

Abstractions Performantes Pour Cartes Graphiques

Soutenance de thèse

Mathias Bourgoïn

Sous la direction de Emmanuel Chailloux et Jean-Luc Lamotte

11.12.2013

Abstractions performantes pour cartes graphiques

- Cartes graphiques → programmation GPGPU
- Abstractions → langages et constructions algorithmiques
- Performances → programmation haute performance
- Applications → calcul scientifique et simulation numérique

Projet OpenGPU

- Pôle Systematic
- Partenaires universitaires et industriels
- Objectif : proposer des solutions open-source pour la programmation GPGPU
- Succès : développement d'applications numériques réalistes

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

Deux principaux systèmes (*frameworks*)

- **Cuda** (NVidia)
- **OpenCL** (Consortium OpenCL)

Des langages différents

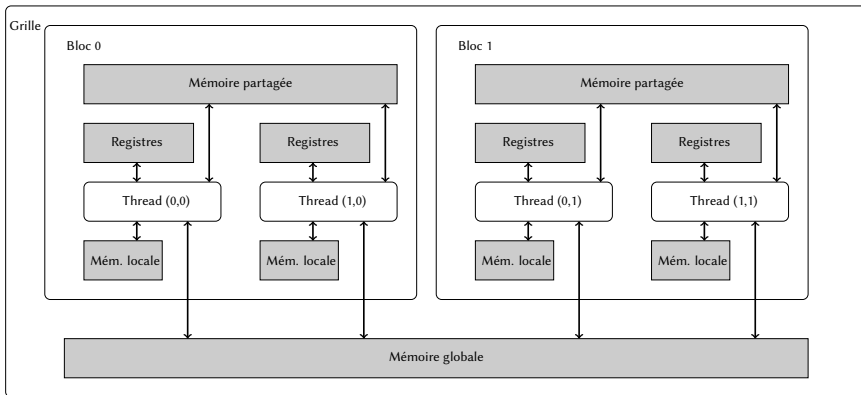
- Pour écrire les noyaux de calcul
 - **Assembleurs** (ptx, il,...)
 - Sous-ensembles de **C/C++**
- Pour manipuler les noyaux de calcul
 - **C/C++/Objective-C**
 - *Bindings* : Fortran, Python, Java, ...



Stream Processing

À partir d'un jeu de données (le flux – *stream*), une série d'opérations (un noyau de calcul – *kernel*) est appliquée à chaque élément du flux.

La programmation GPGPU en pratique 1



Il ne faut pas oublier les transferts entre l'hôte et ses invités

	CPU-X86 i7-3770K	GPU Mobile GTX 680M	GPU Gamer		GPU HPC K20X
Bande passante mem.	25.6GB/s	115.2 GB/s	192.2GB/s	264GB/s	250GB/s

La bande passante max. du PCI-Express 3.0 est de 16GB/s

La programmation GPGPU en pratique 2

Noyau : Un petit exemple en OpenCL

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 2

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);
```

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

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

État de l'art : au commencement de la thèse

Systèmes

- Cuda est disponible depuis 2 ans
- OpenCL est disponible depuis quelques mois

Langages

- DSL : Obsidian (Haskell)
- *Bindings* : pyCuda, jCuda, ...

Bibliothèques

- Calculs optimisés : Cublas, Magma, ...
- Squelettes : StarPU, SkePU

Cuda est l'assembleur de plus haut niveau pour les GPGPU

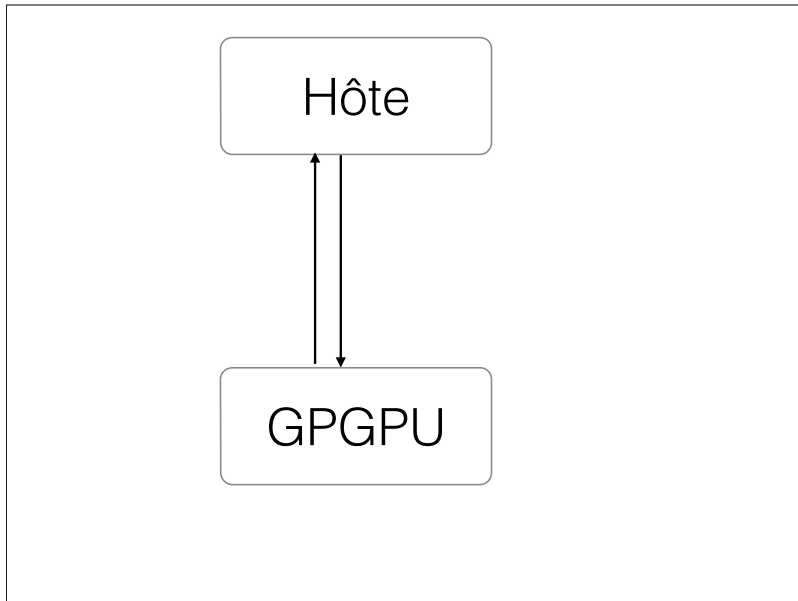
1 Langages pour la programmation GPGPU

2 Noyaux de calcul

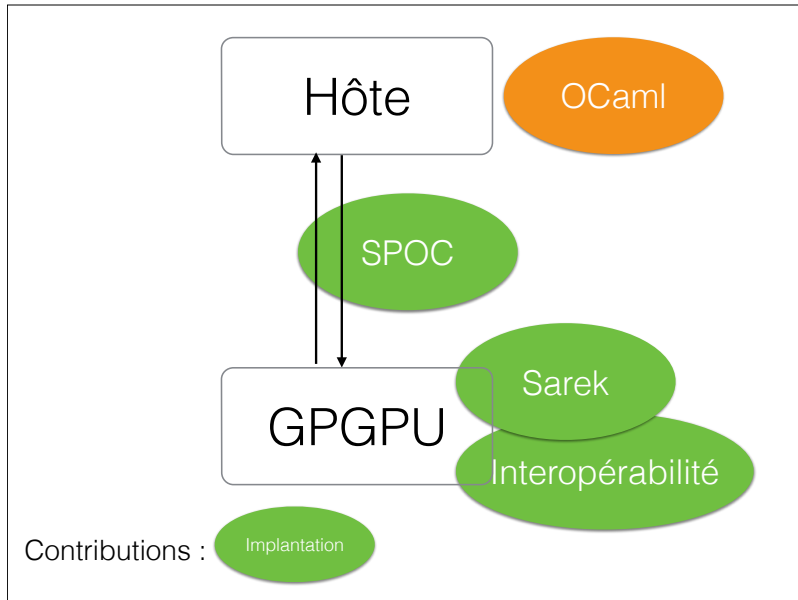
3 Composition de noyaux

4 Tests de performance

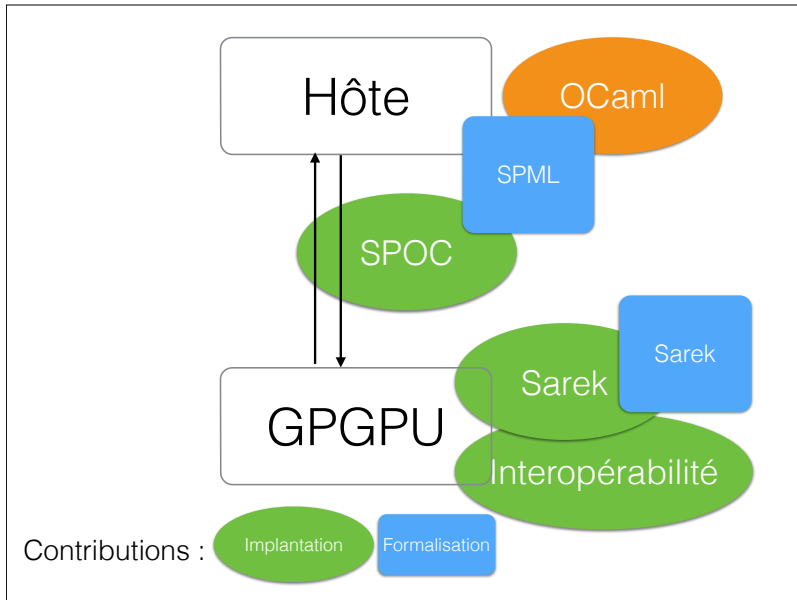
5 Conclusion & Perspectives



Contributions



Contributions



Deux langages pour la programmation GPGPU

Un langage hôte : SPML (Stream Processing miniML)

- Basé sur ML (simple et expressif)
- Fonctionnel et impératif
- Typé statiquement
- Primitives dédiées à la programmation GPGPU
- Transferts automatiques entre les dispositifs

Un langage pour les noyaux de calcul : Sarek (Stream ARchitectures using Extensible Kernels)

- Basé sur les extensions C pour Cuda/OpenCL
- Impératif
- Typé statiquement
- Syntaxe à la ML pour plus de cohérence avec SPML

Formalisation difficile

Très peu fréquent dans la littérature

Les systèmes GPGPU Cuda/OpenCL sont des « boîtes noires »

Un chapitre entier de la thèse

Pourquoi ?

- Décrire les caractéristiques des langages implantés
- Apporter des garanties sur la position des données
- Apporter des garanties sur l'exécution d'un programme GPGPU
- Fournir une spécification pour des implantations dans différents langages

Vecteur transférable

- Ensemble uniforme de données
- Automatiquement transférable d'un espace mémoire à un autre

Propriétés de SPML

- Un vecteur ne peut être présent dans plusieurs espaces mémoire à la fois
- Un noyau de calcul ne peut s'exécuter que si les vecteurs dont il dépend sont en mémoire GPGPU
- L'hôte ne peut modifier les données utilisées par un noyau en cours d'exécution
- Si un noyau termine alors les données sont accessibles par l'hôte

Vecteur transférable

- Ensemble uniforme de données
- Automatiquement transférable d'un espace mémoire à un autre

Propriétés de Sarek

- Les noyaux ne peuvent accéder à la mémoire hôte
- Les noyaux ne peuvent modifier l'adresse d'un vecteur transférable

Principaux objectifs

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

Solution côté hôte : une bibliothèque pour OCaml



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



GPU0 RAM

v1
v2
v3



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;
```

1 Langages pour la programmation GPGPU

2 Noyaux de calcul

3 Composition de noyaux

4 Tests de performance

5 Conclusion & Perspectives

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

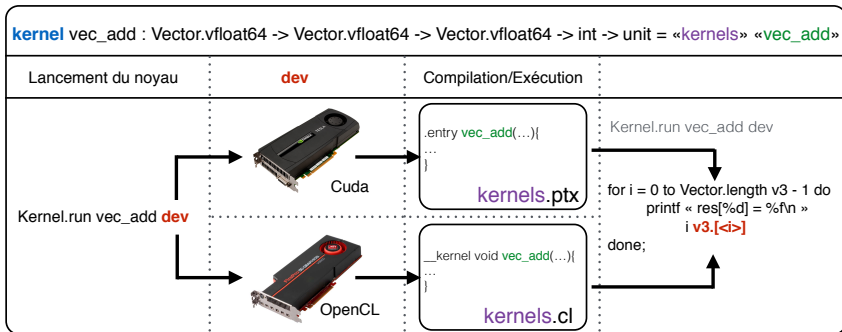
- Simple à exprimer
- Transformation simple depuis OCaml
- Plus sûr

Interopérabilité avec les noyaux Cuda/OpenCL

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

Sûreté de typage

- Vérification statique des types des arguments (à la compilation)
- Kernel.run compile les noyaux depuis des sources .ptx / .cl



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];
}
```

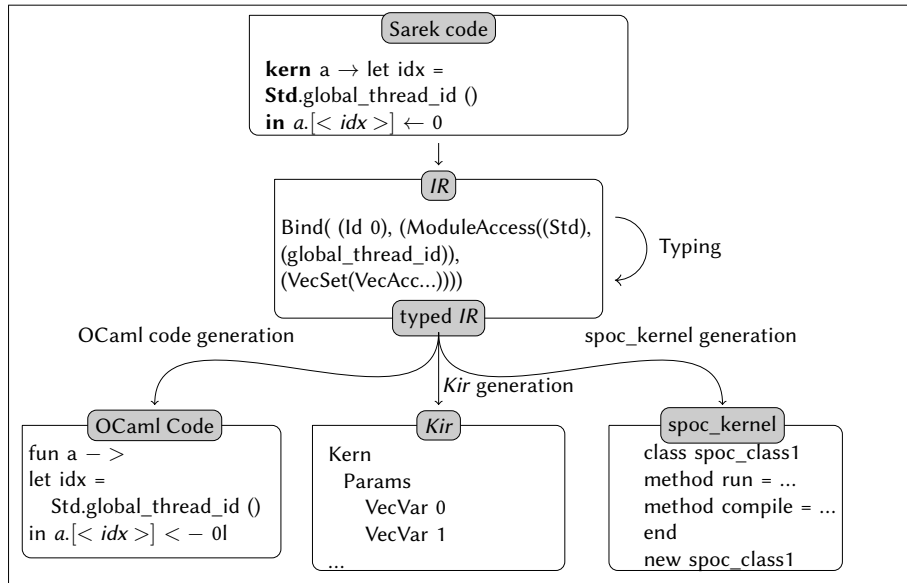
Addition de vecteurs en Sarek

```
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>]
```

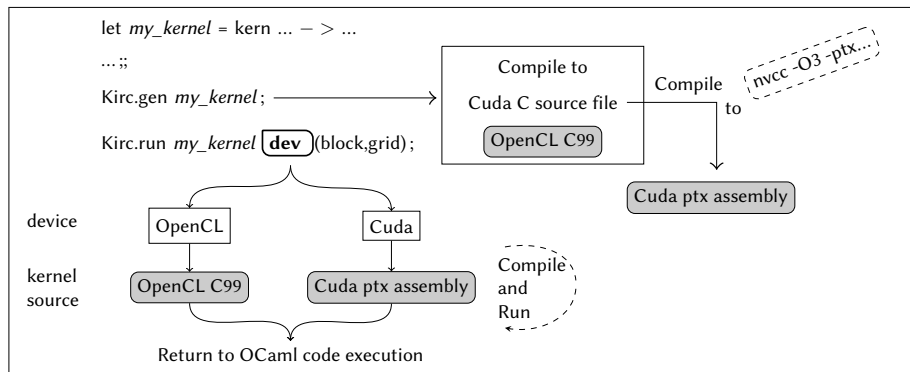
Sarek offre

- 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



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

1 Langages pour la programmation GPGPU

2 Noyaux de calcul

3 Composition de noyaux

4 Tests de performance

5 Conclusion & Perspectives

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 fonctions d'exécution

- *run* : exécute sur un GPGPU
- *par_run* : exécute sur une liste de GPGPU

Exemple

Puissance itérée

SPOC

```
while (iter<IterMax)&&(max_n>eps) do
  let x=A*x0 in
  let m = max(x) in
  let x=u/m in
  let n = abs(x - x0) in
  max_n <- max(n);
  x0<-x; iter<-iter+1;
done
```

Squelettes

```
while (iter<IterMax)&&(max_n>eps) do
  let x= map ( * x0) A in
  let m = reduce (max) x in
  let x= map ( / m) u in
  let n = map (abs) x-x0 in
  max_n <- reduce max n;
  x0<-x; iter<-iter+1;
done
```

Composition

```
while (iter<IterMax)&&(max_n > eps) do
  let m= pipe (map ( *x0)) (reduce max) A in
  max_n <- pipe
    (pipe
      (map ( / m)
        (map (abs(- x0)))))
    (reduce max) u;
  x0<-x; iter<-iter+1;
done
```

Exemple

Puissance itérée

SPOC

```
while (iter<IterMax)&&(max_n>eps) do
  let x=A*x0 in
  let m = max(x)in
  let x=u/m in
  let n = abs(x - x0) in
  max_n <- max(n);
  x0<-x; iter<-iter+1;
done
```

Squelettes

```
while (iter<IterMax)&&(max_n>eps) do
  let x= map ( * x0) A in
  let m = reduce (max) x in
  let x= map ( / m) u in
  let n = map (abs) x-x0 in
  max_n <- reduce max n;
  x0<-x; iter<-iter+1;
done
```

Composition

```
while (iter<IterMax)&&(max_n > eps) do
  let m= pipe (map ( *x0)) (reduce max) A in
  max_n <- pipe
    (pipe
      (map ( / m)
        (map (abs(- x0)))))
    (reduce max) u;
  x0<-x; iter<-iter+1;
done
```

Exemple

Puissance itérée

SPOC

```
while (iter<IterMax)&&(max_n>eps) do
  let x=A*x0 in
  let m = max(x) in
  let x=u/m in
  let n = abs(x - x0) in
  max_n <- max(n);
  x0<-x; iter<-iter+1;
done
```

Squelettes

```
while (iter<IterMax)&&(max_n>eps) do
  let x= map ( * x0) A in
  let m = reduce (max) x in
  let x= map ( / m) u in
  let n = map (abs) x-x0 in
  max_n <- reduce max n;
  x0<-x; iter<-iter+1;
done
```

Composition

```
while (iter<IterMax)&&(max_n > eps) do
  let m= pipe (map ( *x0)) (reduce max) A in
  max_n <- pipe
    (pipe
      (map ( / m)
        (map (abs(- x0)))))
    (reduce max) u;
  x0<-x; iter<-iter+1;
done
```

Bénéfices

- Décrit explicitement les relations entre noyaux/données
- Projection automatique des grilles/blocs sur les GPGPU
 - Simplification du code
 - Optimisation automatique en fonction de l'architecture
- Optimise le positionnement des données
 - Allocation/libération au plus tôt
 - Répartition des données sur plusieurs GPGPU (avec *par_run*)
- Optimise 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 $\rightarrow$ b)

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

Transformation de noyaux Sarek

```
let res =  
  (map (kern a -> a + 1)  
   device vec1)
```

Évaluation du
type de retour

Transformation
scalaires -> vecteurs

Création du
vecteur de sortie et
exécution du noyau

```
(kern a b ->  
  let i = Std.global_thread_id  
  in  
  b.[<i>] <- a.[<i>] + 1)
```

```
let vec_out = Vector.create  
  return_type (Vector.length vec1)  
in  
run (kern a b ->  
  let i = Std.global_thread_id  
  in  
  b.[<i>] <- a.[<i>] + 1)  
  (vec1, vec_out) device;  
vec_out
```

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
```

1 Langages pour la programmation GPGPU

2 Noyaux de calcul

3 Composition de noyaux

4 Tests de performance

5 Conclusion & Perspectives

Environnement de tests complexe

Performances différentes pour Cuda et OpenCL sur une même carte

Comportement des programmes influencé par les *drivers*

Attention au système d'exploitation

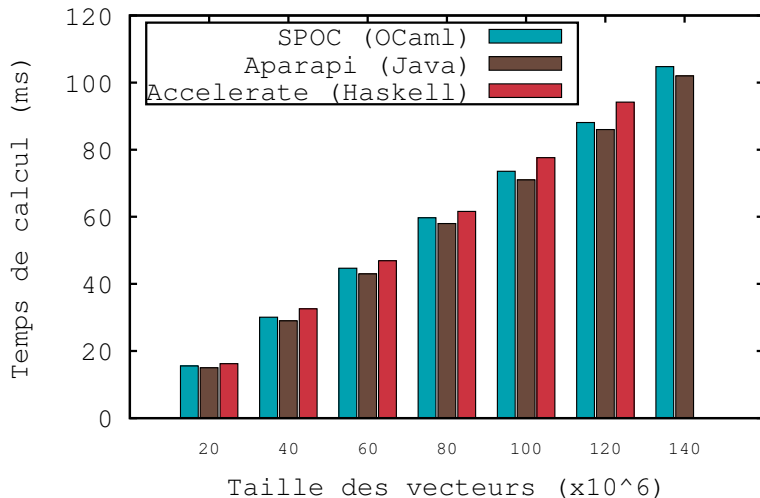
Matériel fragile...

Comparaison

Accelerate (Haskell) :	vecteurs & combinateurs Cuda / OpenCL / séquentiel pas de noyaux externes/internes
Aparapi (Java)	vecteurs OpenCL noyaux en Java
Rust-GPU (Rust - Mozilla)	OpenCL pour les transferts Cuda pour les noyaux

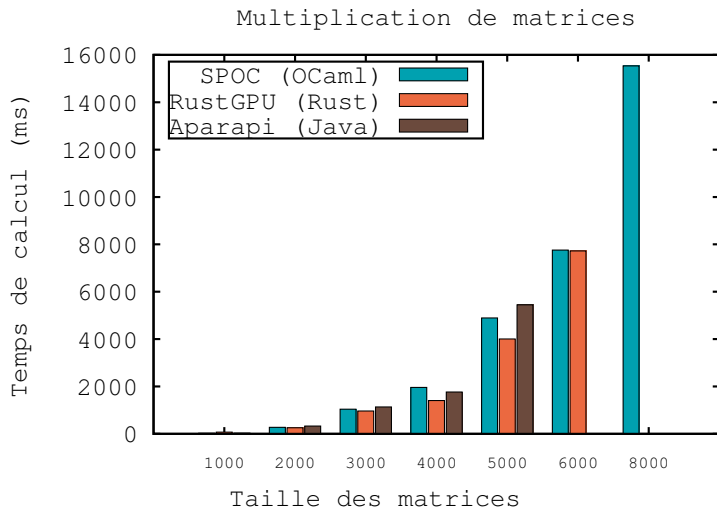
Tests de performance : petits exemples - Comparaisons

Addition de vecteurs



GPU : GTX-680

Tests de performance : petits exemples - Comparaisons



GPU : GTX-680

Tests de performance : petits exemples - Multi-GPU

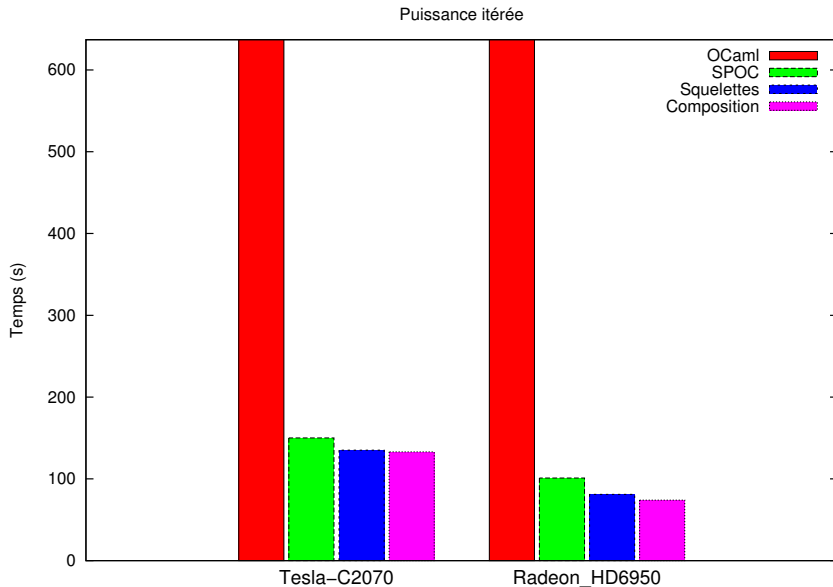
Ordinateur \ Test	MatMult	
	temps (s)	accélération
OCaml séquentiel	85	×1
GPU Seul Tesla C2070	1.23	×65.4
Système multi-GPU 1× GTX 680 + 1× GTX 460 + 2× Quadro Q2000	0.44	×193.18

Calcul par blocs

Equilibrage dynamique

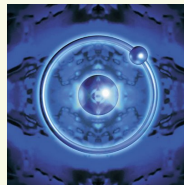
Algorithme simple (~20 lignes d'OCaml)

Tests de performance : Squelettes et Composition



PROP

- Inclus dans la suite 2DRMP
- Primé par le *UK Research Councils' HEC Strategy Committee*
- Simule la diffusion d' e^- dans des ions hydrogénoïdes à des énergies intermédiaires.
- PROP propage une $\mathcal{R}$ -matrice dans un espace entre 2 électrons
- Les calculs impliquent principalement des multiplications de matrices
- Les matrices calculées grandissent au cours des calculs
- Programmé en Fortran
- Compatible avec les architectures séquentielles, les *clusters* HPC, super-calculateurs



Sélectionné par GENCI pour une étude d'implantation GPGPU

Première optimisation (*Caps-Entreprise*)

- Multiplication de matrices portée vers Cuda *via* le compilateur HMPP
- Equation de propagation modifiée pour manipuler de plus grandes matrices
- Ratio transferts/calculs diminué mais toujours de nombreux calculs exécutés par le CPU

Seconde optimisation (*LIP6*)

- Réduit les transferts en réalisant l'ensemble des calculs de la propagation sur le GPGPU
- Recouvre les transferts par des calculs sur différentes sections de la $\mathcal{R}$ -matrice
- Utilise les bibliothèques Cublas et Magma pour les calculs
- Utilise un *binding* C entre Fortran et Cuda

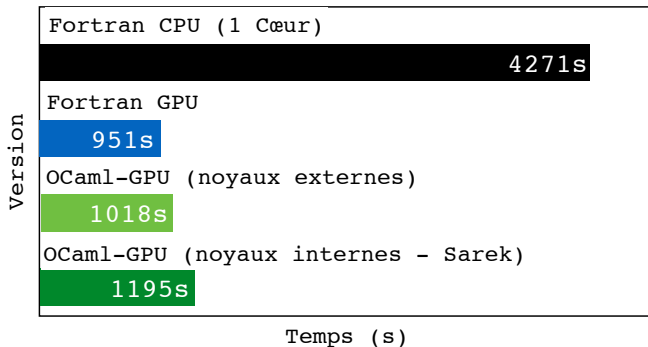
Notre approche

- Traduire uniquement la partie calculs (en conservant I/O et initialisation en Fortran)
- *Binding* d'un sous ensemble des bibliothèques Cublas et Magma pour OCaml (à l'aide de SPOC)
- Glu C entre Fortran et OCaml

Résultat

- Un programme mixant Fortran, C, OCaml et Sarek
- Importante réduction de la taille du code
- Plus aucun transfert !!

Résultats : PROP



SPOC+Sarek 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 de transferts

1 Langages pour la programmation GPGPU

2 Noyaux de calcul

3 Composition de noyaux

4 Tests de performance

5 Conclusion & Perspectives

Contributions de la thèse

Formalisation : SPML et Sarek

- Sémantique opérationnelle
- Offre des garanties sur la position/utilisation des données

Implantation : SPOC

- Unifie Cuda/OpenCL
- Transferts automatiques
- Compatible avec les bibliothèques optimisées existantes

Implantation : Sarek

- Syntaxe à la OCaml
- Inférence de types et typage statique
- Extensible simplement

Contributions de la thèse

Implantation : Squelettes

- Simplifie la programmation
- Permet des optimisations automatiques supplémentaires

Tests de performance

- Au niveau des autres solutions
- Hétérogène
- Performant sur GPGPU mais aussi sur CPU

Application : portage de PROP

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

Résultats

- Haute performance
 - Comparable aux autres solutions de haut niveau
 - Compatible bibliothèques haute performance
 - Validé sur une application réaliste
- Portable et Hétérogène
 - Bibliothèque unifiée
 - Compatible Cuda et OpenCL
 - Liaison dynamique
 - Noyaux compilés automatiquement vers Cuda et OpenCL
- Base pour des abstractions supplémentaires
 - Représentation interne des noyaux accessible depuis le code hôte
 - Le compilateur Sarek est une fonction OCaml
 - Un squelette Sarek est une fonction OCaml
 - OCaml comme langage de composition

Le matériel évolue

- Xeon Phi (manycore x86)
- PlayStation 4, Xbox One, AMD Fusion : mémoire unifiée, pas de transfert
- Nvidia Kepler : noyaux récursifs

Les systèmes évoluent

Cuda 5.5 - 6.0, OpenCL 2.0

OpenCL SPIR , OpenACC , WebCL et RenderScript (Google)

Enrichir la formalisation

- Alléger la sémantique opérationnelle
- Implantation dans d'autres langages : F#, Scala, Java...

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

Diffusion

- Déjà présenté plusieurs fois
- A différentes communautés
- Déjà des utilisateurs
- SPOC : <http://www.algo-prog.info/spoc/>
- Installable via OPAM (OCaml Package Manager)

Enseignement

- Plus simple que les outils classiques
- Libère des questions de transferts
- Permet de cibler les questions d'optimisations de noyaux
- Mais surtout de réfléchir à la composition d'algorithmes parallèles pour la programmation GPGPU

Diffusion

- Déjà présenté plusieurs fois
- A différentes communautés
- Déjà des utilisateurs
- SPOC : <http://www.algo-prog.info/spoc/>
- installable via OPAM (OCaml Package Manager)

Enseignement

- Plus simple que les outils classiques
- Libère des questions de transferts
- Permet de cibler les questions d'optimisations de noyaux
- Mais surtout de réfléchir à la composition d'algorithmes parallèles pour la programmation GPGPU