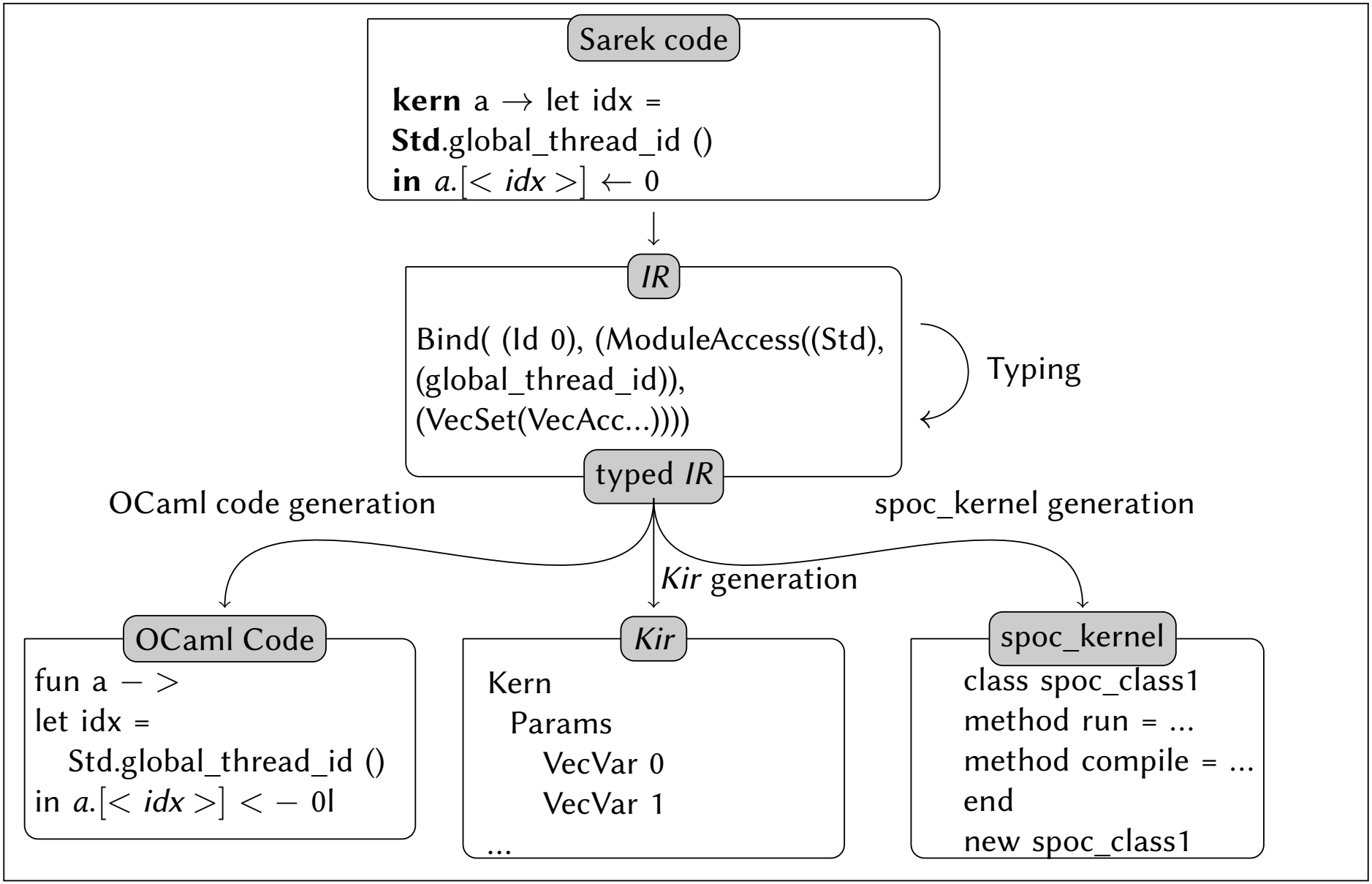
Addition de vecteurs en Sarek

```
let vec_add = kern a b c n ->
  let open Std in
  let idx = global_thread_id in
  if idx < n then
    c.[<idx>] <- a.[<idx>] + b.[<idx>]
```

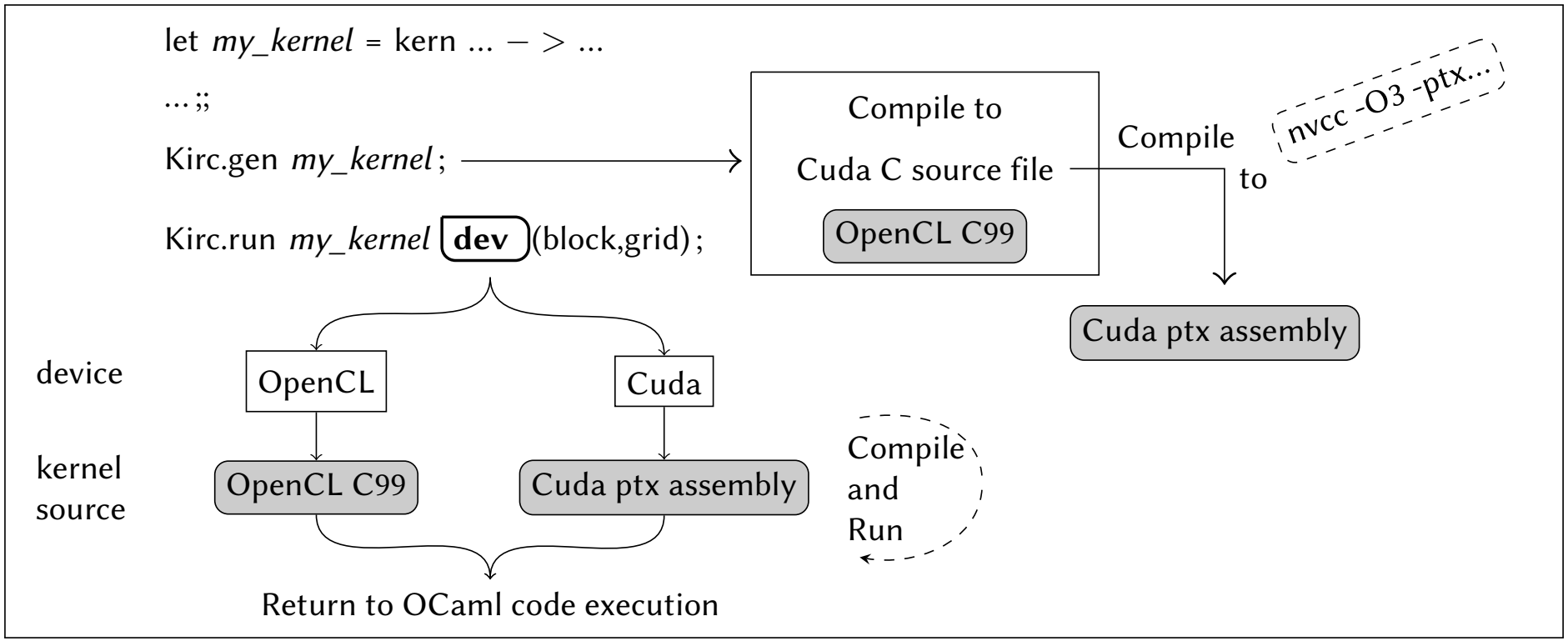
Sarek offre

- une syntaxe à la ML
- de l'inférence de types
- une vérification statique des types
- une compilation statique vers du code OCaml
- une compilation dynamique vers Cuda et OpenCL

Compilation statique de Sarek



Compilation dynamique de Sarek



Addition de vecteurs

SPOC + Sarek

```
open Spoc
let vec_add = kern a b c n ->
  let open Std in
  let idx = global_thread_id in
  if idx < n then
    c.[<idx>] <- a.[<idx>] + b.[<idx>]

let dev = Devices.init ()
let n = 1_000_000
let v1 = Vector.create Vector.float64 n
let v2 = Vector.create Vector.float64 n
let v3 = Vector.create Vector.float64 n

let block = {blockX = 1024; blockY = 1; blockZ = 1}
let grid={gridX=(n+1024-1)/1024; gridY=1; gridZ=1}

let main () =
  random_fill v1;
  random_fill v2;
  Kirc.gen vec_add;
  Kirc.run vec_add  (v1, v2, v3, n) (block,grid) dev.(0);
  for i = 0 to Vector.length v3 - 1 do
   Printf.printf "res[%d] = %f ; " i v3.[<i>]
  done;
```

- OCaml
- Aucun transfert
- Inférence de types
- Typage statique
- Portable
- Hétérogène

Composition de noyaux

Composition

Composer plusieurs noyaux pour exprimer des algorithmes complexes

Bénéfices

- Simplifie la programmation
- Permet de nouvelles optimisations automatiques
  - optimiser la grille virtuelle en fonction de la taille des données
  - recouvrir des transferts par du calcul
  - réaliser des transferts au plus tôt

Problème

Pour être composables, les noyaux doivent avoir des vecteurs d'entrées/sorties.

Squelettes parallèles utilisant des noyaux externes (Cuda/OpenCL)

Avec des noyaux externes :

On décrit un squelette comme :

- un noyau externe
- un environnement d'exécution
- une entrée
- une sortie

Deux types de squelettes

- *map* : $kernel \rightarrow env \rightarrow vector \rightarrow skeleton$
- *reduce* : $kernel \rightarrow env \rightarrow vector \rightarrow skeleton$

- *pipe* : $skeleton \rightarrow skeleton \rightarrow skeleton$

Bénéfices des squelettes et de la composition

Bénéfices

- Décrivent explicitement les relations entre noyaux/données
- Projection automatique des grilles/blocs sur les GPGPU
  - Simplification du code
  - Optimisation automatique en fonction de l'architecture
- Optimisent le positionnement des données
  - Allocation/libération au plus tôt
  - Répartition des données sur plusieurs GPGPU
- Optimisent les transferts automatiques
  - *Multi-buffering*
  - Recouvrement des transferts par du calcul (avec la composition *pipe*)

Squelettes parallèles utilisant des noyaux internes (Sarek)

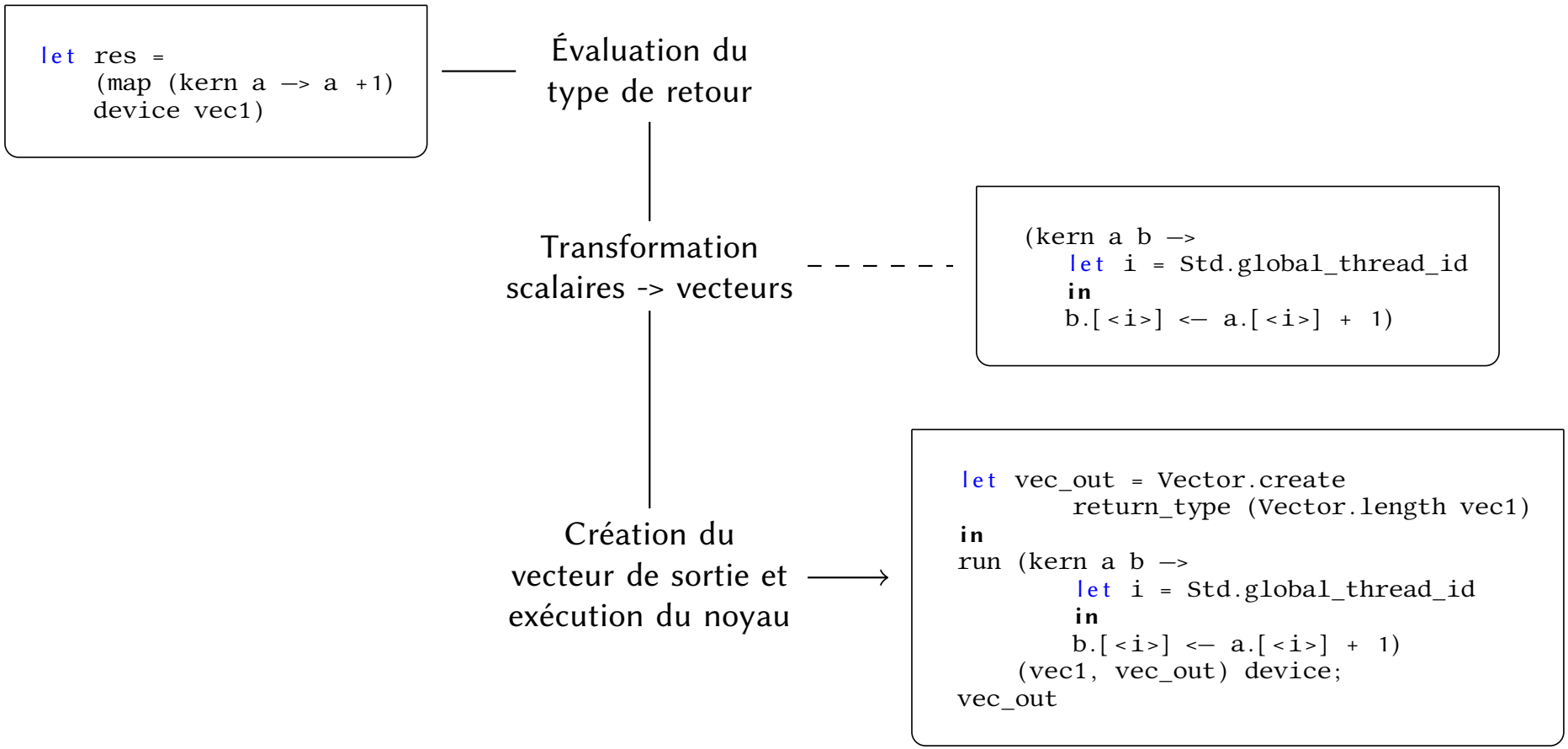
Avec Sarek

Sarek expose l'AST Kir (Kernel Internal Representation) des noyaux
au programme hôte. Les squelettes sont des **fonctions transformant l'AST Kir** :

Exemple :
map (kern a -> b)

Les calculs scalaires $('a \rightarrow 'b)$ sont transformés
en calculs vectoriels $('a\ vector \rightarrow 'b\ vector)$.

Transformation de noyaux Sarek



Addition de vecteurs

Exemple

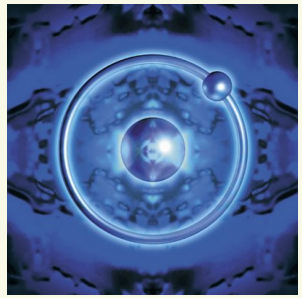
```
let v1 = Vector.create Vector.float64 10_000
and v2 = Vector.create Vector.float64 10_000
in
let v3 = map2 (kern a b -> a + b) v1 v2
```

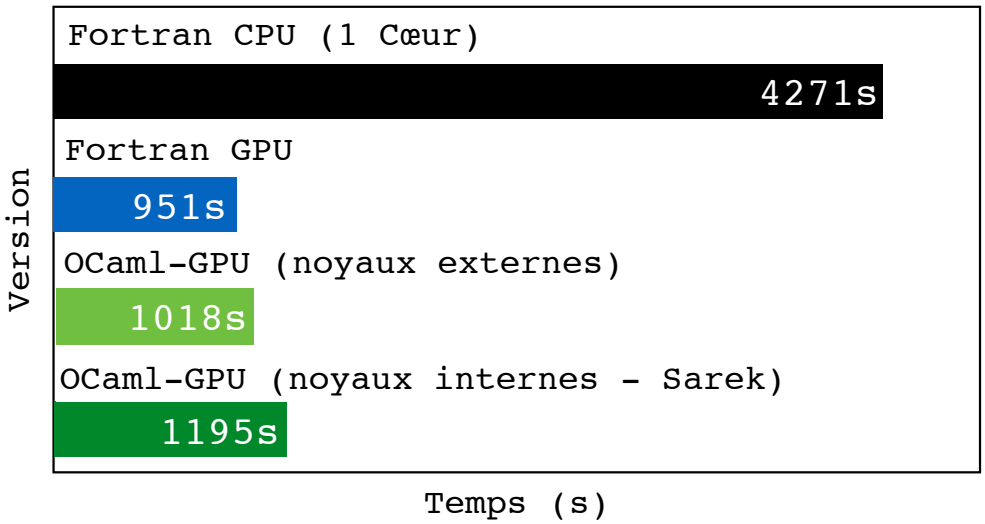
```
val map2 :
  ('a -> 'b -> 'c, 'd) kirc_kernel ->
  ?dev:Spoc.Devices.device ->
  'a Spoc.Vector.vector ->
  'b Spoc.Vector.vector -> 'd Spoc.Vector.vector
```

Un exemple réaliste

PROP

- Primé par le *UK Research Councils' HEC Strategy Committee*
- Simule la diffusion d'e^- dans des ions à des énergies intermédiaires
- Programmé en Fortran
- Compatible : architectures séquentielles, *clusters* HPC, super-calculateurs





SPOC+DSL conservent 80% des performances du code Fortran optimisé

SPOC+noyaux externes au niveau du Fortran (93%)

Sûreté de typage
Gestionnaire mémoire

Garbage collector

Réduction du code~30%
Plus aucun transfert explicite

Conclusion

Implantation : SPOC et Sarek

- Unifient Cuda/OpenCL
- Transferts automatiques
- Inférence de types et typage statique
- Extensibles simplement
- Compatibles avec les bibliothèques optimisées existantes

Implantation : Squelettes et Transformations

- Simplifient la programmation
- Permettent des optimisations automatiques supplémentaires

Test de performance : portage de PROP

- Plus de sûreté (mémoire/typage)
- Conservation des performances
- Validation de la démarche

Travaux en cours

Enrichir l'implantation

- Enrichir Sarek : types, récursion, polymorphisme...
- Optimiser la génération de code
- Optimisations automatiques pour différentes architectures

Enrichir les squelettes

- Modèle de coût pour Sarek
- Plus de squelettes basés sur Sarek
- Squelettes dédiés aux architectures très hétérogènes (ex : supercalculateurs)

Travaux en cours

SPOC pour le web

- Accéder aux performances des GPGPU depuis les navigateurs.
- En utilisant le compilateur js_of_ocaml
- Portage de la partie bas niveau et écriture d'un gestionnaire mémoire

- Source et démos web : `http://www.algo-prog.info/spoc/`
- SPOC est installable *via* OPAM (OCaml Package Manager)

Diffusion et Enseignement

- Web = immédiatement accessible
- Plus simple que les outils classiques : libère des transferts

- Permet de cibler l'optimisation des noyaux
- Mais surtout la composition d'algorithmes parallèles

Merci

Emmanuel Chailloux
Jean-Luc Lamotte

SPOC : `http://www.algo-prog.info/spoc/`
Spoc est compatible x86_64 : Unix (Linux, Mac OS X), Windows

Pour plus d'informations :
mathias.bourgoin@lip6.fr