Parallélisation d'applications de traitement d'images vers processeur Kalray MPPA

Journées de la compilation 2019

Pierre Guillou

Dammarie-les-Lys, 30 janvier 2019

(Centre de recherche en informatique, MINES ParisTech, PSL Université Paris)

Plan

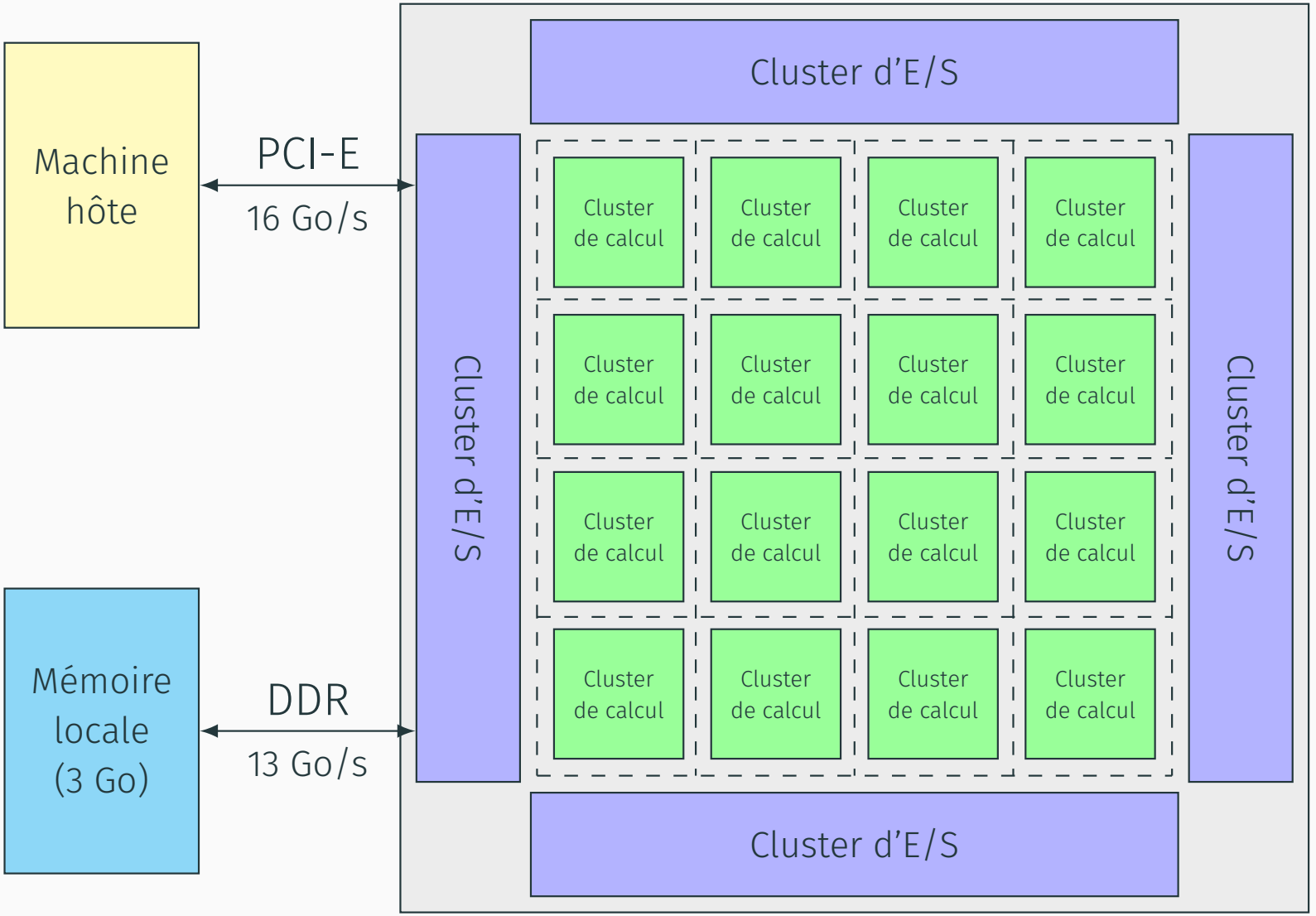
Introduction : cibles et contexte

Les processeurs *manycore* : le MPPA de Kalray

256 cœurs de calcul, 10 W

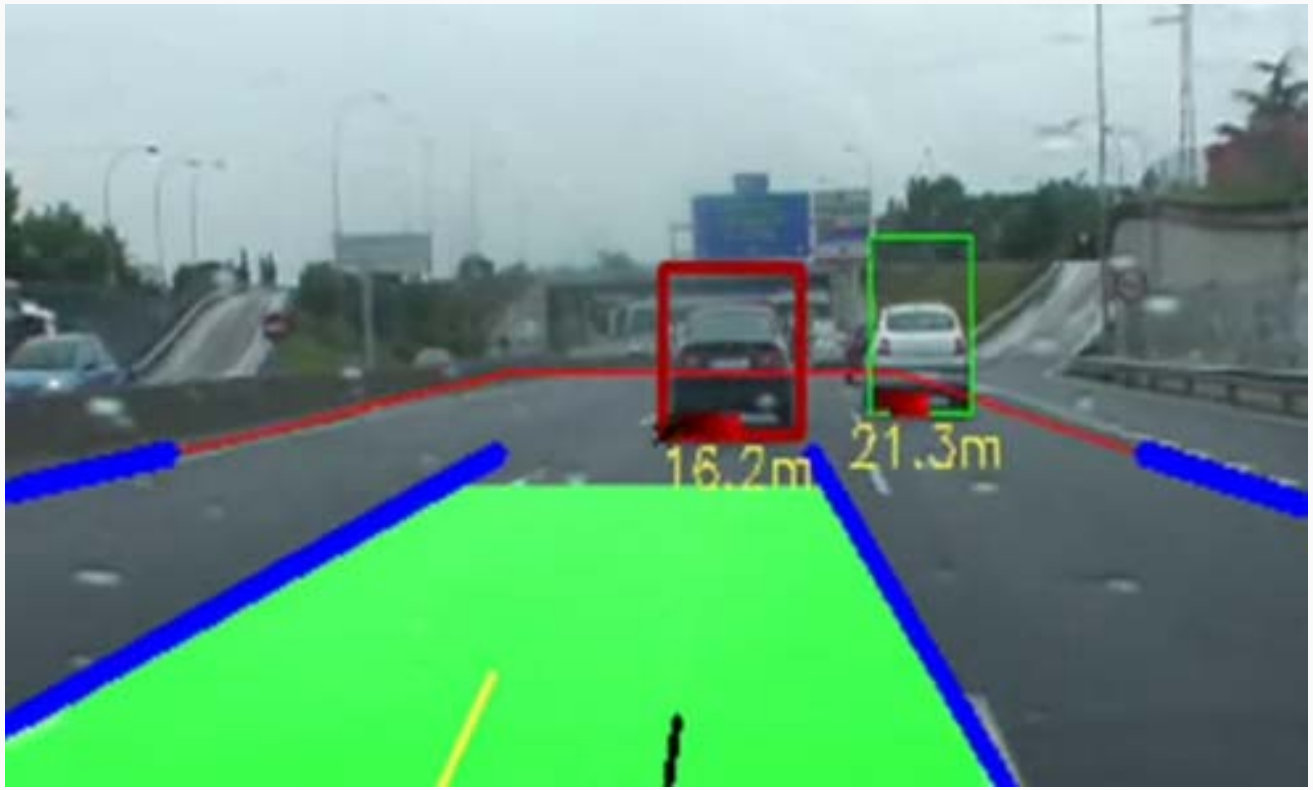
2 générations : Andey (400 MHz), Bostan (600 MHz)

L'architecture du processeur MPPA



16 clusters × (16 cœurs + 2 Mo)

Le traitement d'images, un domaine innovant et dynamique



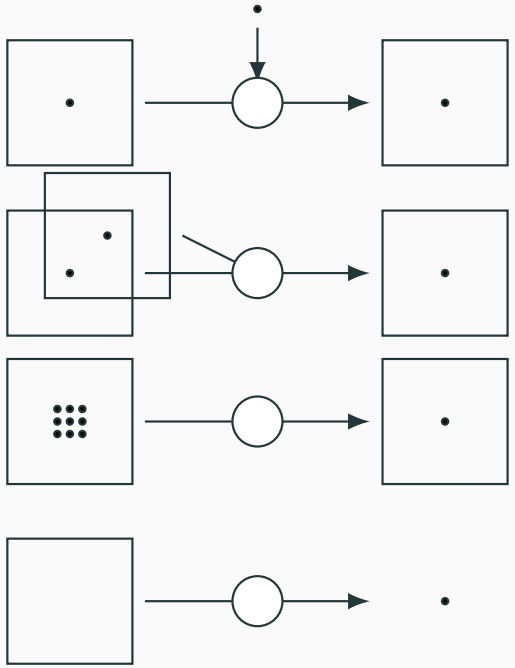
Une branche : la *morphologie mathématique*

Propriétés géométriques

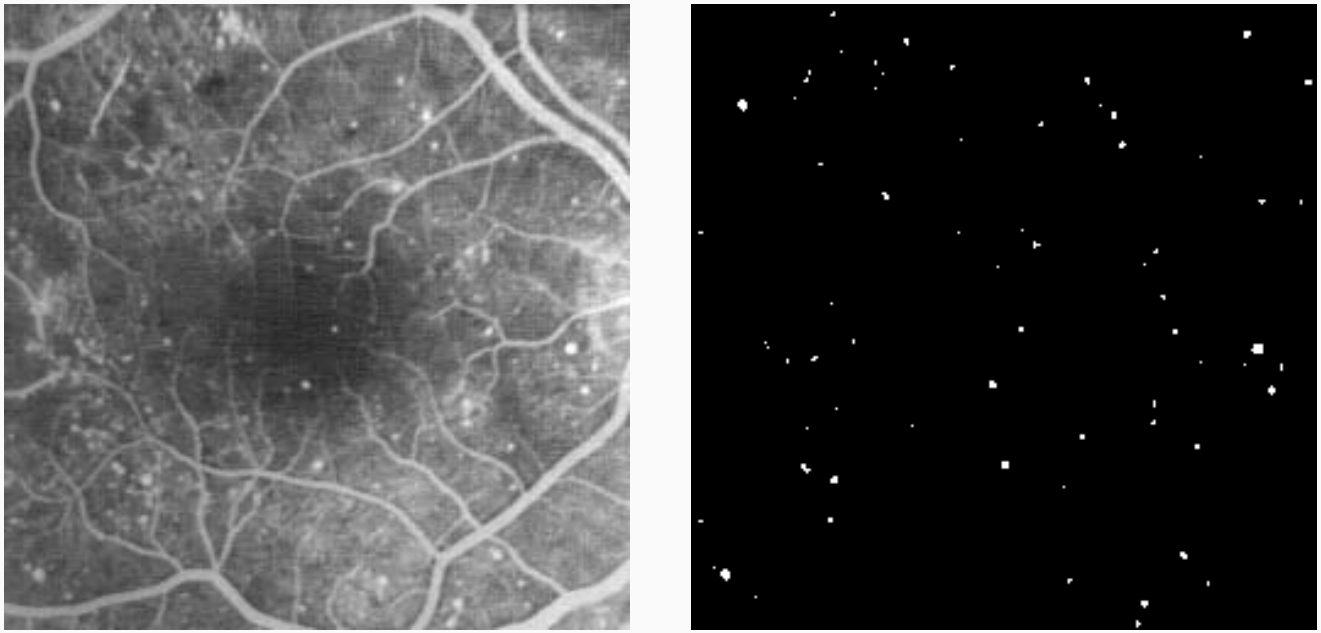
- détection, extraction

Algèbre d'opérateurs images

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

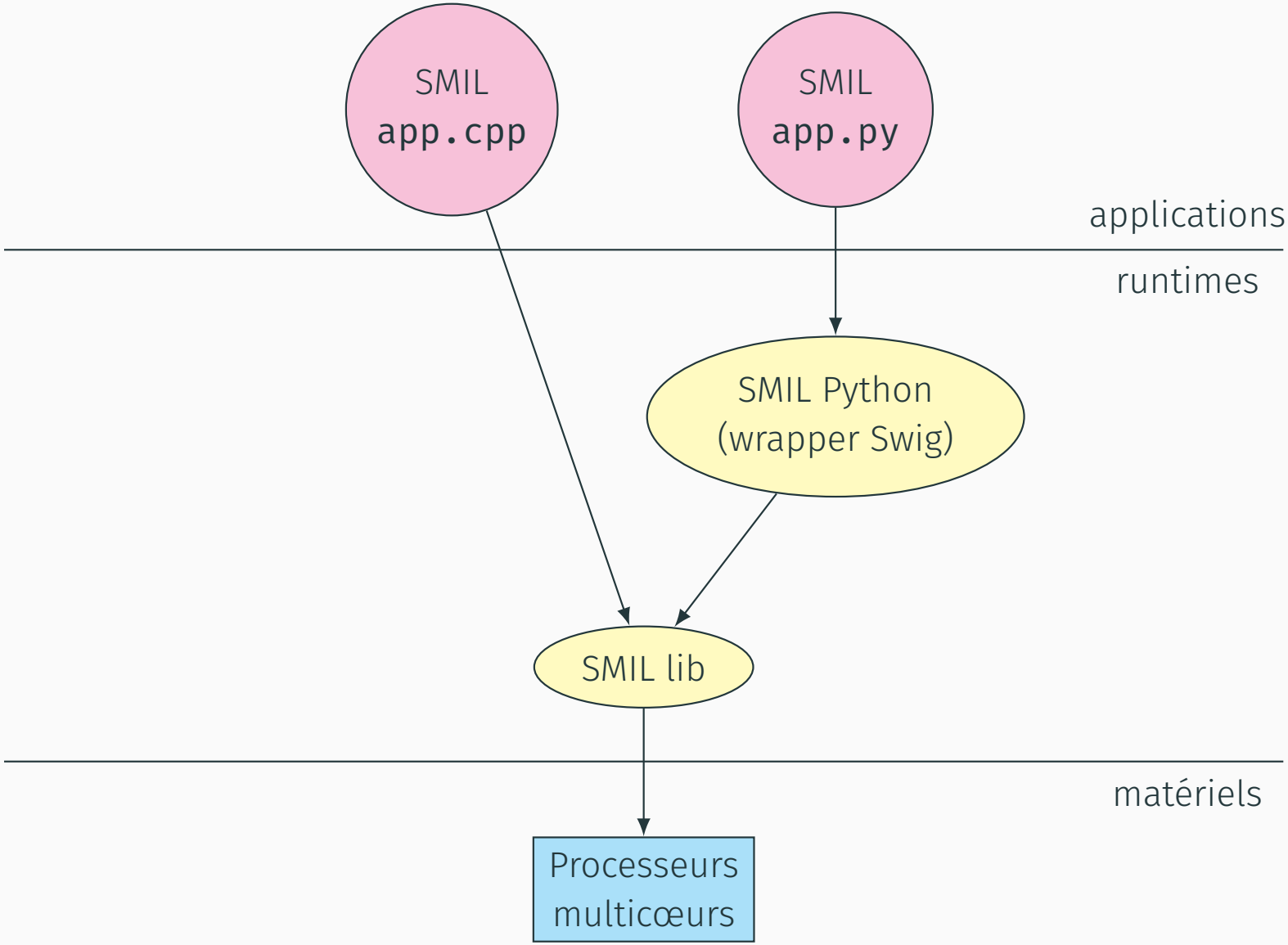


licensePlate : détection de plaque d'immatriculation



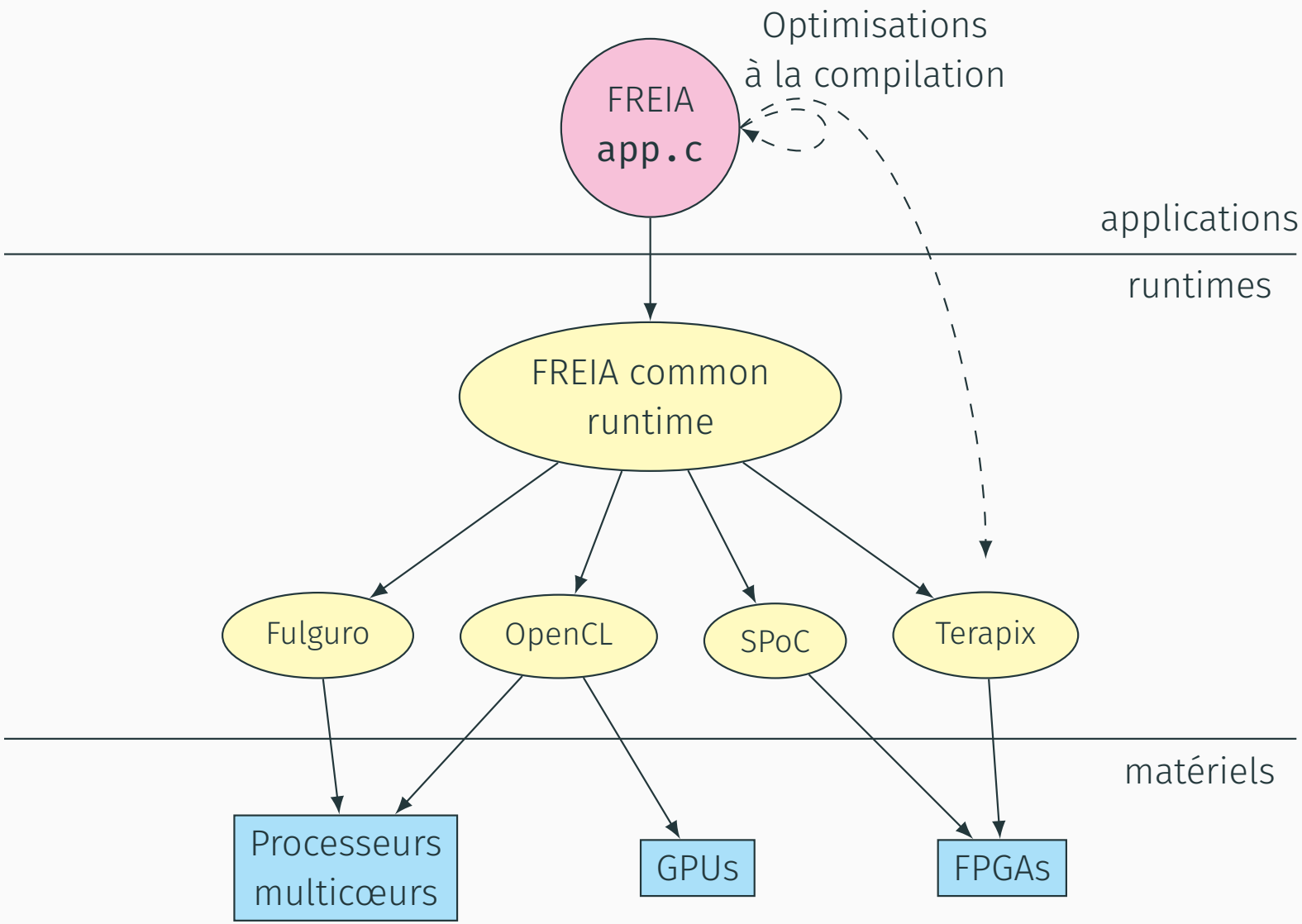
retina : détection des lésions de la rétine

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

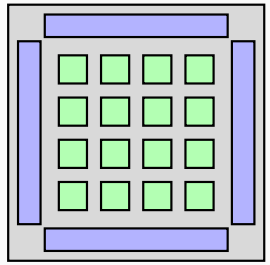
Compilateur source-à-source

- 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

Dans les épisodes précédents...

Thèse : contributions principales

Comparaisons expérimentales sur MPPA



- 3 modèles de programmation parallèle
- 2 générations de processeurs

Preuve de la possibilité d'atteindre les *3P*
(Programmabilité, Portabilité, Performance)

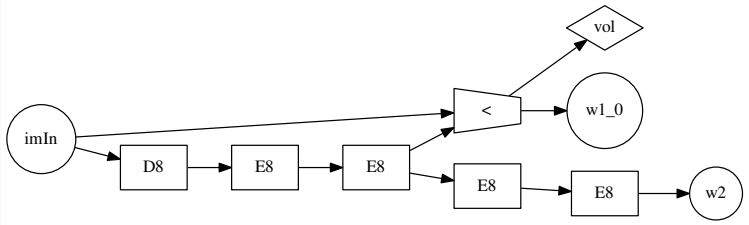
- développement d'un environnement logiciel intégré
- regroupant plusieurs chaînes de compilation
- restreint au traitement d'images

Thèse soutenue en novembre 2016

- Compilation efficace d'applications de traitement d'images pour processeurs manycore [1]

2014 : le modèle flot de données

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
- compilation automatique *3P*
- bonnes performances sur MPPA malgré usage partiel
- *publication LCPC 2014* [2]

2015 : le modèle OpenCL

FREIA $\longrightarrow$ OpenCL $\longrightarrow$ MPPA

- modèle répandu
- E/S gérées par le modèle
- support partiel d'OpenCL
- performances à améliorer

évolutions de la pile logicielle

accès à la mémoire globale ?

$\longrightarrow$ à éviter sur MPPA pour ce type d'applications

2015 : compilation croisée de SMIL

SMIL $\longrightarrow$ MPPA

- portage direct et rapide
- 1 cluster de calcul / 16
- environnement d'exécution
- passage à l'échelle
- limitations mémoire

compilation croisée

modèle mémoire

transferts de données

imagettes

$\longrightarrow$ mixer avec des communications inter-clusters

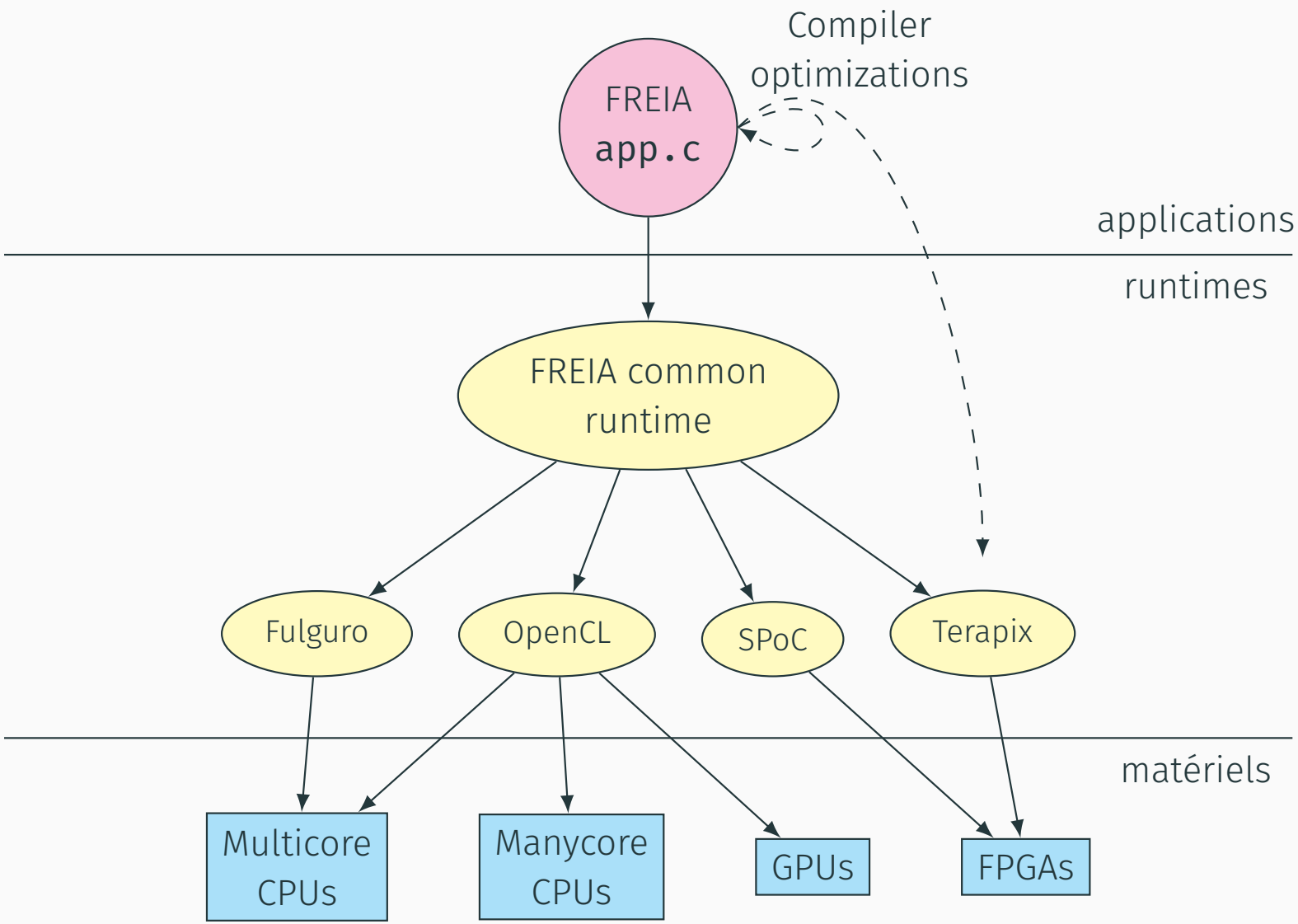
$\longrightarrow$ réduire l'empreinte mémoire

2015 : SMIL nouvelle cible logicielle de FREIA

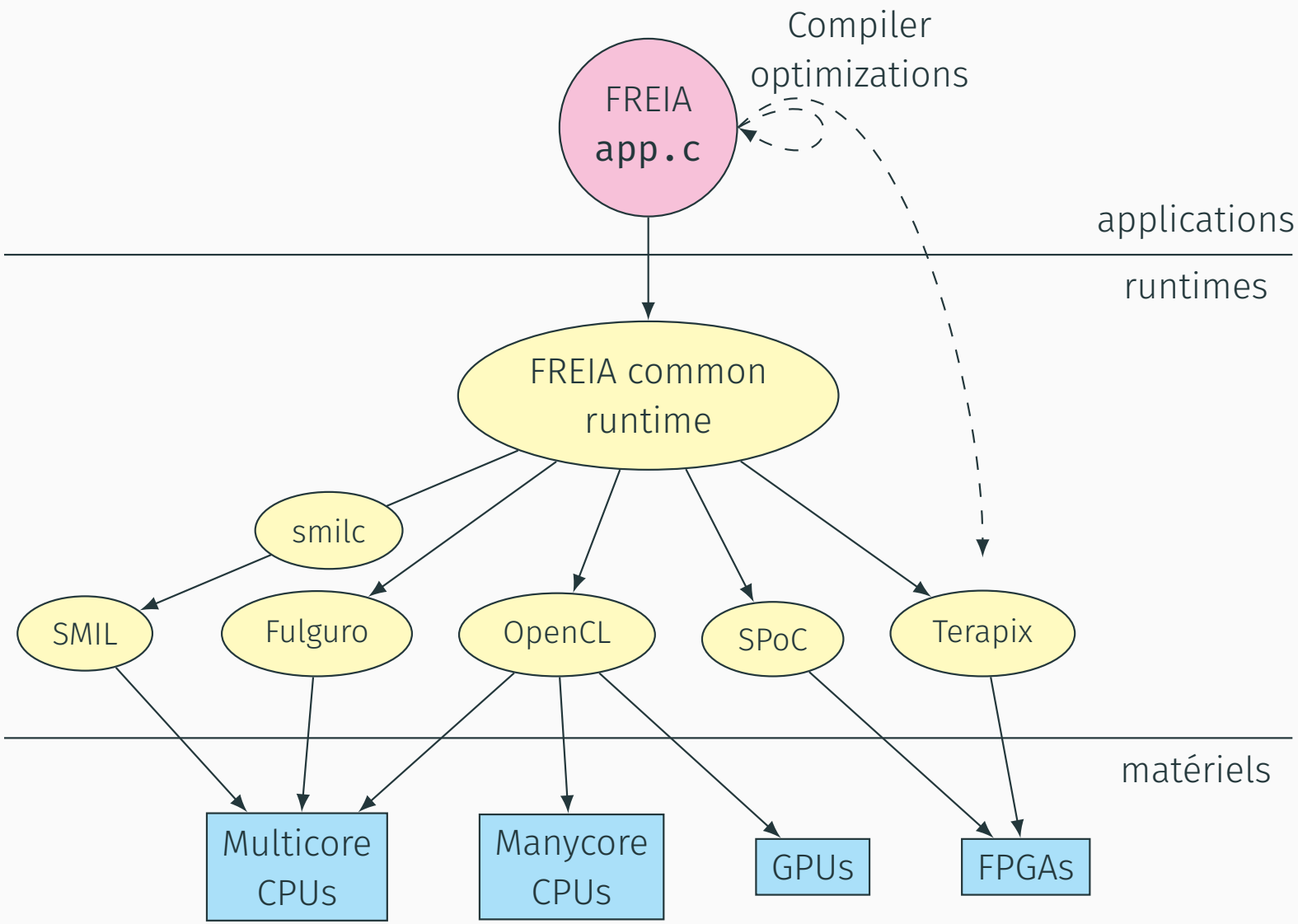
Exécuter des applications FREIA par SMIL

- rediriger les appels atomiques FREIA vers SMIL
- FREIA C, SMIL C++ smilc couche intermédiaire fine
- spécialisation des templates, *handlers* structures opaques...
- comparaison avec Fulguro $\times 13$
- comparaison avec SMIL C++ $\times 0.84$

2015 : SMIL nouvelle cible logicielle de FREIA



2015 : SMIL nouvelle cible logicielle de FREIA



2016 : convertir SMIL en FREIA

SMIL

- \+ interface Python
- \+ optimisée pour multicœurs
- − CPUs

FREIA

- − interface C
- \+ optimisations via PIPS
- \+ CPUs, GPUs, FPGAs…

Combiner la portabilité de FREIA et la programmabilité de SMIL ?

- porter SMIL à la main sur chaque cible — *coûteux*
- ré-implémenter SMIL au-dessus de FREIA — *perte de PIPS*
- supporter SMIL dans PIPS — *très coûteux*
- convertir du code SMIL Python en FREIA C — smiltofreia

2016 : convertir SMIL en FREIA

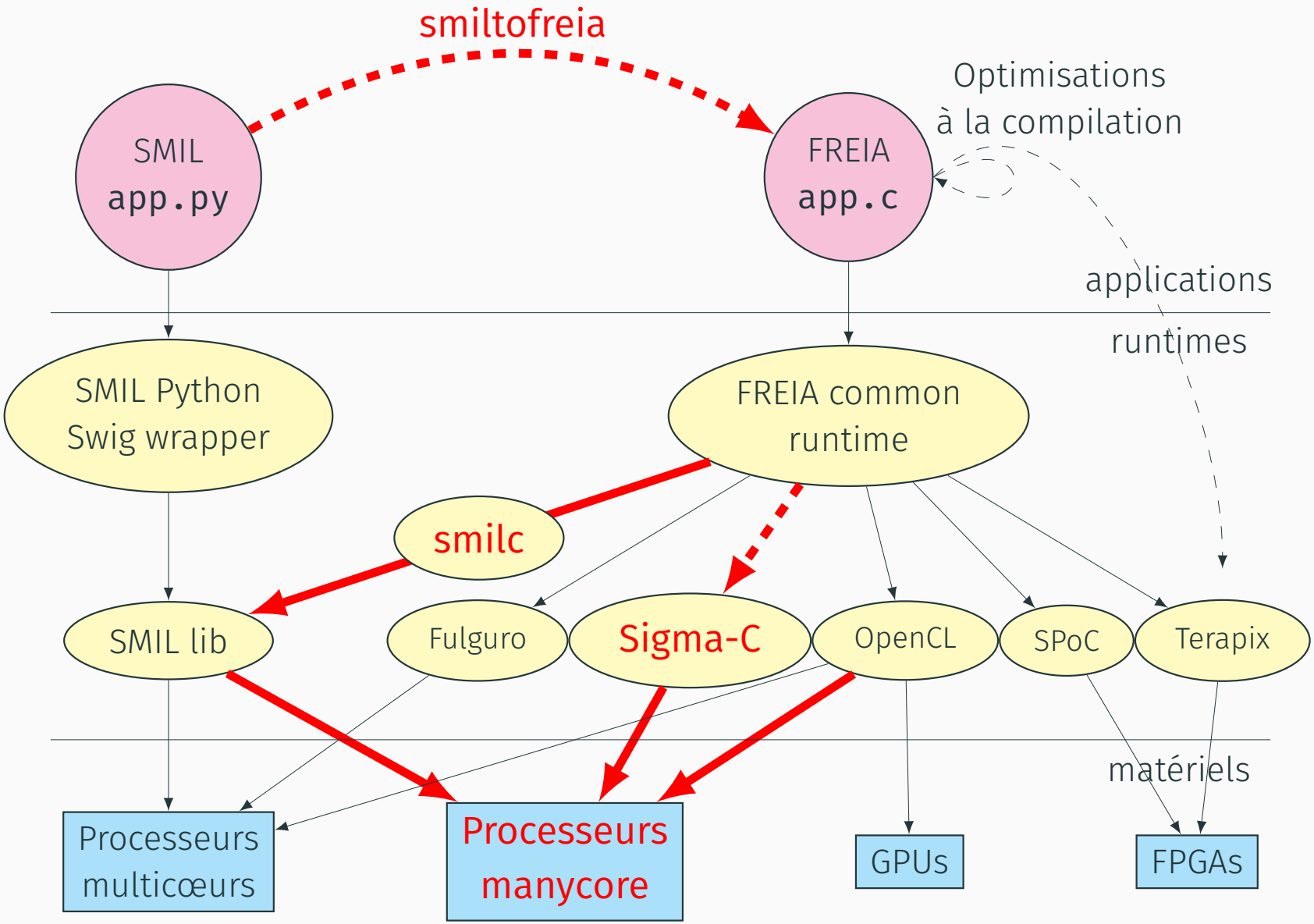
Compilateur de DSLs : smiltofreia

- génère du FREIA à partir d'applications SMIL Python
- écrit en Python
- chaque appel SMIL est converti en son équivalent FREIA
- gestion de la mémoire
- déclarations de variables

Bénéfices de SMIL Python $\longrightarrow$ FREIA C

- portabilité améliorée — *cibles de FREIA*
- grande programmabilité — *API SMIL Python*
- réutilisation d'existant — *compilateur FREIA*
- performance — *proche du code FREIA écrit à la main*
- *présentation CPC 2016* [3]

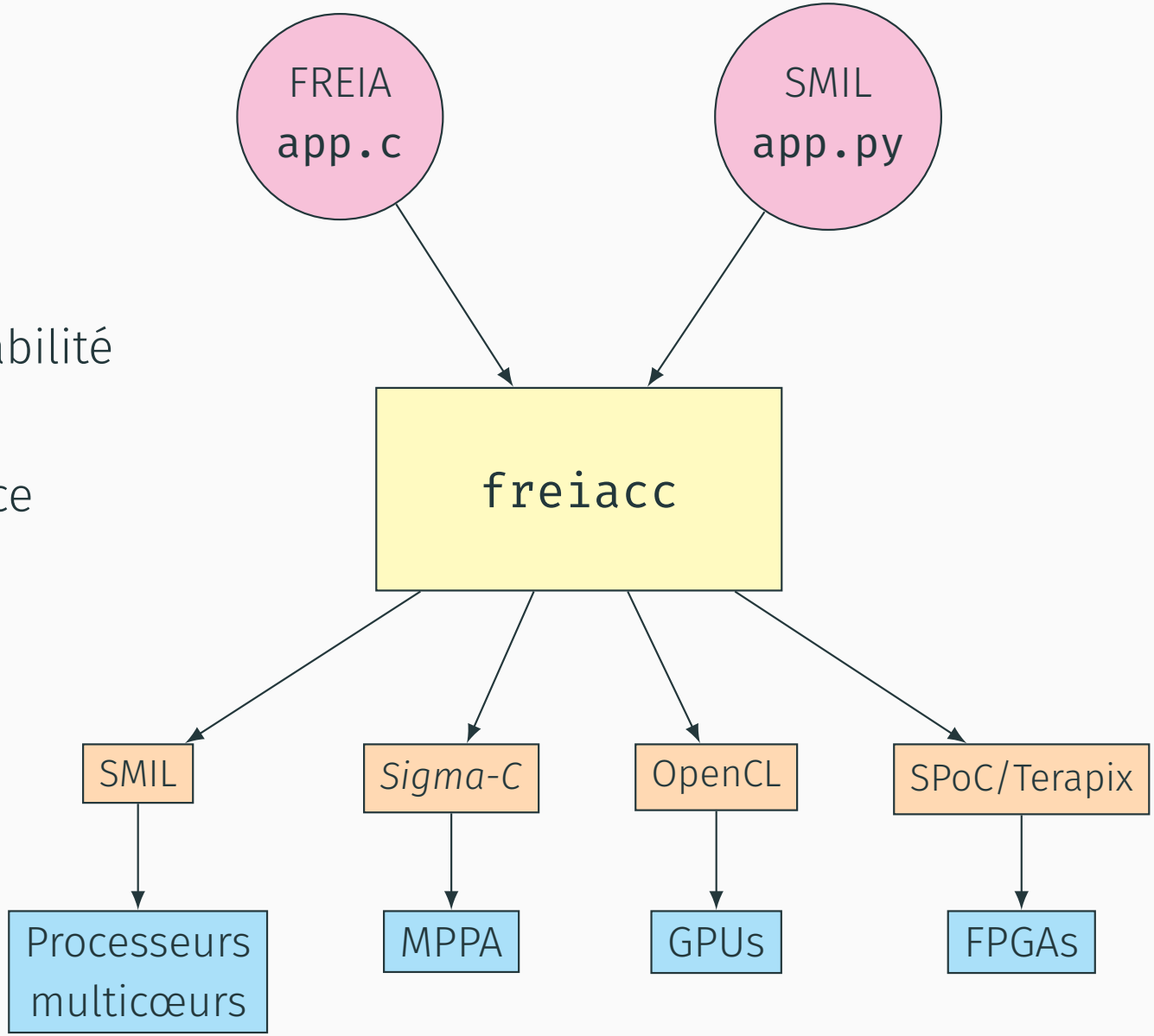
Les piles logicielles SMIL et FREIA



Environnement logiciel : `freiacc`

1 app $\longrightarrow$ *3P*

- ✓ Programmabilité
- ? Performance
- ✓ Portabilité

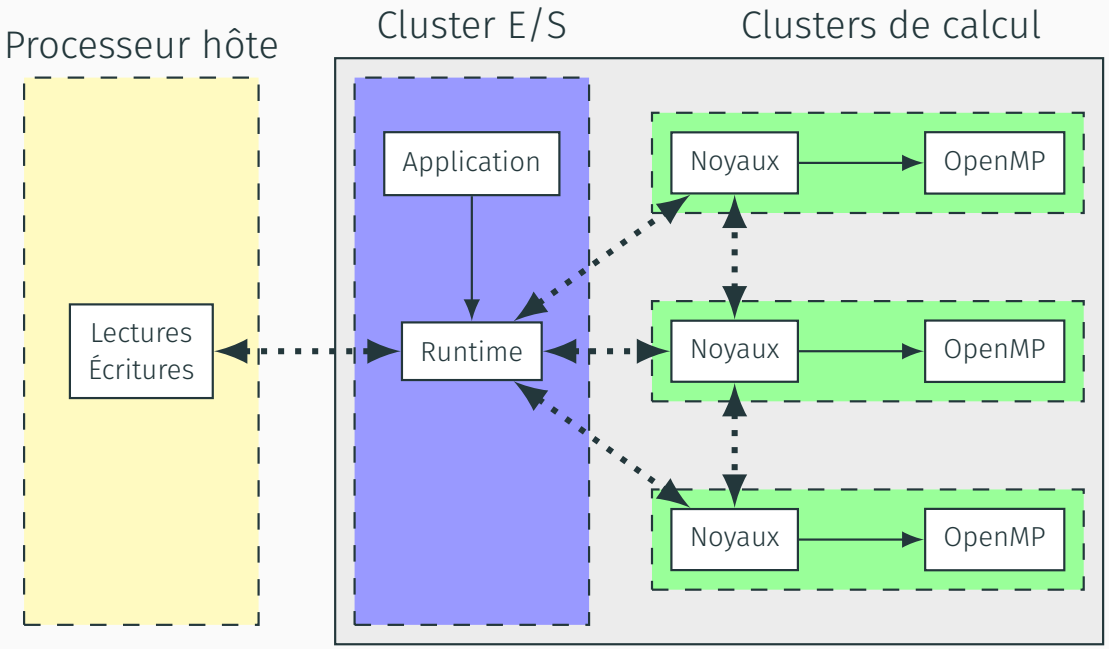


Une dernière cible pour la route

2017 : un environnement d'exécution distribué

Objectif : utiliser tous les cœurs du MPPA

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



Réécrire les noyaux de calcul

Bibliothèque d'opérateurs de traitement d'images

- ∼65 opérateurs arithmétiques, morphologiques, de réduction
- des macros préprocesseur pour factoriser le code
- OpenMP pour paralléliser les boucles de calcul
- interface commune de la forme

  `void mppa_kernel_NAME(mppa_params_t *params)`
- manipulation via un tableau de pointeurs indexé par un enum

```
void mppa_execute(mppa_kernel_enum kernel, mppa_params_t *params) {
  if (kernel != MPPA_KERNEL_NOP)
    mppa_kernel_array[kernel](params);
}
```

2019 : benchmarking des noyaux de calcul

Comparaison avec SMIL

- opérateurs au voisinage : algorithmes différents

Leçons apprises

- même unité de compilation
- même code de benchmark
- comportements différents
  - GCC vs Clang
  - `libgomp` vs `libomp`
- boucles : attention aux casts
  - variables de boucles `unsigned` $\longrightarrow$ `size_t`
- éviter `omp for collapse(2)`

Exemple : seuillage binaire avant/après (i7–7700HQ)

```
void mppa_kernel_threshold_bin(mppa_params_t *params) {
  unsigned int i, j, o;
  PIXEL_T b = params->scalars[0];
  PIXEL_T c = params->scalars[1];
  /* clip parameters to fit in uint8_t range */
  b = b > PIXEL_MAX ? PIXEL_MAX : b;
  c = c < PIXEL_MIN ? PIXEL_MIN : c;
  size_t h = params->height;
  size_t w = params->width;
#pragma omp parallel for collapse(2) private(o)
  for (i = 0; i < h; i++) {
    for (j = 0; j < w; j++) {
      o = i * w + j;
      if (params->in[0][o] >= b && params->in[0][o] <= c)
        params->out[o] = PIXEL_MAX;
      else
        params->out[o] = PIXEL_ZER;
    }
  }
}
```

GCC : 70 µs, Clang : 750 µs

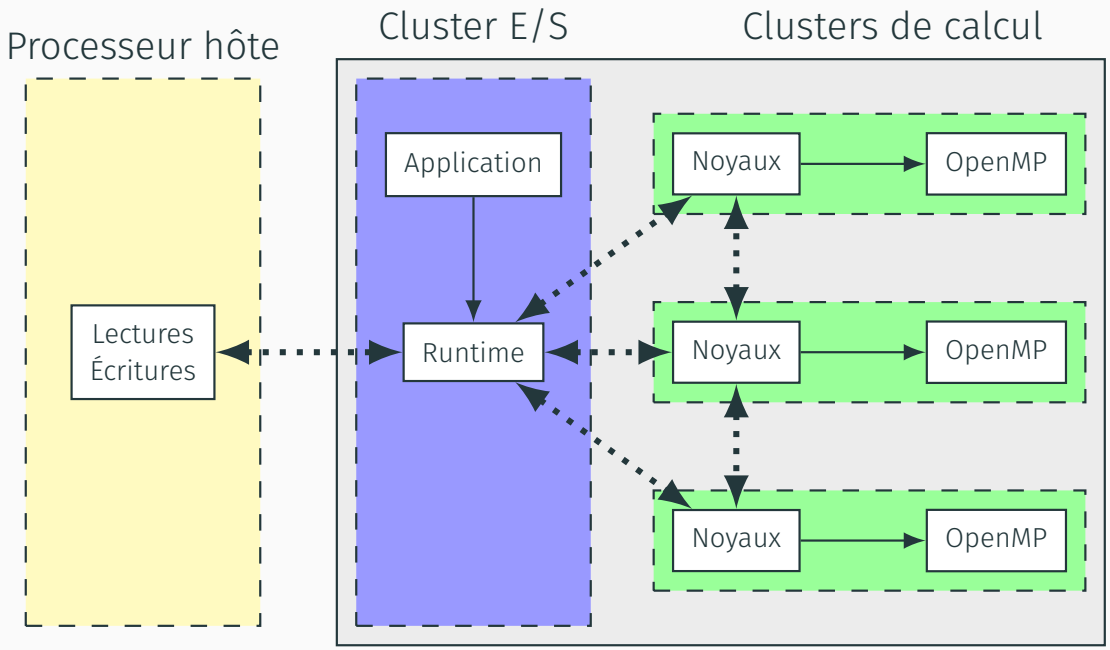
```
void mppa_kernel_threshold_bin(mppa_params_t *params) {
  /* clip parameters to fit in uint8_t range */
  PIXEL_T b = params->scalars[0] % 256;
  PIXEL_T c = params->scalars[1] % 256;
  size_t h = params->height;
  size_t w = params->width;
#pragma omp parallel for firstprivate(w, h)
  for (size_t i = 0; i < h; i++) {
    for (size_t j = 0; j < w; j++) {
      size_t o = i * w + j;
      if (params->in[0][o] >= b && params->in[0][o] <= c)
        params->out[o] = PIXEL_MAX;
      else
        params->out[o] = PIXEL_ZER;
    }
  }
}
```

GCC/Clang : 9 µs

Environnement d'exécution

Fonctionnalités

- un des cluster I/O exécute le code de l'application
- les clusters de calcul exécutent nos noyaux OpenMP
- l'hôte lit et écrit les images (via SMIL (via smilc))
- les images sont stockées complètes dans la mémoire globale, puis tuilées pour tenir dans les SMEMs



Backend de FREIA

Une API « simple » à connecter/générer

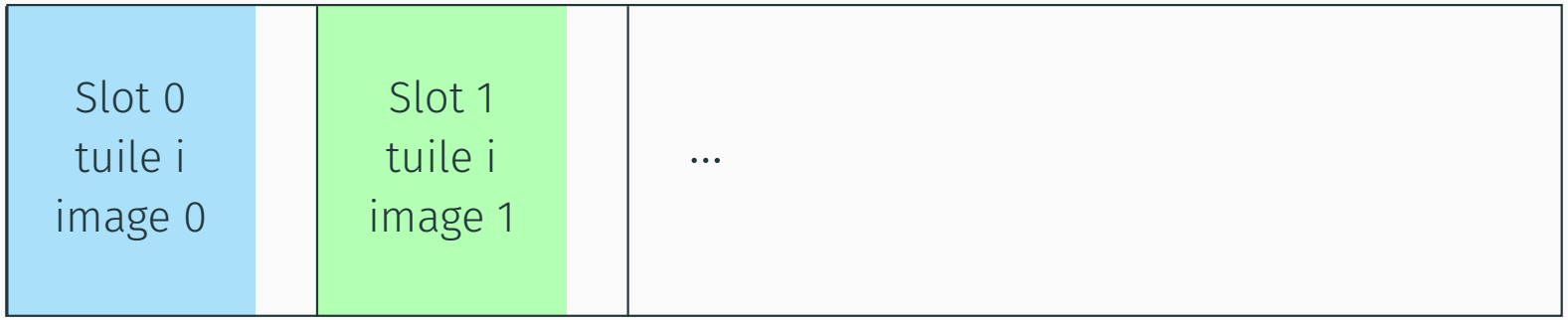
- commande
  - tableau d'instructions de taille fixe
  - remplie par l'utilisateur
  - envoyée telle quelle aux clusters de calcul
  - exécutée d'un trait
- instruction
  - transfert de tuiles DDR $\longleftrightarrow$ SMEM
  - exécution de tel noyau de calcul
- appel opérateur FREIA
  1. tuilage de l'image en DDR
  2. transferts DDR $\longrightarrow$ SMEM
  3. appel noyau calcul correspondant
  4. transferts SMEM $\longrightarrow$ DDR

Exemple de commande : seuillage binaire

```
int main_mppa_helper_0_0(freia_data2d *imout, freia_data2d *imin) {
  mppa_cc_instr_t *instrs;
  unsigned int i = 0;
  mppa_cc_cmd_t cmd0;
  instrs = cmd0.instrs;

  instrs[i].kind = MPPA_CMD_GET_IO_TILE;
  instrs[i].com.io_pos = ((io_image_h *)imin->mppa_ptr)->pos;
  instrs[i].com.cc_pos = 0;
  i++;

  instrs[i].kind = MPPA_CMD_EXECUTE_KERNEL;
  instrs[i].opr.kernel = MPPA_KERNEL_THRESHOLD_BIN;
  instrs[i].opr.pos[1] = 0; /* input */
  instrs[i].opr.pos[0] = 1; /* output */
  i++;

  instrs[i].kind = MPPA_CMD_PUT_IO_TILE;
  instrs[i].com.cc_pos = 1;
  instrs[i].com.io_pos = ((io_image_h *)imout->mppa_ptr)->pos;
  i++;

  mppa_compute(i, &cmd0); /* launch computation... */
  return 0;
}
```

Gestion de la mémoire : pattern *arena*

Cluster I/O : DDR



Clusters de calcul : SMEM

Les APIs de communication de Kalray

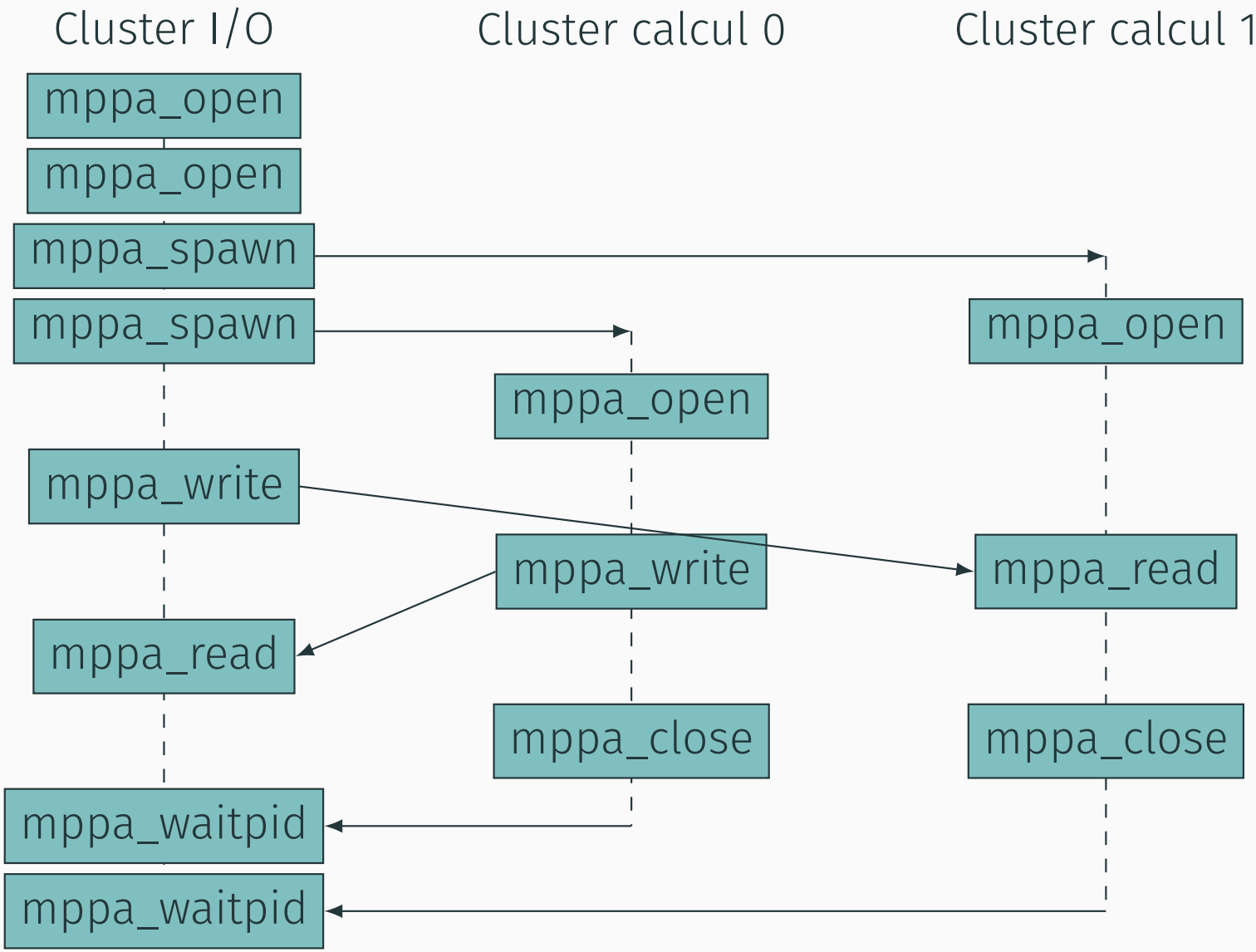
`libmppaipc`

- PCIe, NoC $\longrightarrow$ 1 seule bibliothèque
- des structures « haut niveau » : Portal, Sync
- communications synchrones et asynchrones
- facile à comprendre mais lourdeurs, perfs bof et bugs ☹

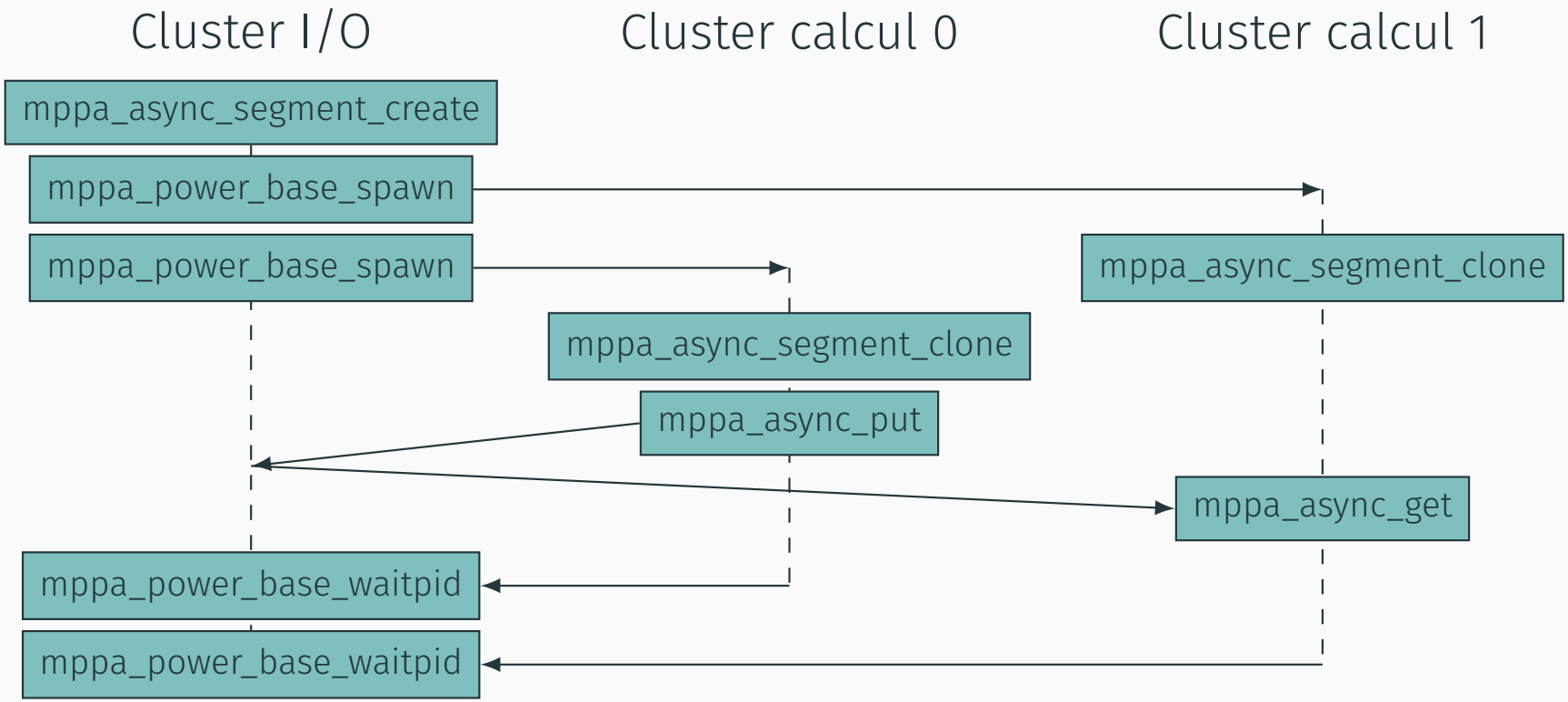
`libmppa_async`

- plusieurs petites bibliothèques de plus bas niveau
- démon sur le cluster d'E/S clusters de calcul $\longleftrightarrow$ DDR
- RDMA asynchrone sur des *memory segments*
- démarrage difficile mais bien pensée et performante ☺

Les APIs de Kalray : `libmppaipc`



Les APIs de Kalray : `libmppa_async`



Tuilage et recouvrements

Un tuilage simplifié

- une image $\longrightarrow$ 1, 4 ou 16 tuiles
- dépend de la capacité des slots SMEM
- on minimise les communications

Recouvrement

- 2 pixels par dimension
- sauf aux bords de l'image
- échange des bords entre clusters de calcul voisins
  - après chaque opérateur au voisinage
  - permet de garder les tuiles en SMEM

Perspectives

Mixer calculs et communications

- calculer les bords de tuile d'abord pour initier les transferts
- coupler plus fortement les noyaux de calcul à l'environnement d'exécution

Ajouter du parallélisme de tâches

- petites images (1 ou 4 tuiles) $\longrightarrow$ clusters inutilisés

Simuler l'environnement d'exécution

- ré-implémenter `libmppa_async` sur CPU
- OpenMP ? processus Unix ? MPI ?

Générer automatiquement des lots d'instructions

Intégré dans PIPS

- IR de FREIA dans PIPS
- expérience pour Sigma-C
- but : éviter des aller-retours DDR $\longrightarrow$ SMEM $\longrightarrow$ DDR
- allocation de slots SMEM ($\sim$registres) *à la main*
  - nombre de slots mémoire fixé par l'utilisateur

```
/**
 * @brief Get first unused SMEM slot
 */
static _int get_free_slot() {
  unsigned int max_smem_slots = get_int_property("HWAC_MPPA_MAX_SMEM_SLOTS");
  _int slot = SMEM_SLOT_UNDEFINED;
  for (unsigned int i = 0; i < max_smem_slots; i++) {
    if (smem_slot_users[i] == NULL) {
      slot = i;
      break;
    }
  }
  pips_assert("enough SMEM slots", slot != SMEM_SLOT_UNDEFINED);
  return slot;
}
```

Du scotch pour tenir le tout

Logiciels utilisés

Nom	Langage	Système de build
SMIL	C++	CMake
smiltofreia	Python/C	setuptools/Makefile
PIPS	C	Bash/Makefile
├── newgen	C	Makefile
└── linear	C	Makefile
FREIA	C	Bash/Makefile
└── smilc	C/C++	Meson
Applications	Python/C	Bash

Faire tenir le tout

Un système de build plus efficace

- convertir PIPS et ses dépendances à Meson
- convertir FREIA à Meson
- convertir SMIL à Meson (toujours en cours…)

Un Makefile pour les contrôler tous

- build/installe les différents composants
- des variables d'environnement pour savoir où chercher
- un script Bash pour compiler : `freiacc`

Démo

Application : pré-traitement détection voies de circulation

- écrite en SMIL/Python
- 1300 frames (1 min)
- 46 noyaux de calcul (36 opérateurs au voisinage)
  - PIPS $\longrightarrow$ MPPA : 1 commande, 49 instructions

Démonstration : screencast

- chaîne de compilation
- aperçu du résultat

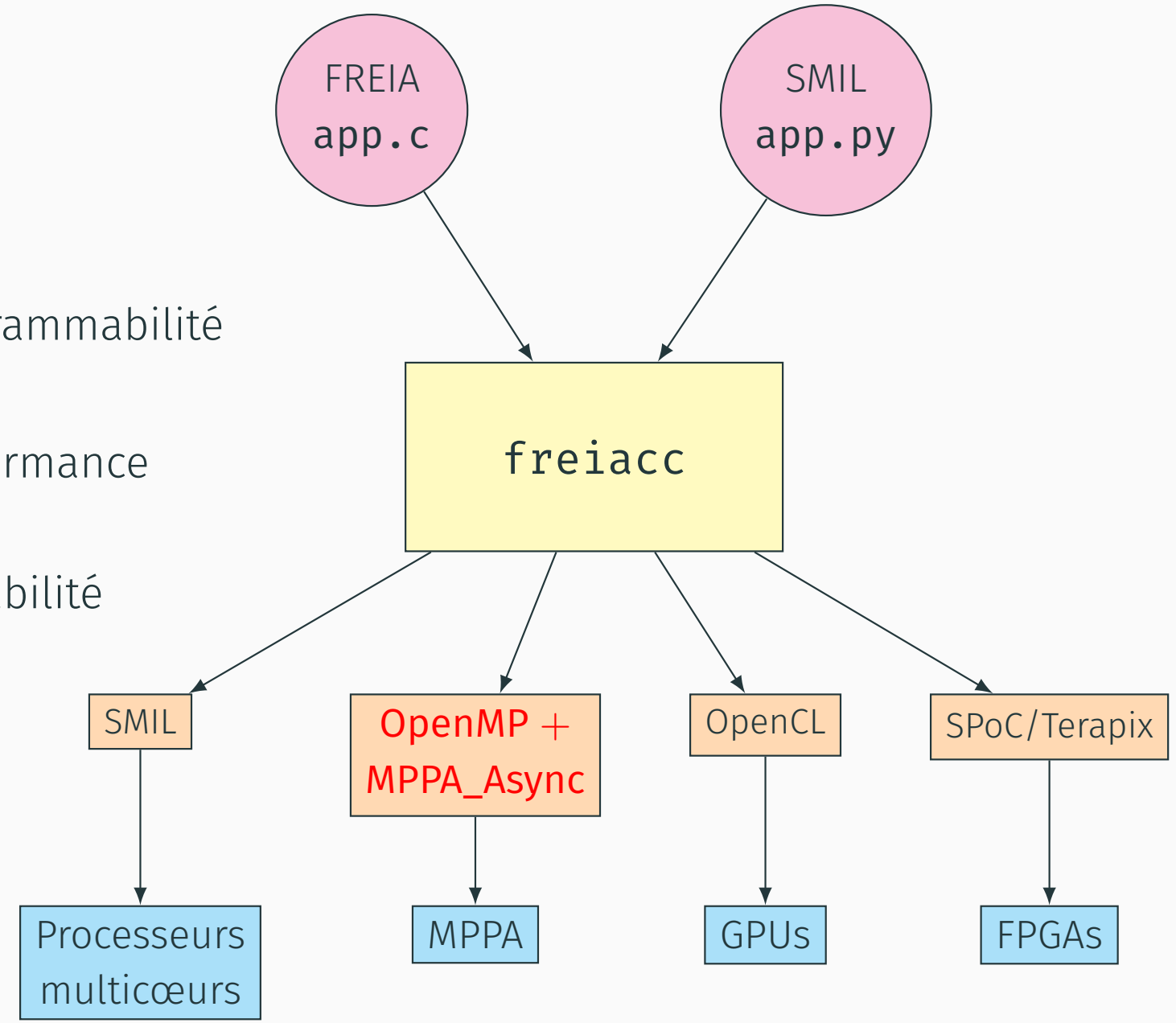
Des chiffres

Backend	Matériel	Temps d'exécution (s)
SMIL/Python	i7–3820 (4 threads)	54
FREIA/SMIL	id.	54
FREIA/SMIL + PIPS	id.	54
FREIA/MPPA	MPPA (4 clusters)	74
FREIA/MPPA + PIPS	id.	35
FREIA/MPPA	MPPA (16 clusters)	59
FREIA/MPPA + PIPS	id.	26
FREIA/OpenCL	id.	1570
FREIA/OpenCL + PIPS	id.	194

Conclusion

Conclusion : les 3P

- ✓ Programmabilité
- ✓ Performance
- ✓ Portabilité



Questions?

Bibliographie i


Pierre Guillou. « Efficient Compilation of Image Processing Applications for Manycore Processors ». Theses. PSL Research University, nov. 2016. URL : https://pastel.archives-ouvertes.fr/tel-01531196.


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 ii

 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.