Compilation efficace d’applications de traitement d’images pour processeurs manycore

Soutenance de thèse

Pierre Guillou

mercredi 30 novembre 2016

MINES ParisTech, PSL Research University

Évolution des capacités de calcul

1987

2000

2016

De nouvelles architectures

Limitations des processeurs monocœurs

- fréquence d'horloge — bloquée depuis 2005
- consommation d'énergie — systèmes embarqués
- utilisation efficace des transistors ?

Nouvelles architectures parallèles

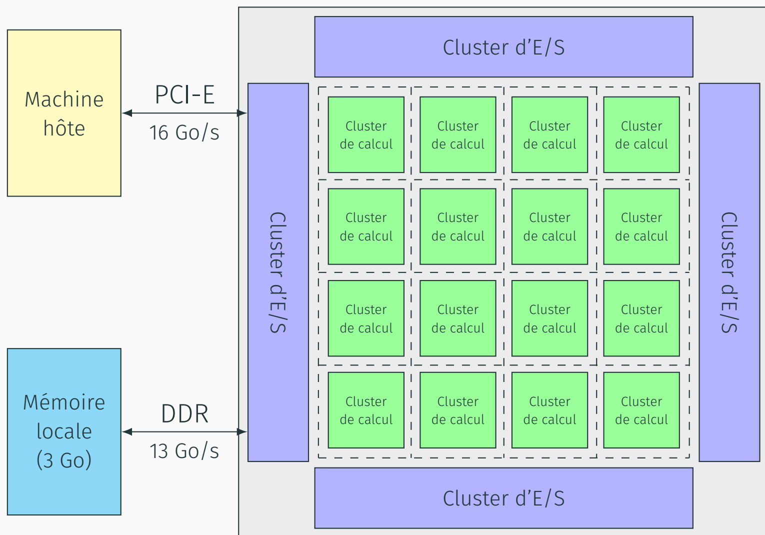
- processeurs multicœurs — jusqu'à 10 cœurs...
- GPGPUs — SIMD, 30 — 4000 cœurs
- processeurs *manycores* — 60 — 300 cœurs

> *The free lunch is over. Now welcome to the hardware jungle.*
> — Herb Sutter, 2011

Les processeurs *manycore* : le MPPA de Kalray

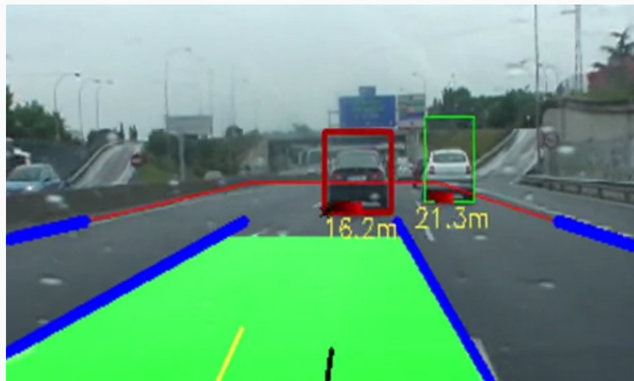
256 cœurs de calcul, 10 W
2 générations : Andey (400 MHz), Bostan (600 MHz)

L'architecture du processeur MPPA



16 clusters × (16 cœurs + 2 Mo)

Le traitement d’images, un domaine innovant et dynamique



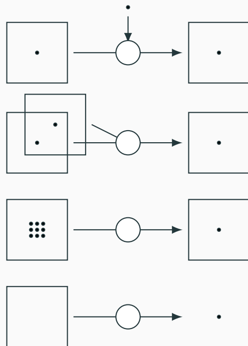
Une branche : la *morphologie mathématique*

Propriétés géométriques

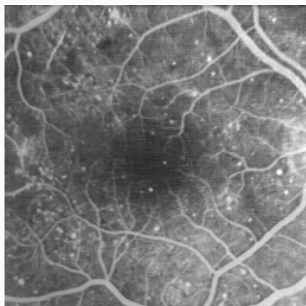
- détection, extraction

Algèbre d'opérateurs images

- réguliers, parallèles
- arithmétiques
  unaires & binaires
- « morphologiques »
  érosion, dilatation, convolution
- réductions
  min, max, somme



licensePlate : détection de plaque d'immatriculation



retina : détection des lésions de la rétine

Difficultés du développement logiciel

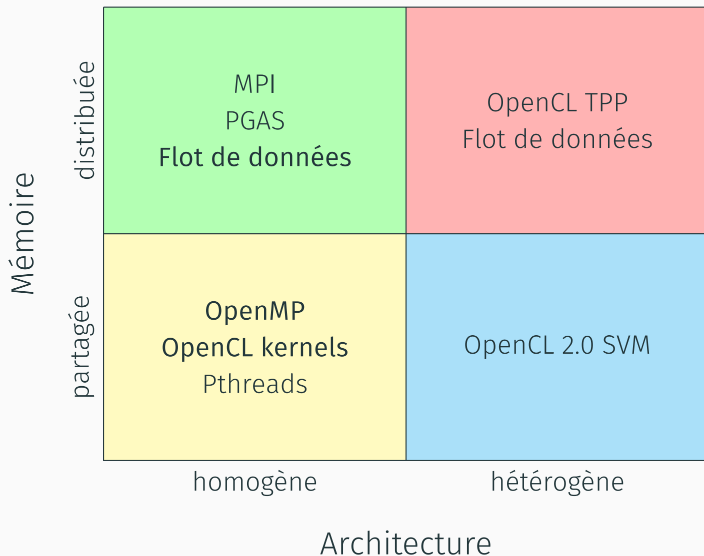
Applications $\longrightarrow$ matériels

- fonctionnalités — temps de développement
- matériels cibles — travail de portage
- optimisations — fluidité, temps de réponse, autonomie

Enjeux : les *3P*

- programmabilité — temps de développement, maintenance
- portabilité — cibles matérielles
- performance — temps d'exécution, énergie

Classes d’architectures et modèles de programmation



Plan

1. Méthodologie de recherche
2. I — Le modèle OpenMP
3. II — Le modèle flot de données
4. III — Le modèle OpenCL
5. Quel modèle choisir ?
6. Conclusions & Perspectives

Méthodologie de recherche

Méthodologie de recherche

Sujet

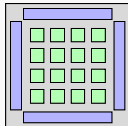
- compilation efficace — 1 programme $\longrightarrow$ *3P*
- traitement d'images — utile, parallèle
- architectures manycore — complexité vs. énergie

Étude de cas

- le processeur manycore MPPA de Kalray
- le traitement d'images par morphologie mathématique
- des langages de traitement d'images (DSLs $\rightarrow$ *3P*)

Contributions principales

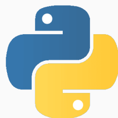
Comparaisons expérimentales sur MPPA



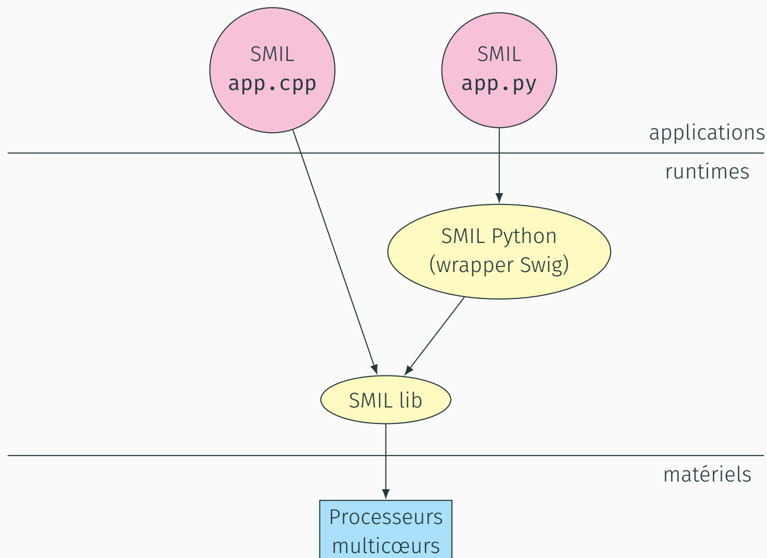
- 3 modèles de programmation parallèle
- 2 générations de processeurs

Preuve de la possibilité d'atteindre les *3P*

- développement d'un environnement logiciel intégré
- regroupant plusieurs chaînes de compilation
- restreint au traitement d'images

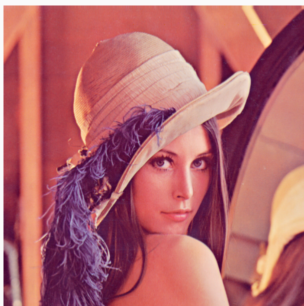


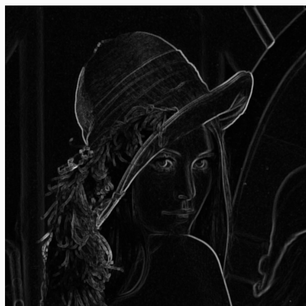
SMIL : Simple (but efficient) Morphological Image Library

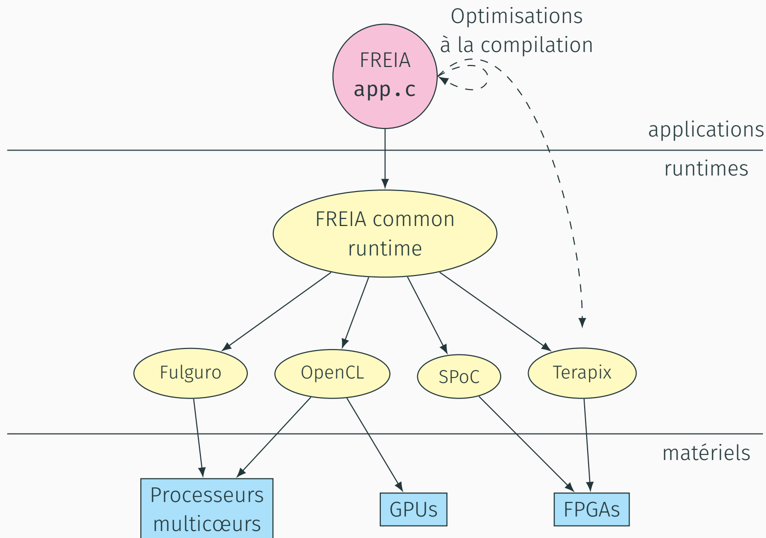


Exemple de code SMIL

```python
import smilPython as smil

imin = smil.Image("input.png")  # lecture sur le disque
imout = smil.Image(imin)        # allocation de imout
smil.gradient(imin, imout)      # gradient morphologique
imout.save("output.png")        # écriture sur le disque
```






Optimisations
à la compilation
FREIA
app.c
applications
runtimes
FREIA common
runtime
Fulguro
OpenCL
SPoC
Terapix
matériels
Processeurs
multicœurs
GPUs
FPGAs

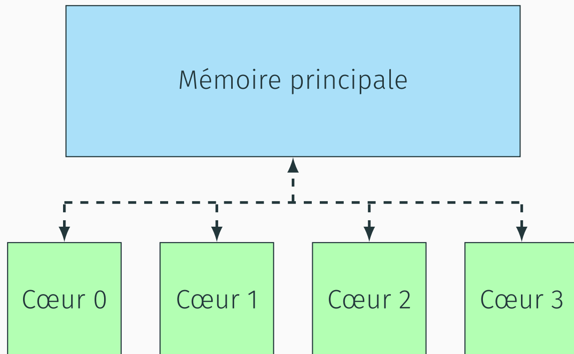
PIPS pour FREIA

Compilateur source-à-source

- entrée DSL : code C + appels FREIA
- optimisations spécifiques traitement d'images :
  - décomposition des opérateurs complexes
  - élimination des temporaires
  - élimination des sous-expressions communes
  - propagation des copies
  - fusion d'opérateurs
- sortie, génération de code :
  - FREIA simplifié
  - SPoC, Terapix (FPGAs)
  - OpenCL
  - Sigma-C

I — Le modèle OpenMP

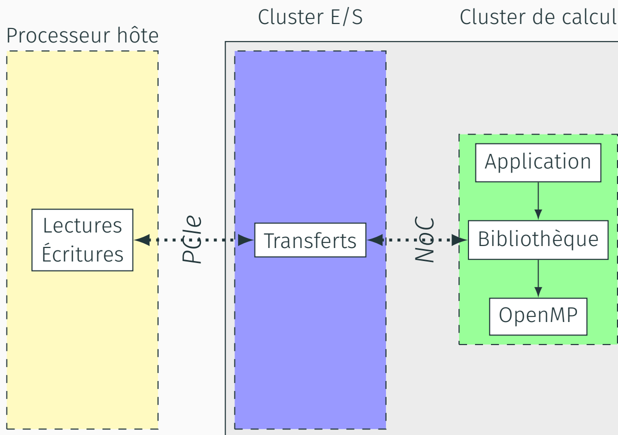
OpenMP, un modèle pour architectures à mémoire partagée



Addition de vecteurs en OpenMP

```
void vecAdd(float *a, float *b, float *c,
            const unsigned int n) {

  unsigned int i;

#pragma omp parallel for
  for (i = 0; i < n; i++)
    c[i] = a[i] + b[i];

}
```

Environnement d'exécution OpenMP : 1 cluster

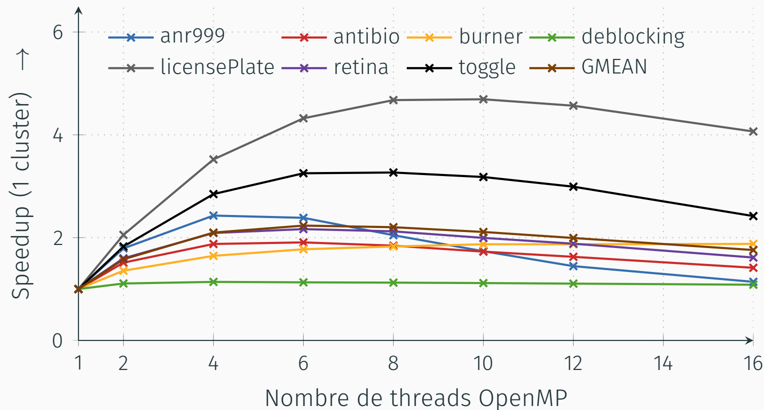


Portage SMIL sur 1 cluster MPPA

Simple* Morphological Image Library
(*but efficient)

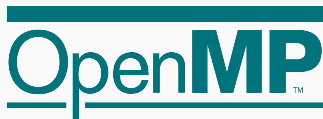
SMIL	bibliothèque C++, OpenMP	*standard*
CMake	compilation croisée	*portage rapide*
transferts	avec l'hôte	*3 environnements*
mémoire	très limitée : 2 Mo	*imagettes*

Évaluation : passage à l'échelle de SMIL avec OpenMP



petites images

Bibliothèque OpenMP sur MPPA



SMIL $\longrightarrow$ MPPA

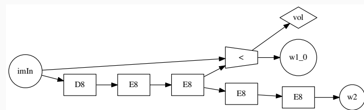
- portage direct et rapide — compilation croisée
- 1 cluster de calcul / 16 — modèle mémoire
- environnements d'exécution — transferts de données
- passage à l'échelle
- limitations mémoire — imagettes

$\longrightarrow$ mixer avec des communications inter-clusters

$\longrightarrow$ réduire l'empreinte mémoire

II — Le modèle flot de données

Le flot de données



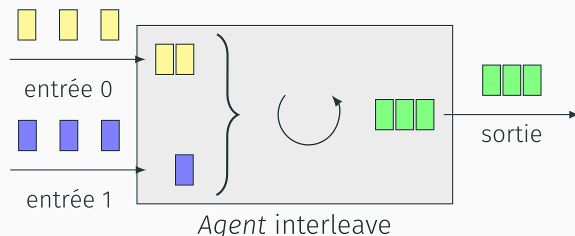
Le modèle et les langages

- graphe producteurs/consommateurs
- années 1970 — KPN
- adapté aux exigences industrielles — Lustre/SCADE
- adapté aux architectures parallèles — StreamIt

Sigma-C, un langage flot de données

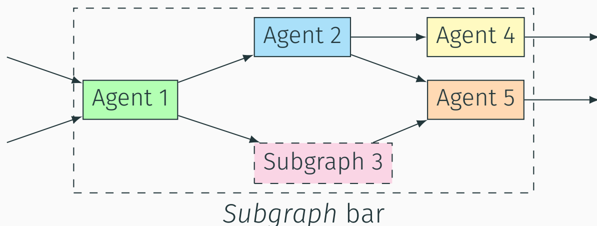
- développé pour le MPPA — Andey, ~~Bostan~~
- CEA List

Agents Sigma-C




```
agent interleave() {
  interface {
    in<int> in0, in1;
    out<int> out0;
    spec{ in0[2], in1, out0[3] };
  }
  void start() exchange(in0 i0[2], in1 i1, out0 o[3]) {
    o[0] = i0[0], o[1] = i1, o[2] = i0[1];
  }
}
```

Subgraphs Sigma-C



Subgraph bar

```
subgraph bar() {
  interface {
    in<int> in0[2]; out<int> out0, out1;
    spec { { in0[][3]; out0 }; { out1[2] } };
  }
  map {
    agent a1 = new Agent1();
    agent a3 = new Subgraph3();
    connect(in0[0], a1.input0);
    connect(a5.output, out1);
    connect(a1.output0, a2.input);
    connect(a3.output, a5.input1);
  }
}
```

Conversion de FREIA en Sigma-C

À partir de code FREIA séquentiel

```
freia_aipo_threshold(im1, im0, 50, 150, true); // arithmétique
freia_aipo_erode_8c(im2, im1, kernel);         // morphologique
freia_aipo_dilate_8c(im3, im2, kernel);        // morphologique
```

Générer automatiquement des subgraphs Sigma-C

```
subgraph foo() {
  int16_t kernel[9] = {0,1,0, 0,1,0, 0,1,0};
  ...
  agent thr = new img_threshold_img();
  agent ero = new img_erode(kernel);
  agent dil = new img_dilate(kernel);
  ...
  connect(thr.output, ero.input);
  connect(ero.output, dil.input);
  ...
}
```

Contraintes sur les applications Sigma-C

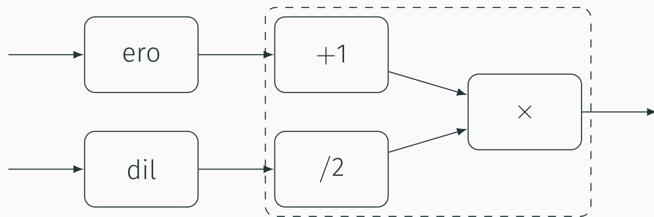
Agents

- 1 agent → 1 cœur de calcul — *utilisation partielle*
- 2 Mo pour 16 cœurs — *mem(1 agent) $\leq$ 128 ko*
- coût d'itération fixe — *grouper les pixels par lignes*

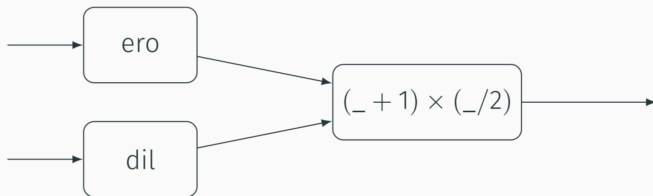
Graphes

- débit : nœud le plus lent — *équilibrage fusion/fission*
- placement : communications inter-clusters — *peu de clusters*
- temps d'activation constant — *gros graphes*

Fusion des opérateurs arithmétiques



Fusion des opérateurs arithmétiques



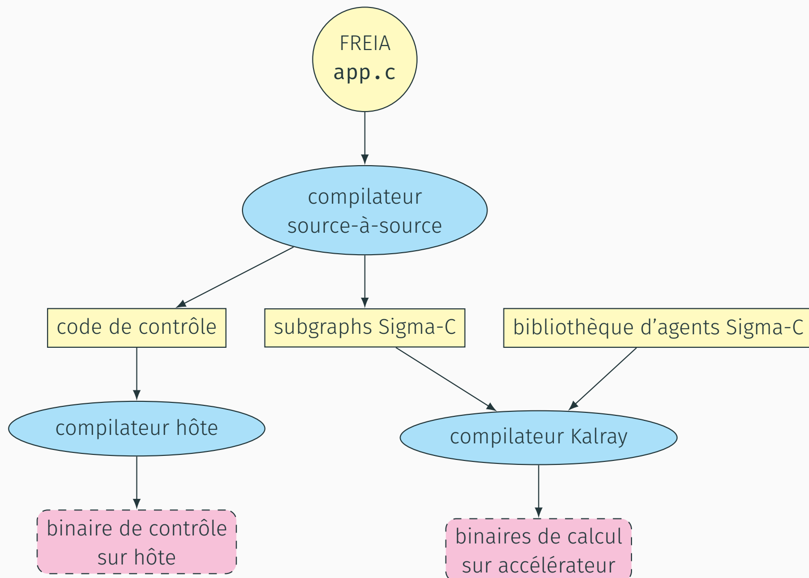
Optimisation : réduction des communications

```c
void freia_cipo_geodesic_reconstruct(freia_data2d *immarker,
                                     freia_data2d *immask) {
  int32_t volcurrent, volprevious;
  freia_global_vol(immarker, &volcurrent);
  do {
    volprevious = volcurrent;
    freia_dilate(immarker, immarker);
    freia_inf(immarker, immarker, immask);
    freia_global_vol(immarker, &volcurrent);
  } while (volcurrent != volprevious);
}
```

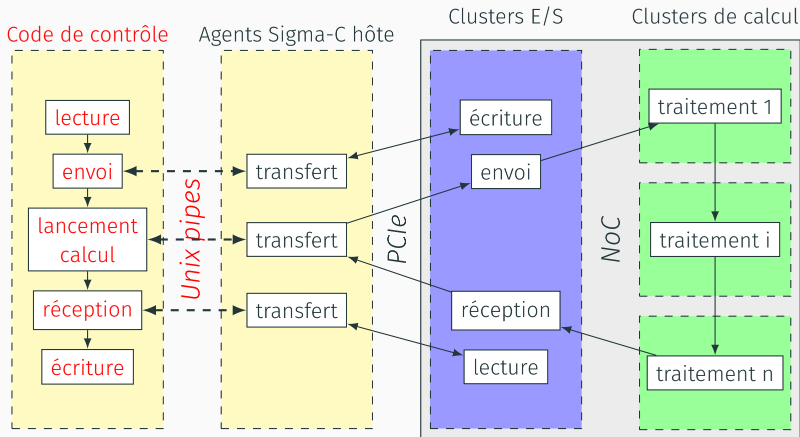
- pas de boucle dynamique en Sigma-C $\longrightarrow$ `while` sur hôte
- reconstruction géodésique : suite stationnaire d'itérations

$\Longrightarrow$ déroulement partiel

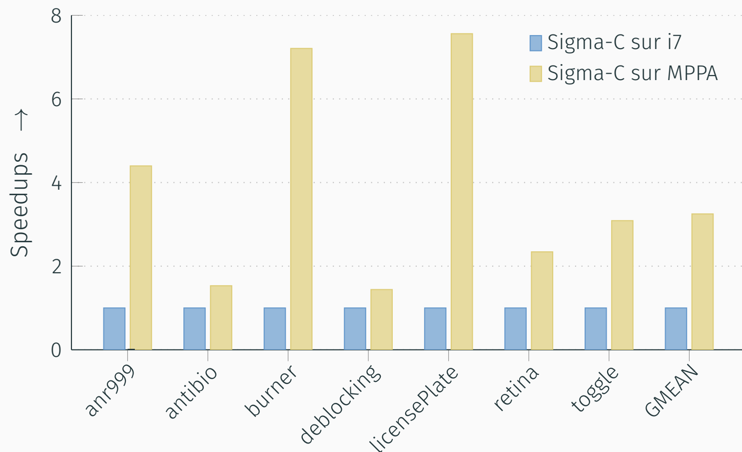
Chaîne de compilation FREIA → Sigma-C



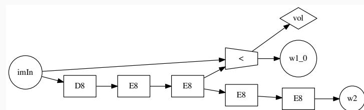
Environnement d'exécution Sigma-C



Évaluation : Sigma-C sur multicœur et MPPA



Flot de données et processeurs manycore



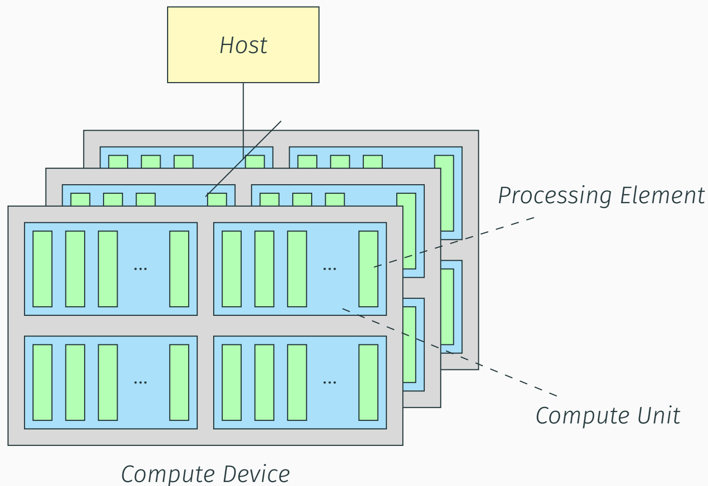
FREIA → Sigma-C → MPPA

- bibliothèque d'opérateurs
- compilation source-à-source FREIA → Sigma-C
- environnement d'exécution pour E/S
- optimisations pour le processeur MPPA

Résultats

- modèle strict — morceaux d'applications
- compilation automatique — *3P*
- bonnes performances sur MPPA — malgré usage partiel
- *publication LCPC 2014*

III — Le modèle OpenCL


Host
...
...
...
...
Processing Element
Compute Unit
Compute Device

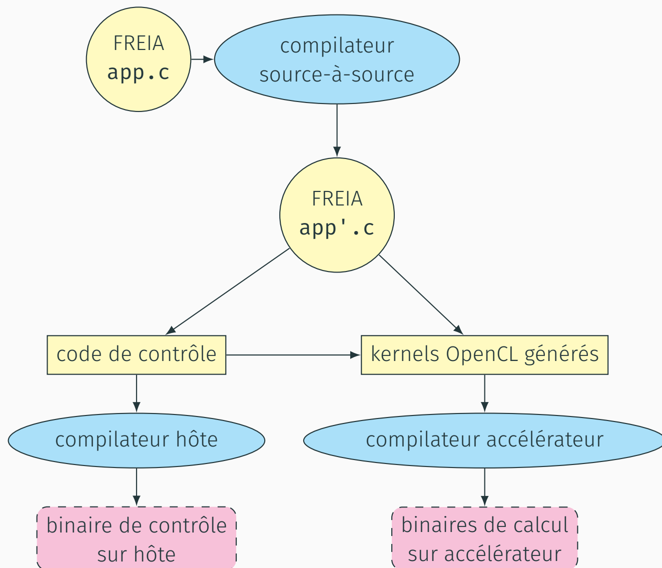
Addition de vecteurs en OpenCL : noyau de calcul

```
__kernel void vecAdd(__global float *a,
                     __global float *b,
                     __global float *c,
                     const unsigned int n) {

  unsigned int id = get_global_id(0);

  if (id < n)
    c[id] = b[id] + a[id];
}
```

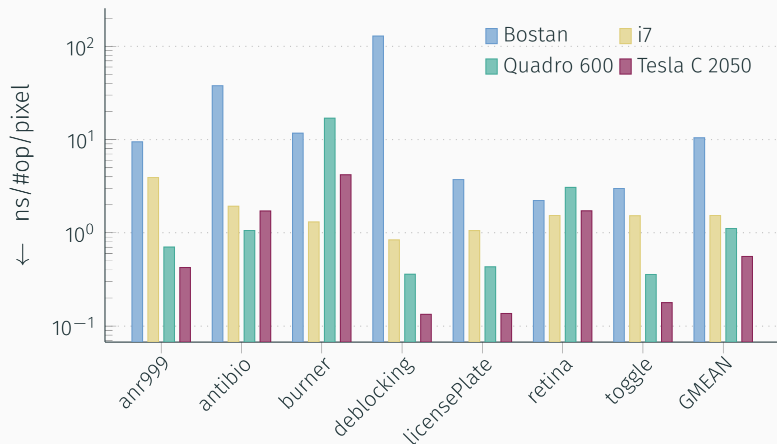
Addition de vecteurs en OpenCL : code hôte

```
// Initializations
clGetPlatformIDs(1, &cpPlatform, NULL);
clGetDeviceIDs(cpPlatform, CL_DEVICE_TYPE_GPU, 1, &device_id, NULL);
ctx = clCreateContext(0, 1, &device_id, NULL, NULL, &err);
q = clCreateCommandQueue(ctx, device_id, 0, &err);

// Create the compute program from the source buffer
prgm = clCreateProgramWithSource(ctx, 1, (const char **)&kernelSource, NULL, &err);
clBuildProgram(prgm, 0, NULL, NULL, NULL, NULL);

// Create the compute kernel in the program we wish to run
knl = clCreateKernel(prgm, "vecAdd", &err);

// Create the input and output arrays in device memory
d_a = clCreateBuffer(ctx, CL_MEM_READ_ONLY, bytes, NULL, NULL);
d_b = clCreateBuffer(ctx, CL_MEM_READ_ONLY, bytes, NULL, NULL);
d_c = clCreateBuffer(ctx, CL_MEM_WRITE_ONLY, bytes, NULL, NULL);

// Write our data set into the input array in device memory
err = clEnqueueWriteBuffer(q, d_a, CL_TRUE, 0, bytes, h_a, 0, NULL, NULL);
err |= clEnqueueWriteBuffer(q, d_b, CL_TRUE, 0, bytes, h_b, 0, NULL, NULL);

// Set the arguments to our compute kernel
err |= clSetKernelArg(knl, 0, sizeof(cl_mem), &d_a);
err |= clSetKernelArg(knl, 1, sizeof(cl_mem), &d_b);
err |= clSetKernelArg(knl, 2, sizeof(cl_mem), &d_c);
err |= clSetKernelArg(knl, 3, sizeof(unsigned int), &n);

// Execute the kernel over the entire range of the data set
err = clEnqueueNDRangeKernel(q, knl, 1, NULL, &globalSize, &localSize, 0, NULL, NULL);
clFinish(q);

// Read the results from the device
clEnqueueReadBuffer(q, d_c, CL_TRUE, 0, bytes, h_c, 0, NULL, NULL);
```

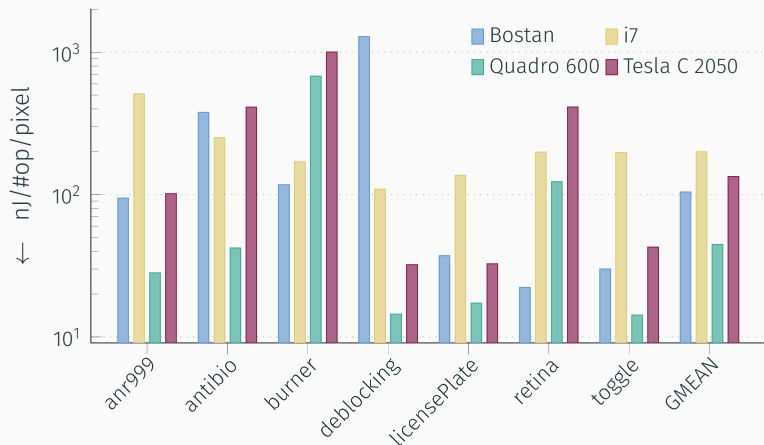
Chaîne de compilation FREIA → OpenCL



OpenCL sur le MPPA, multicœur et GPU : temps d'exécution



OpenCL sur le MPPA, multicœur et GPU : énergie



Le modèle OpenCL

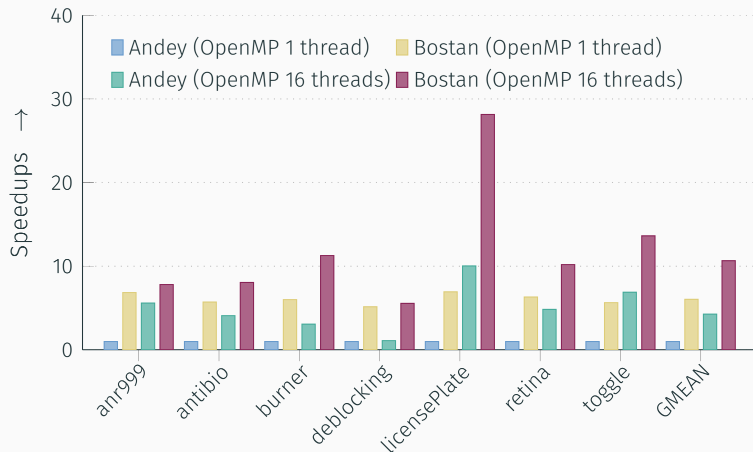
OpenCL ⟶ MPPA

- modèle répandu
- E/S gérées par le modèle
- support partiel d'OpenCL — évolutions de la pile logicielle
- performances à améliorer — accès à la mémoire globale ?

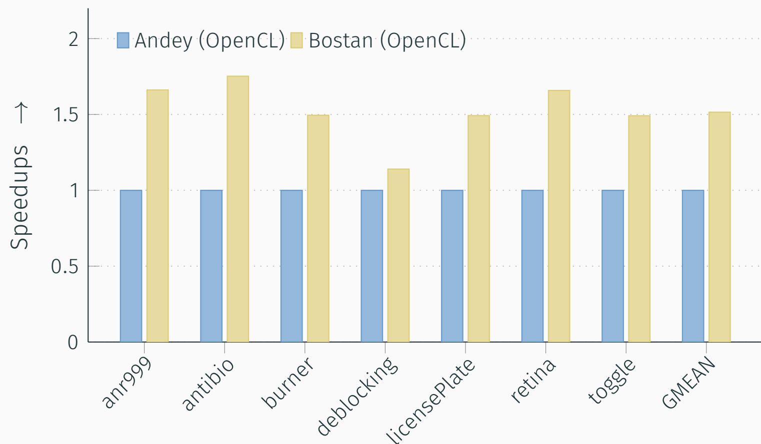
⟶ à éviter actuellement sur MPPA pour ce type d'applications

Quel modèle choisir ?

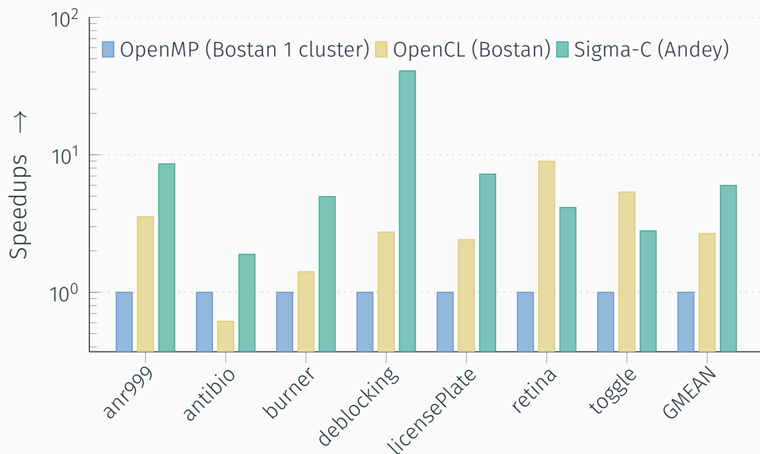
Andey vs. Bostan : OpenMP sur 1 cluster



OpenCL sur MPPA : Andey vs. Bostan



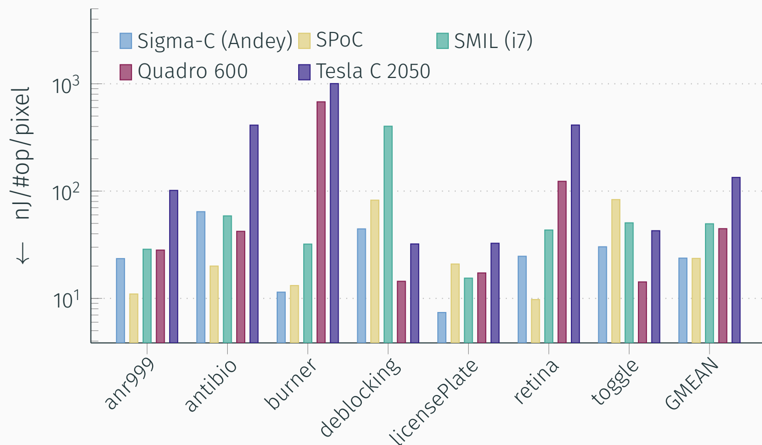
Le meilleur modèle de programmation sur MPPA ?



Comparaison des modèles

Modèle	Avantages	Inconvénients
OpenMP	• répandu • portage rapide	• 16 cœurs sur 256 • mémoire : 2 Mo (code, données) • transferts depuis l'hôte
Sigma-C	• performant • communications	• trop statique • contrôle hôte • nouveau langage • abandonné
OpenCL	• répandu • portage rapide	• implémentation en cours • accès mémoire • performances ?
OpenMP + IPC	• 256 cœurs • performances ?	• tuilage • communications

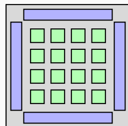
Comparaison avec d’autres accélérateurs



Conclusions & Perspectives

Contributions principales

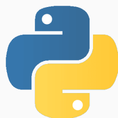
Comparaisons expérimentales sur MPPA



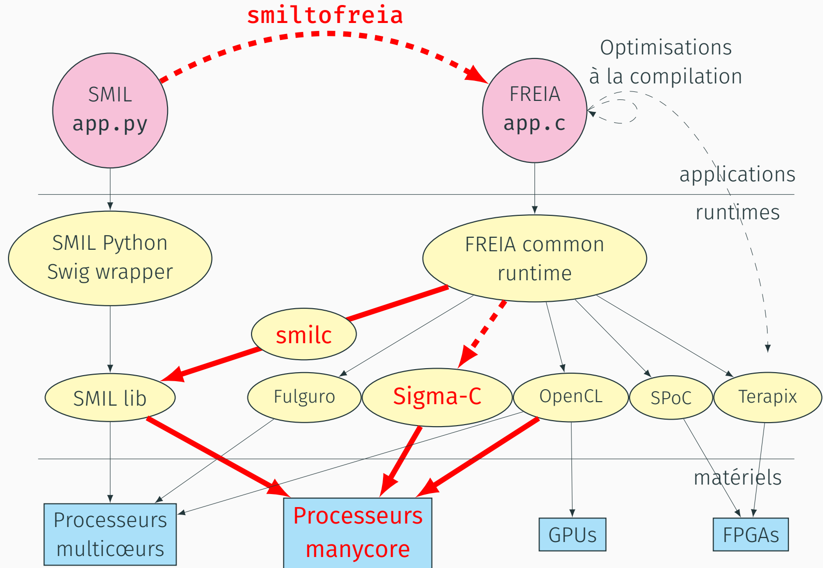
- 3 modèles de programmation parallèle
- 2 générations de processeurs

Preuve de la possibilité d'atteindre les *3P*

- développement d'un environnement logiciel intégré
- regroupant plusieurs chaînes de compilation
- restreint au traitement d'images
- *publication CPC 2016*

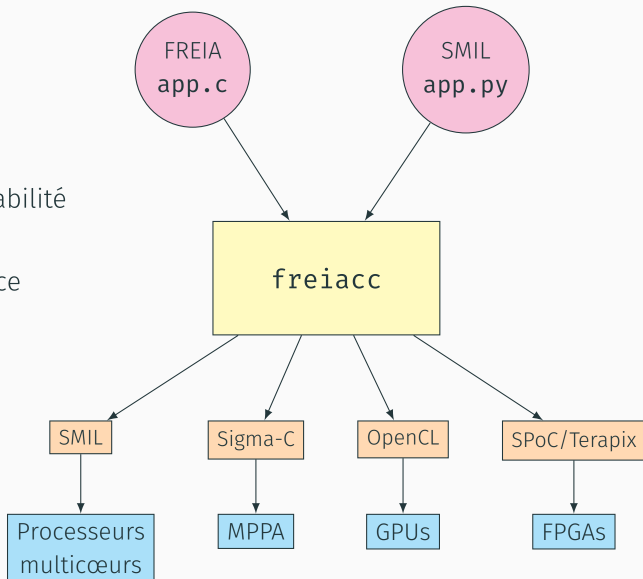


Les piles logicielles SMIL et FREIA



Environnement logiciel : `freiacc` 1 app ⟶ *3P*

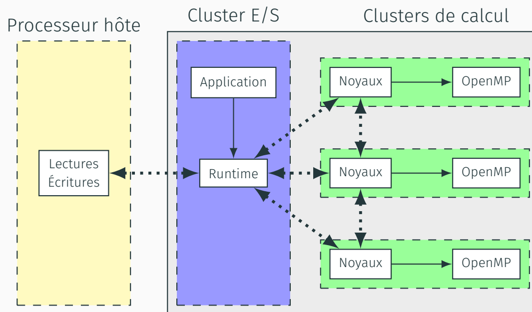
- ✓ Programmabilité
- ? Performance
- ✓ Portabilité



Perspective : autre modèle pour le MPPA

Objectif : utiliser tous les cœurs du MPPA

- OpenMP sur les clusters en C : réécriture des noyaux
- environnement d'exécution : transferts, tuilage, recouvrements
- communications inter-clusters (à la MPI)
- optimisations via PIPS (fusion d'opérateurs, etc.)



Perspective : DSLs et nouvelles architectures

Le processeur MPPA

- potentiel embarqué vs. complexité architecturale *ok!*
- mémoire : taille SMEM, accès DDR…

Traitement d'image

- portage d'algorithmes non locaux *segmentation*

Langages spécifiques à un domaine — DSLs

- *3P* : programmabilité, portabilité, performance
- normalisation, basés sur des langages généralistes

Compilation efficace d'applications de traitement d'images pour processeurs manycore

Soutenance de thèse

Pierre Guillou

mercredi 30 novembre 2016

MINES ParisTech, PSL Research University

Bibliographie personnelle I

Pierre Guillou. *Compilation d'applications de traitement d'images sur architecture MPPA-Manycore.* Conférence d'informatique en Parallélisme, Architecture et Système (ComPAS 2014). Poster. Avr. 2014. URL : `https://hal-mines-paristech.archives-ouvertes.fr/hal-01096993`.

Pierre Guillou. *Compiling Image Processing Applications for Many-Core Accelerators.* ACACES Summer School : Eleventh International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems. Poster. Juil. 2015. URL : `https://hal-mines-paristech.archives-ouvertes.fr/hal-01254412`.

Bibliographie personnelle II

Pierre GUILLOU. `smilc` : *A C wrapper for SMIL*. URL : https://scm.cri.ensmp.fr/git/smilc.git.

Pierre GUILLOU, Fabien COELHO et François IRIGOIN. « Languages and Compilers for Parallel Computing : 27th International Workshop, LCPC 2014, Hillsboro, OR, USA, September 15-17, 2014, Revised Selected Papers ». In : sous la dir. de James BRODMAN et Peng TU. Cham : Springer International Publishing, 2015. Chap. Automatic Streamization of Image Processing Applications, p. 224–238. ISBN : 978-3-319-17473-0. DOI : 10.1007/978-3-319-17473-0_15. URL : http://dx.doi.org/10.1007/978-3-319-17473-0_15.

Bibliographie personnelle III

Pierre Guillou et al. « A Dynamic to Static DSL Compiler for Image Processing Applications ». In : *19th Workshop on Compilers for Parallel Computing*. Valladolid, Spain, juil. 2016. URL : `https://hal-mines-paristech.archives-ouvertes.fr/hal-01352808`.

Nelson Lossing, Pierre Guillou et François Irigoin. « Effects Dependence Graph : A Key Data Concept for C Source-to-Source Compilers ». In : *2016 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)* (oct. 2016).

Benoît Pin, Pierre Guillou et Fabien Coelho. *`smiltofreia` : A SMIL Python to FREIA C compiler*. URL : `https://scm.cri.ensmp.fr/git/smilpyc.git`.