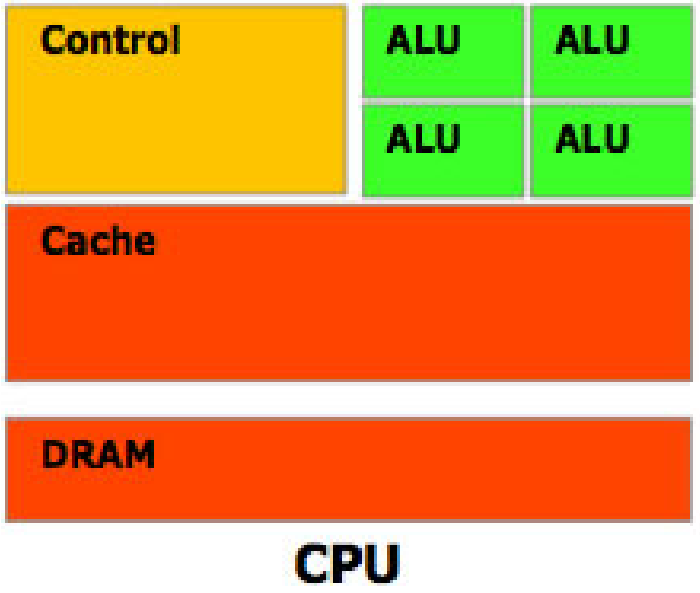
Compilation et optimisation statique des communications hôte-accélérateur

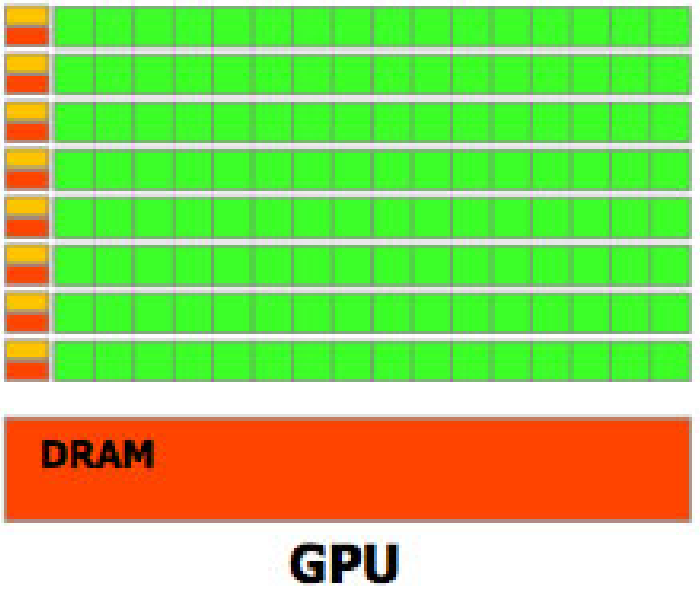
Mehdi Amini, Fabien Coelho, François Irigoin, Ronan Keryell

Renpar, 11 mai 2011

YAPAG

Yet Another Paper About GPU



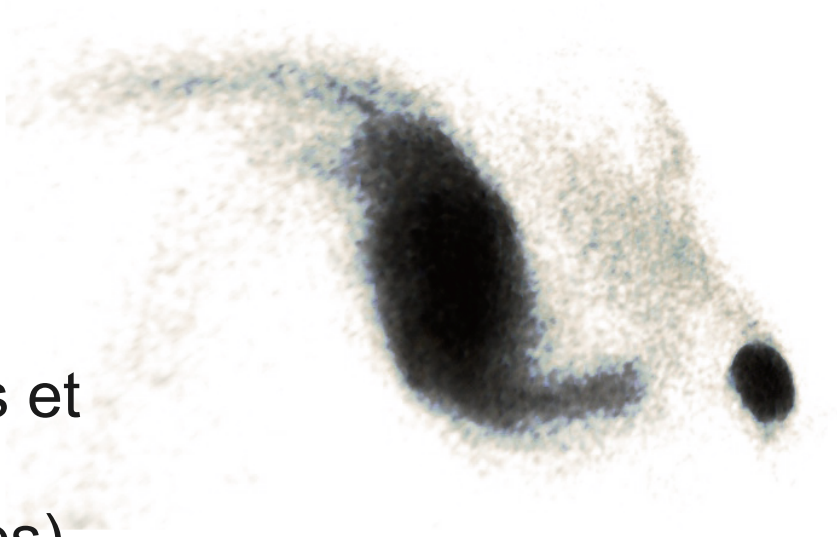



```
// Sequential execution
for ( int i =0; i <n; i ++)
  a[ i ] = b[ i ] + c [ i ] ;
```

```
// Each "thread" get one of
// the possible value of i
i = getLocalId ( ) ;
a[ i ] = b[ i ] + c [ i ] ;
```

Résultats

Exemple : simulation cosmologique



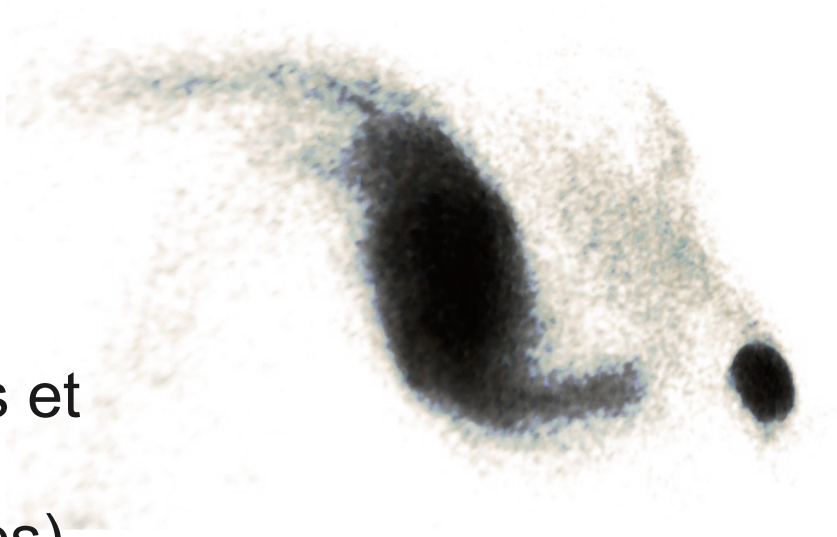
- 800 lignes de code,
- calculs dominés par la bande passante mémoire,
- principalement des nids de boucles et un appel à FFTW3
- simulations longues (jours/semaines)

	Simulation cosmologique		Jacobi	
Séquentiel		98.4s		48.6s
OpenMP	x5.9	16.6s	x3.5	13.8s
Cuda de base	x3.8	25.9s	x0.5	95.6s
Cuda comm. opt.	x46.9	2.1s	x12.8	3.8s
Cuda manuel	x54.7	1.8s		N/A

bi-Xeon Nehalem X5670 soit 12 cœurs à 2,93 GHz Nvidia Tesla C2060

Résultats

Exemple : simulation cosmologique



- 800 lignes de code,
- calculs dominés par la bande passante mémoire,
- principalement des nids de boucles et un appel à FFTW3
- simulations longues (jours/semaines)

	Simulation cosmologique		Jacobi	
Séquentiel		98.4s		48.6s
OpenMP	x5.9	16.6s	x3.5	13.8s
Cuda de base	**x3.8**	**25.9s**	**x0.5**	**95.6s**
Cuda comm. opt.	**x46.9**	**2.1s**	**x12.8**	**3.8s**
Cuda manuel	x54.7	1.8s		N/A

bi-Xeon Nehalem X5670 soit 12 cœurs à 2,93 GHz Nvidia Tesla C2060

Hypothèses

Par simplicité, nous travaillons selon les hypothèses suivantes :

- Les tableaux sont déclarés statiquement (C99)
- La mémoire de l'accélérateur est suffisante pour contenir les tableaux nécessaires aux kernels

Parallélisation automatique

Basée sur les nids de boucle

```
void step1(n,m,
           a[n][m] ,
           b[n][m] ) {
  / / A parallel loop nest
  for ( int i = 0; i <n; i++)
    for ( int j = 0; j <m; j++)
      //  f( ) is known
      a[i][j] = f( b[i][j], i, j ) ;
    }
  }
}

int f( int x, int y, int z ) {
...
}
```

```
void kernel1 (n,m,
              a[n][m] ,
              b[n][m] ) {
  int i = getId (DIM0) ;
  int j = getId (DIM1) ;
  // f() is known
  a[i][j] = f(b[i][j]) ;
}
void step1(n,m,
           a[n][m],
           b[n][m] ) {
  ... d_b[n][m]
  cuda_malloc(d_b,n,m) ;
  COPY_IN(n,m,b, d_b) ;
  // This replace the loop nest
  call_kernel( kernel1, n, m, a, d_b) ;
  COPY_OUT(n,m,a) ;
  cuda_free(d_b)
}
```

Parallélisation automatique

Peut être sioux...

```
void histogram(int n,
               int a[n],
               int b[n],
               int c[n] ) {

  / / A parallel loop nest ??
  for ( int i = 0; i <n; i++)
    a[ b[ i ] ] += c[ i ] ;
}
```

```
void kernel1 (int n
              a[n],
              b[n] ) {
  int i = getId (DIM0) ;
  AtomicInc( &a[ b[ i ] ], c[ i ] ) ;
}
```

Cibles

Le schéma global d'une simulation numérique se présente souvent ainsi :

```
int main( int argc , char *argv [] ) {
  /* Read data from a file */
  init (argv[1] ) ;

  /* Main temporal loop */
  for ( t = 0; t < T; t+=DT) {
    iteration (n, m, a, b, c, d) ;
    if ( ... ) { // sometimes
      display (a) ;
      checkpoint (a) ;
    }
  }

  /* Output the results to a file */
  dump(argv [2] ) ;
  return 0;
}
```

```
void iteration ( ... ) {

  // Produce b with a
  step1(b, a) ;

  // Produce c with d
  step2(c , d) ;

  // Produce a with b and c
  step3(a,b, c) ;

}
```

Cibles

Le schéma global d'une simulation numérique se présente souvent ainsi :

```
int main( int argc , char *argv [] ) {
  /* Read data from a file */
  init (argv[1] ) ;

  /* Main temporal loop */
  for ( t = 0; t < T; t+=DT) {
    iteration (n, m, a, b, c, d) ;
    if ( ... ) { // sometimes
      display (a) ;
      checkpoint (a) ;
    }
  }

  /* Output the results to a file */
  dump(argv [2] ) ;
  return 0;
}
```

```
void iteration ( ... ) {

  // Produce b with a
  step1(b, a) ;

  // Produce c with d
  step2(c , d) ;

  // Produce a with b and c
  step3(a,b, c) ;

}
```

Schéma naïf

```
/* Main temporal loop */
for ( t = 0; t < T; t+=DT) {
  iteration (n, m, a, b, c, d) ;
  if ( … ) { / / sometimes

    display (a) ;
    checkpoint (a) ;
  }
}
```

```
void iteration ( … ) {
  // Produce b with a
  step1(b, a) ;
  // Produce c with d
  step2(c , d) ;
  // Produce a with b and c
  step3(a,b, c) ;
}
```

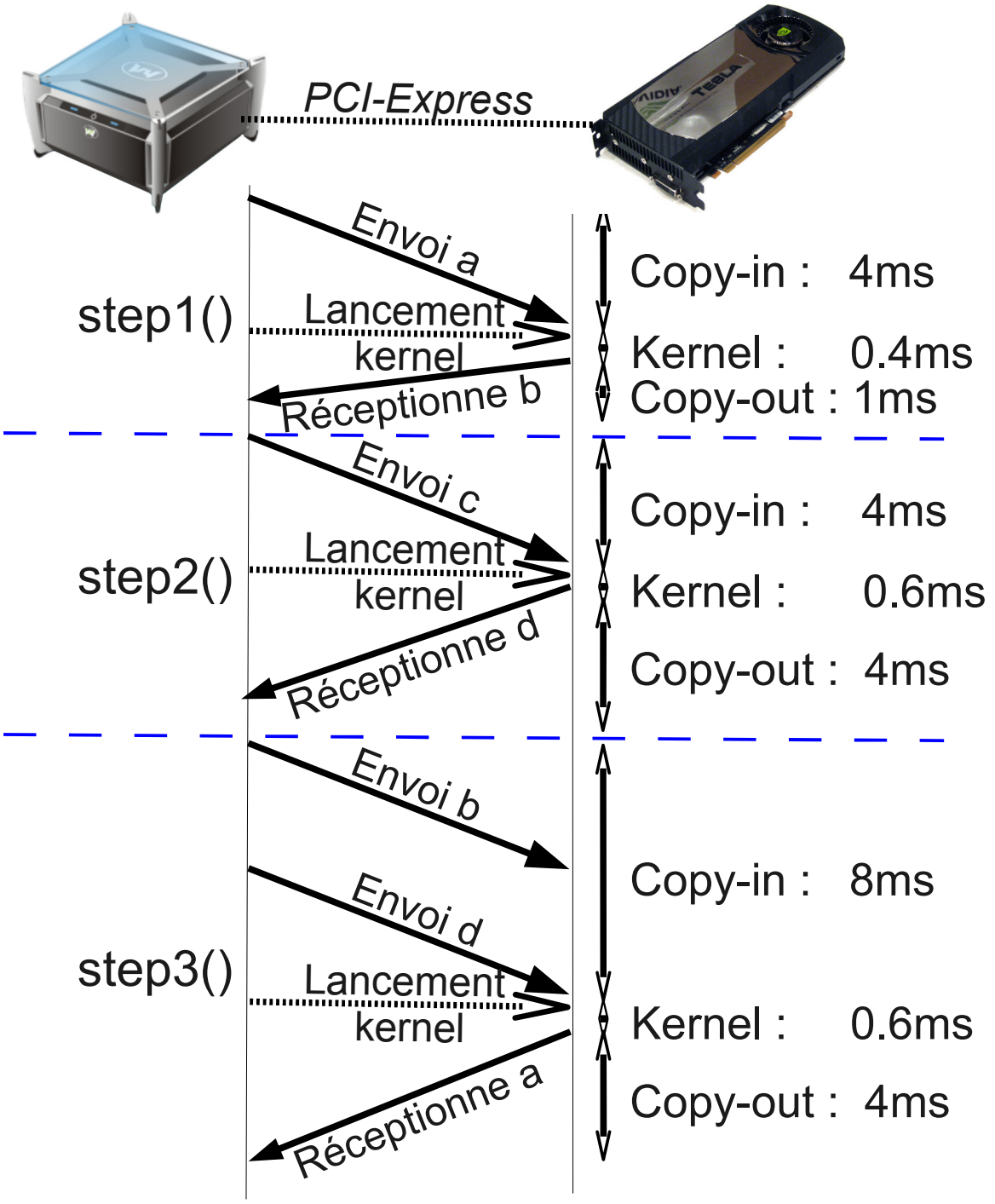


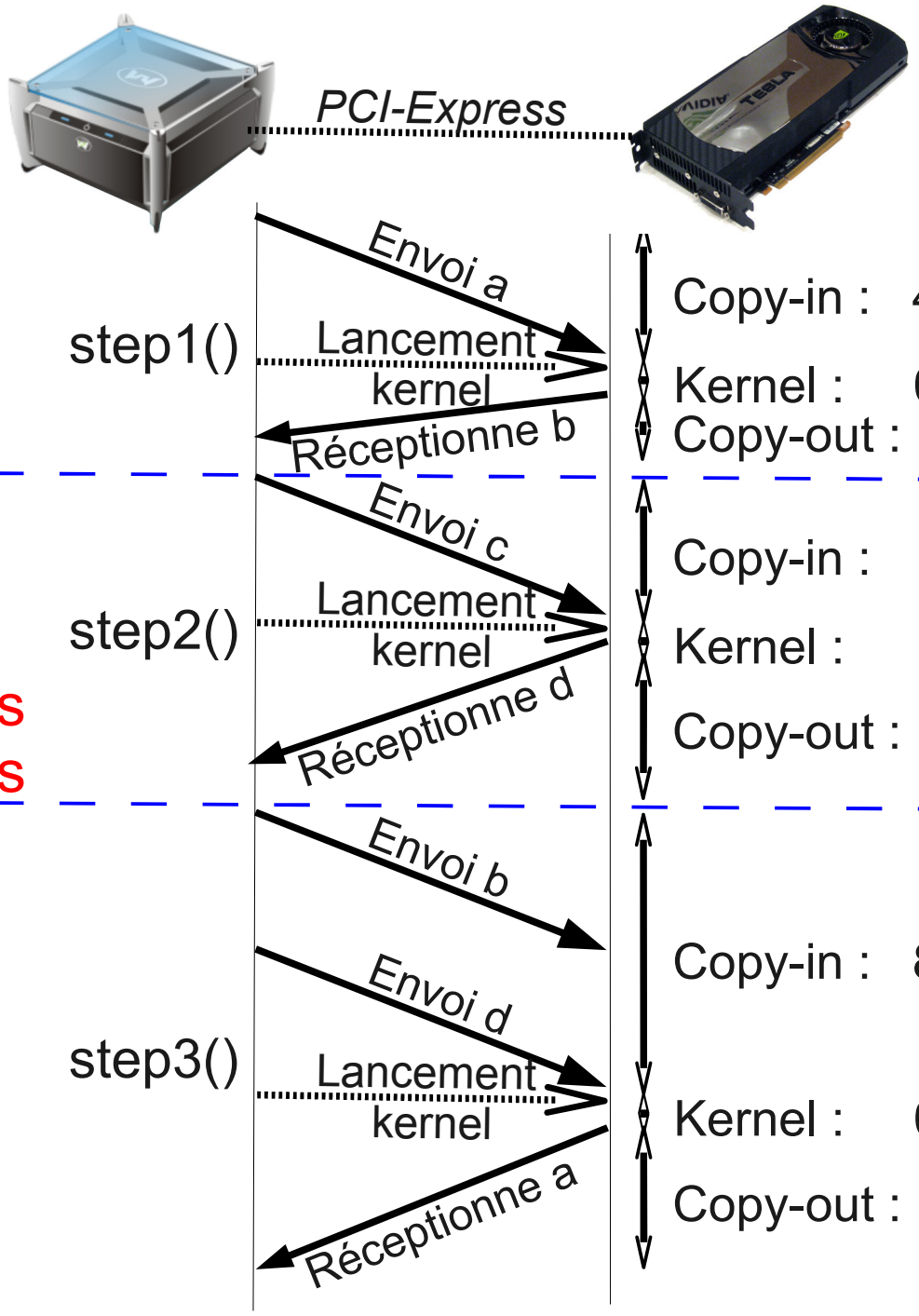
Schéma naïf

```
/* Main temporal loop */
for ( t = 0; t < T; t+=DT) {
  iteration (n, m, a, b, c, d) ;
  if ( … ) { / / sometimes

    display (a) ;
    checkpoint (a) ;
  }
}
```

Communications : 25ms
Kernel : 1.6ms

```
void iteration ( … ) {
  // Produce b with a
  step1(b, a) ;
  // Produce c with d
  step2(c , d) ;
  // Produce a with b and c
  step3(a,b, c) ;
}
```

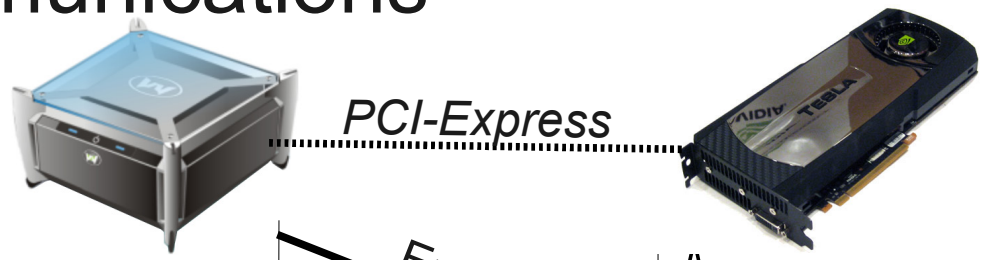


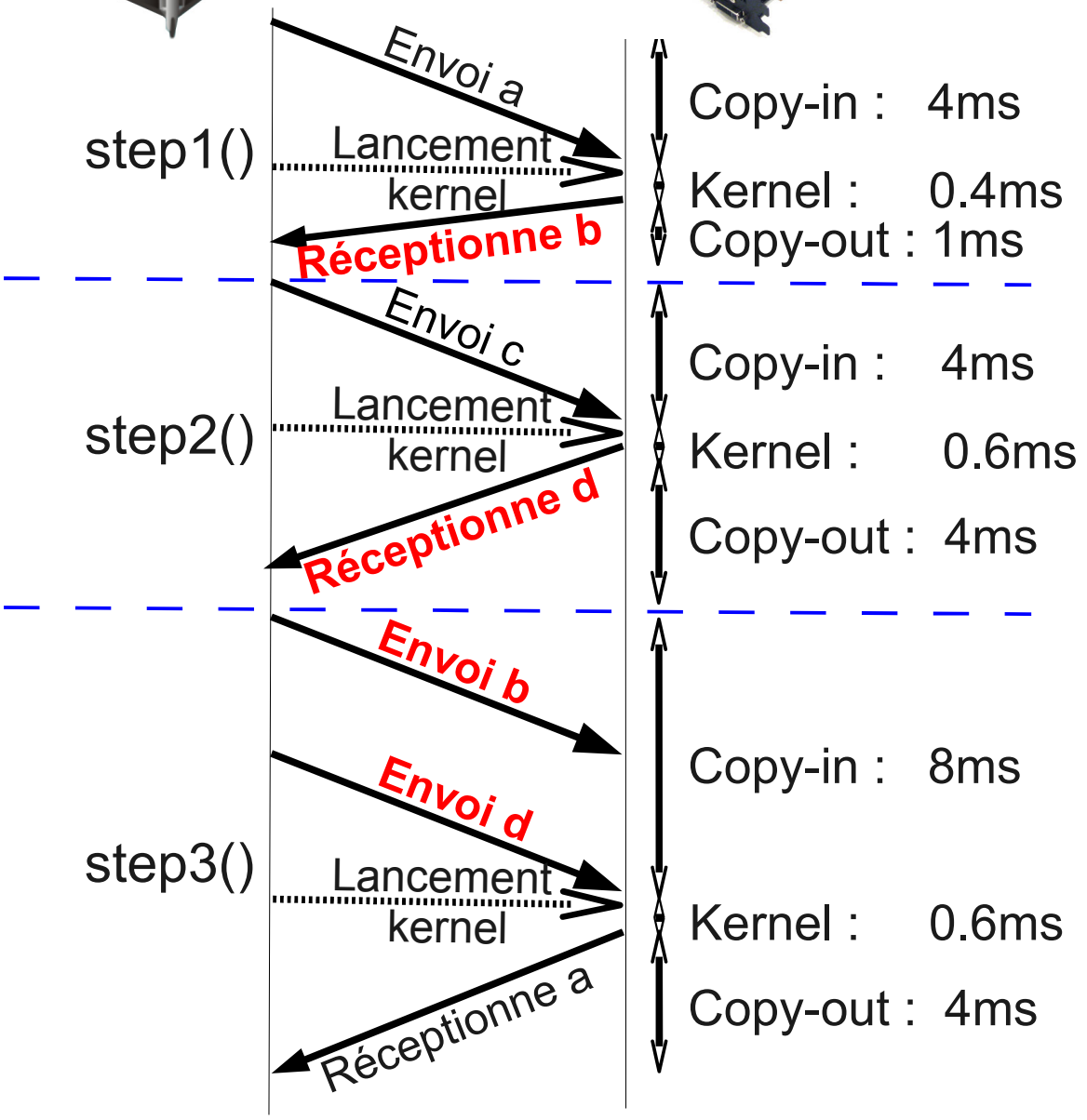
Suppression des communications redondantes

```
/* Main temporal loop */
for ( t = 0; t < T; t+=DT) {
  iteration (n, m, a, b, c, d) ;
  if ( … ) { / / sometimes

    display (a) ;
    checkpoint (a) ;
  }
}
```

```
void iteration ( … ) {
  // Produce b with a
  step1(b, a) ;
  // Produce c with d
  step2(c , d) ;
  // Produce a with b and c
  step3(a,b, c) ;
}
```



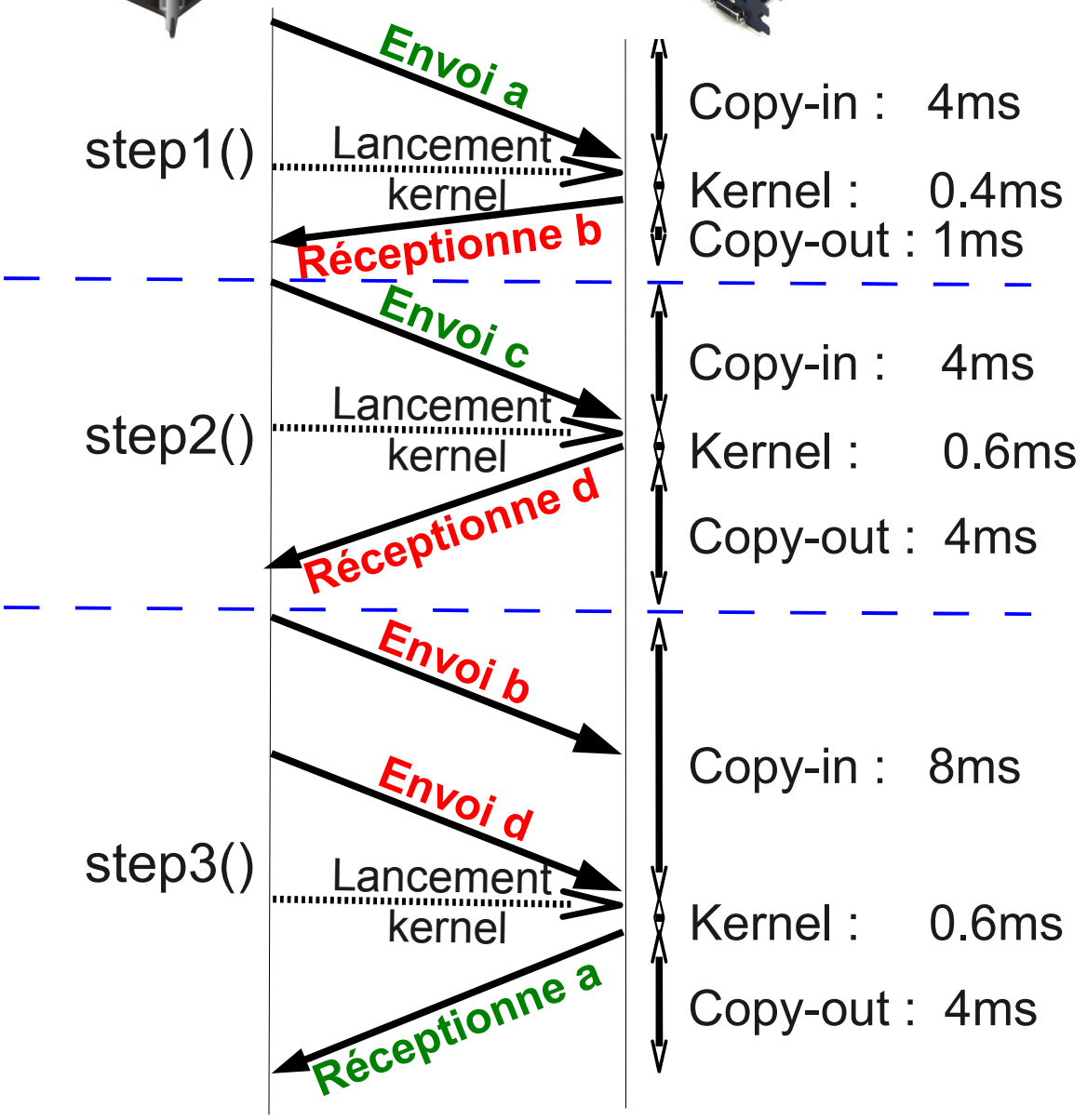


Déplacement des communications hors des boucles

```
/* Main temporal loop */
for ( t = 0; t < T; t+=DT) {
  iteration (n, m, a, b, c, d) ;
  if ( … ) { / / sometimes

    display (a) ;
    checkpoint (a) ;
  }
}
```

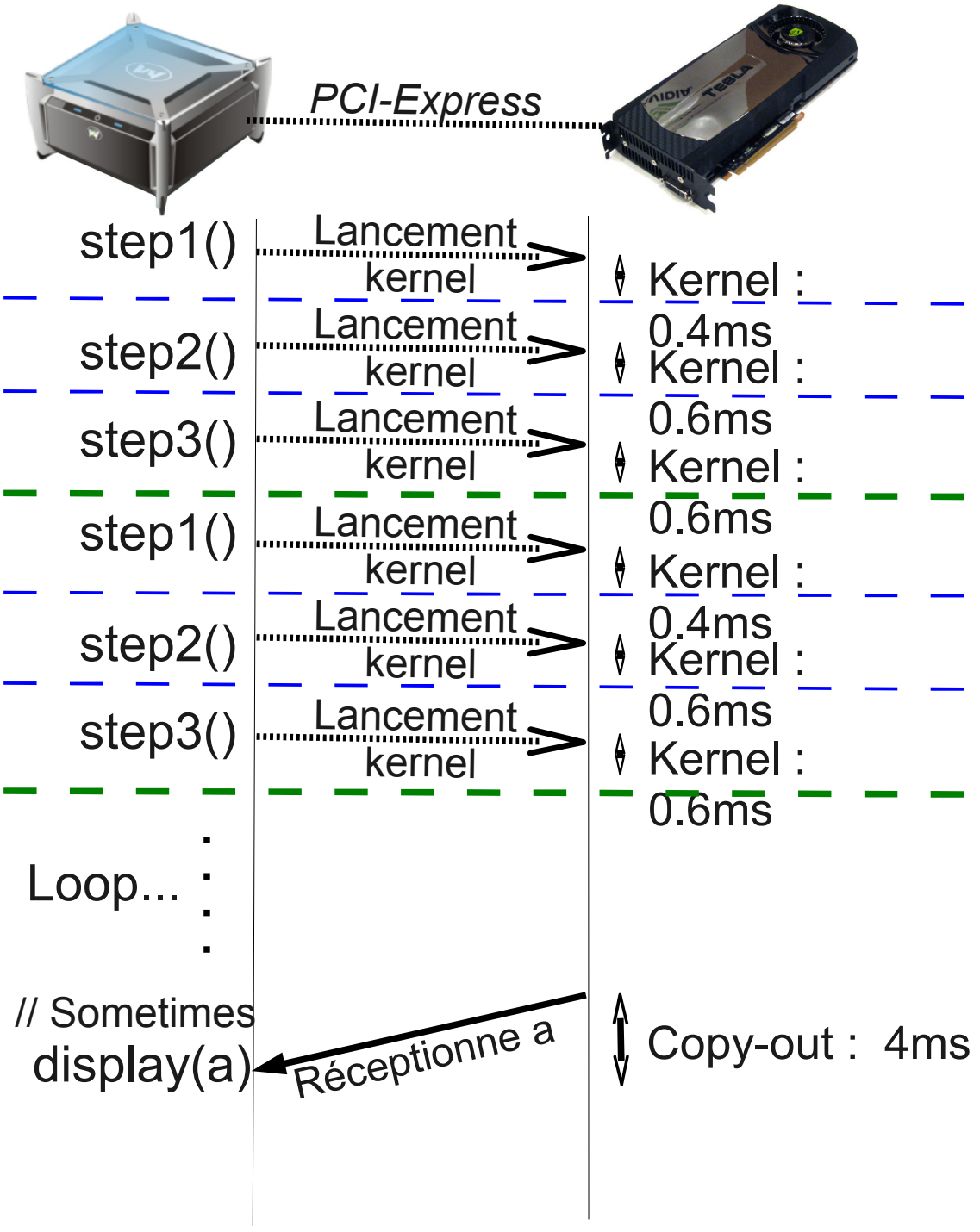
```
void iteration ( … ) {
  // Produce b with a
  step1(b, a) ;
  // Produce c with d
  step2(c , d) ;
  // Produce a with b and c
  step3(a,b, c) ;
}
```



Après optimisation

```
COPY_IN(a) ; COPY_IN(d) ;
/* Main temporal loop */
for ( t = 0; t < T; t+=DT) {
  iteration (n, m, a, b, c, d) ;
  if ( … ) { / / sometimes
    COPY_OUT(a)
    display (a) ;
    checkpoint (a) ;
  }
}
COPY_OUT(a)

void iteration ( … ) {
  // Produce b with a
  step1(b, a) ;
  // Produce c with d
  step2(c , d) ;
  // Produce a with b and c
  step3(a,b, c) ;
}
```



Contexte

- Codes générés par des outils de haut niveau
- Codes générés depuis du code Scilab
- Nouveaux développements, sans les moyens de spécialiser pour chaque plateforme existantes

Notre solution

Parallélisation automatisée source-à-source

- Interaction avec le reste du monde (le source comme *RI*)
- Écrire une fois, exécuter partout (multi-cibles)
- Coût d'entrée, coût de sortie
- Deboguage / maintenance
- Transformations qui s'expriment bien au niveau source
- Permet de se concentrer sur les transformations de parallélisation et délègue le reste au compilateur de la cible (exemple : openmp+mpi+cuda)

Génération optimisée, 1 : détection des kernels

```
/* Main temporal loop */
for ( t = 0; t < T; t+=DT) {
  iteration (n, m, a, b, c, d) ;
  if ( … ) { // sometimes

    display (a) ;
    checkpoint (a) ;
  }
}
```

Dans le contexte de la parallélisation automatique, le compilateur identifie les kernels lorsque le compilateur les génère.

```
void iteration ( … ) {
  // Produce b with a
  step1(b, a) ;
  // Produce c with d
  step2(c , d) ;
  // Produce a with b and c
  step3(a,b, c) ;
}
```

```
void step1(n,m,
           a[n][m],b[n][m]) {
 call_kernel(kernel1,n,m,a,b);
}
```

Génération optimisée, 2 : Ensemble $\mathcal{D}_A^<$

On associe à chaque instruction les tableaux dont la dernière définition a été faite sur l'accélérateur et qui n'ont pas été utilisés par l'hôte entre temps.

$$\mathcal{D}_A^<(I) = \left(\bigcap_{I' \in prec(I)} \mathcal{D}_A^<(I') \right) - \mathcal{R}(I) - \mathcal{W}(I)$$

$$\mathcal{D}_A^<(I_k) = \mathcal{OUT}(I_k) \cup \bigcap_{I' \in prec(I_k)} \mathcal{D}_A^<(I')$$

```
void iteration ( … ) {
  // Produce b with a
  step1(b, a) ;
  // Produce c with d
  step2(c , d) ;
  // Produce a with b and c
  step3(a,b, c) ;
}
```

Ajoute b, $\mathcal{D}_A^<$ **= { b }**

Ajoute c, $\mathcal{D}_A^<$ **= { b, c }**

Ajoute a, $\mathcal{D}_A^<$ **= { b, c, a }**

$\mathcal{D}_A^<$ **= { b, c, a }**

Génération optimisée, 3 : Ensemble $\mathcal{U}_A^{>}$

On associe à chaque instruction les tableaux dont il est certain que la prochaine utilisation se fera sur l'accélérateur ;

$$\mathcal{U}_A^{>}(I) = \left(\bigcup_{I' \in succ(I)} \mathcal{U}_A^{>}(I') \right) - \mathcal{W}(I)$$

$$\mathcal{U}_A^{>}(I_k) = \mathcal{IN}(I_k) \cup \mathcal{W}(I_k) \cup \bigcup_{I' \in succ(I_k)} \mathcal{U}_A^{>}(I')$$

$\mathcal{U}_A^{>}$ **= { a, d }**

```
void iteration ( … ) {
  // Produce b with a
  step1(b, a) ;
  // Produce c with d
  step2(c , d) ;
  // Produce a with b and c
  step3(a,b, c) ;
}
```

$\mathcal{D}_A^{<}$ **= { b, c, a }**

Ajoute a, supprime b $\mathcal{U}_A^{>}$ **= { a, d }**

Ajoute d, supprime c $\mathcal{U}_A^{>}$ **= { b, d }**

Ajoute b et c, $\mathcal{U}_A^{>}$ **= { b, c }**

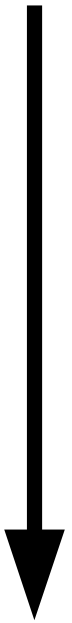
Génération optimisée, 4 : interprocédural

$$\mathcal{D}_A^{<}(I) = \left(\bigcap_{I' \in prec(I)} \mathcal{D}_A^{<}(I') \right) - \mathcal{R}(I) - \mathcal{W}(I)$$

$$\mathcal{D}_A^{<}(I_k) = \mathcal{OUT}(I_k) \cup \bigcap_{I' \in prec(I_k)} \mathcal{D}_A^{<}(I')$$

$$\mathcal{D}_A^{<}(c) = \left(trans_{f->c}(\overline{\mathcal{D}_A^{<}}(f)) \cup \bigcap_{I' \in prec(c)} \mathcal{D}_A^{<}(I') \right) - \mathcal{R}(c) - \mathcal{W}(c)$$

```
/* Main temporal loop */
for ( t = 0; t < T; t+=DT) {
  iteration (n, m, a, b, c, d) ;
  if ( … ) {
  // sometimes
    display (a) ;
    checkpoint (a) ;
  }
}
```



Ajoute b,c,a — $\mathcal{D}_A^{<}$ **= { b, c, a }**

Supprime a — $\mathcal{D}_A^{<}$ **= { b, c }**

Intersection — $\mathcal{D}_A^{<}$ **= { b, c }**

Génération optimisée, 5 : point fixe

$$\mathcal{D}_A^<(I) = \left(\bigcap_{I' \in prec(I)} \mathcal{D}_A^<(I') \right) - \mathcal{R}(I) - \mathcal{W}(I)$$

$$\mathcal{D}_A^<(I_k) = \mathcal{OUT}(I_k) \cup \bigcap_{I' \in prec(I_k)} \mathcal{D}_A^<(I')$$

$$\mathcal{D}_A^<(c) = \left(trans_{f->c}(\overline{\mathcal{D}_A^<}(f)) \cup \bigcap_{I' \in prec(c)} \mathcal{D}_A^<(I') \right) - \mathcal{R}(c) - \mathcal{W}(c)$$

```
/* Main temporal loop */
for ( t = 0; t < T; t+=DT) {
  iteration (n, m, a, b, c, d) ;
  if ( … ) {
  // sometimes
    display (a) ;
    checkpoint (a) ;
  }
}
```

Ajoute b,c,a

Supprime a

Intersection

Intersecte et recommence si changement

$\mathcal{D}_A^<$ = { b, c }

$\mathcal{D}_A^<$ = { b, c, a }

$\mathcal{D}_A^<$ = { b, c }

$\mathcal{D}_A^<$ = { b, c }

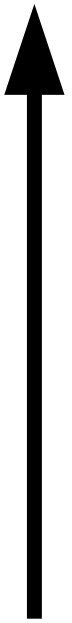
Génération optimisée, 4 : interprocédural

$$\mathcal{U}_A^{>}(I) = \left(\bigcup_{I' \in succ(I)} \mathcal{U}_A^{>}(I') \right) - \mathcal{W}(I)$$

$$\mathcal{U}_A^{>}(I_k) = \mathcal{IN}(I_k) \cup \mathcal{W}(I_k) \cup \bigcup_{I' \in succ(I_k)} \mathcal{U}_A^{>}(I')$$

$$\mathcal{U}_A^{>}(c) = \left(trans_{f->c}(\overline{\mathcal{U}_A^{>}}(f)) \cup \bigcup_{I' \in succ(c)} \mathcal{U}_A^{>}(I') \right) - \mathcal{W}(c)$$

```
/* Main temporal loop */
for ( t = 0; t < T; t+=DT) {
  iteration (n, m, a, b, c, d) ;
  if ( … ) {
  // sometimes
    display (a) ;
    checkpoint (a) ;
  }
}
```



Ajoute a,b,c,d

$\mathcal{U}_A^{>}$ **= { a, d }**

$\mathcal{U}_A^{>}$ **= { }**

$\mathcal{U}_A^{>}$ **= { }**

$\mathcal{U}_A^{>}$ **= { }**

Génération optimisée, 5 : point fixe

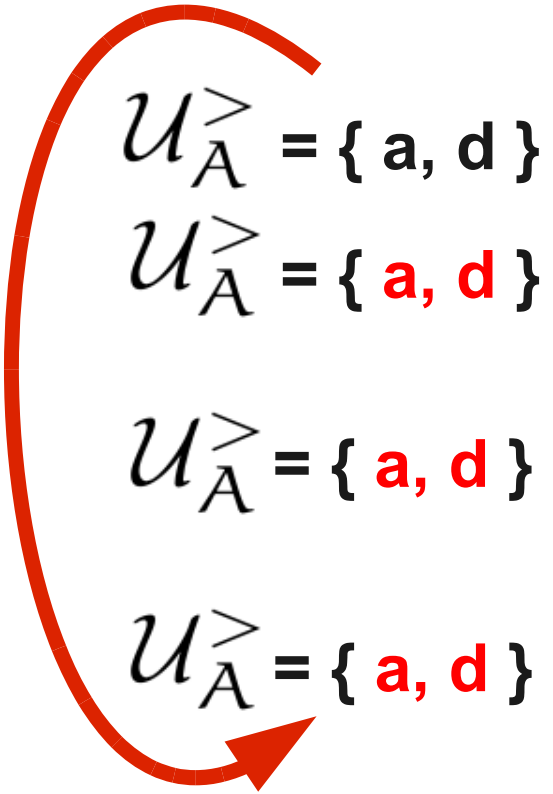
$$\mathcal{U}^{\geq}_{A}(I) = \left(\bigcup_{I' \in succ(I)} \mathcal{U}^{\geq}_{A}(I') \right) - \mathcal{W}(I)$$

$$\mathcal{U}^{\geq}_{A}(I_k) = \mathcal{IN}(I_k) \cup \mathcal{W}(I_k) \cup \bigcup_{I' \in succ(I_k)} \mathcal{U}^{\geq}_{A}(I')$$

$$\mathcal{U}^{\geq}_{A}(c) = \left(trans_{f->c}(\overline{\mathcal{U}^{\geq}_{A}}(f)) \cup \bigcup_{I' \in succ(c)} \mathcal{U}^{\geq}_{A}(I') \right) - \mathcal{W}(c)$$

```
/* Main temporal loop */
for ( t = 0; t < T; t+=DT) {
  iteration (n, m, a, b, c, d) ;
  if ( … ) {
  // sometimes
    display (a) ;
    checkpoint (a) ;
  }
}
```

Ajoute a et d

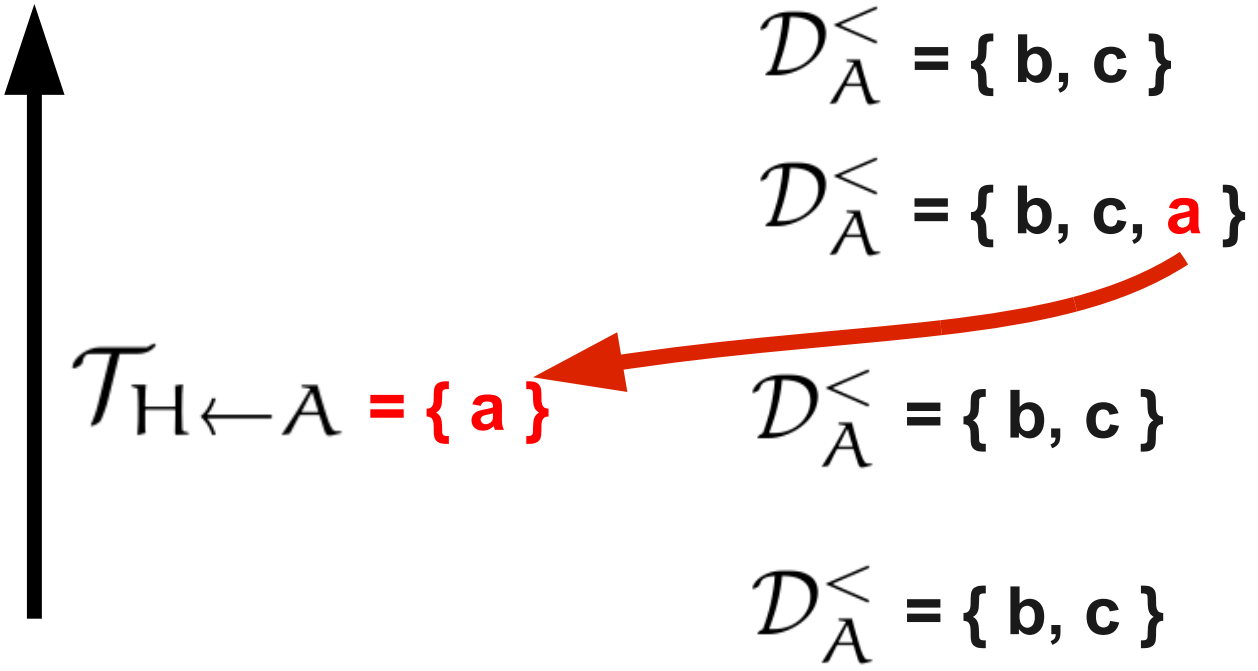


Génération optimisée, 6 : ensemble $\mathcal{T}_{H \leftarrow A}$

On associe à chaque instruction les tableaux à transférer vers l'accélérateur juste après exécution de l'instruction.

$$\mathcal{T}_{H \leftarrow A}(I) = \mathcal{D}_A^{<}(I) - \bigcap_{I' \in succ(I)} \mathcal{D}_A^{<}(I')$$

```
/* Main temporal loop */
for ( t = 0; t < T; t+=DT) {
  iteration (n, m, a, b, c, d) ;
  if ( … ) {
  // sometimes
    display (a) ;
    checkpoint (a) ;
  }
}
```



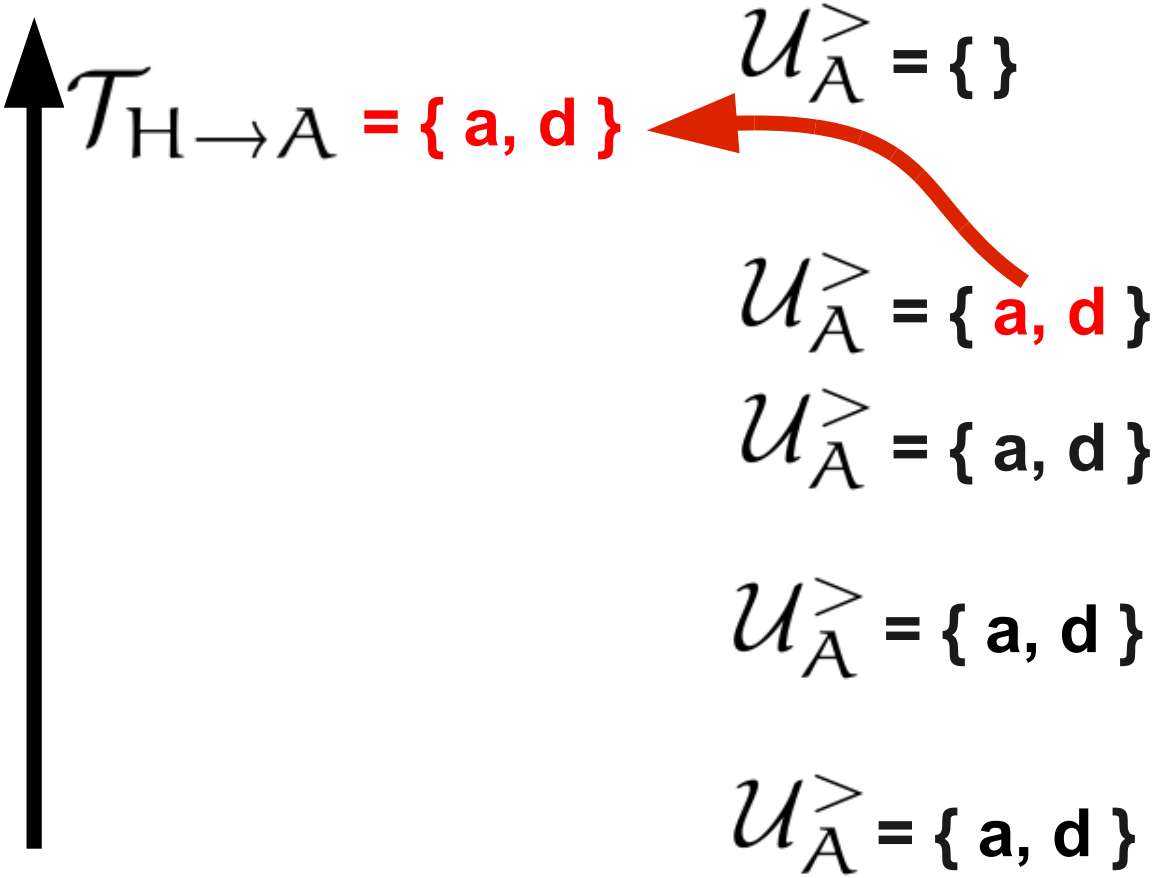
Génération optimisée, 7 : ensemble $\mathcal{T}_{H \to A}$

On associe à chaque instruction les tableaux à transférer depuis l'accélérateur juste avant exécution de l'instruction

$$\mathcal{T}_{H \to A}(I) = \mathcal{W}(I) \cap \bigcup_{I' \in succ(I)} \mathcal{U}_A^{\geq}(I')$$

```
init_data(a,d) ;


for ( t = 0; t < T; t+=DT) {
  iteration (n, m, a, b, c, d) ;
  if ( … ) {
  // sometimes
    display (a) ;
    checkpoint (a) ;
  }
}
```



Runtime léger

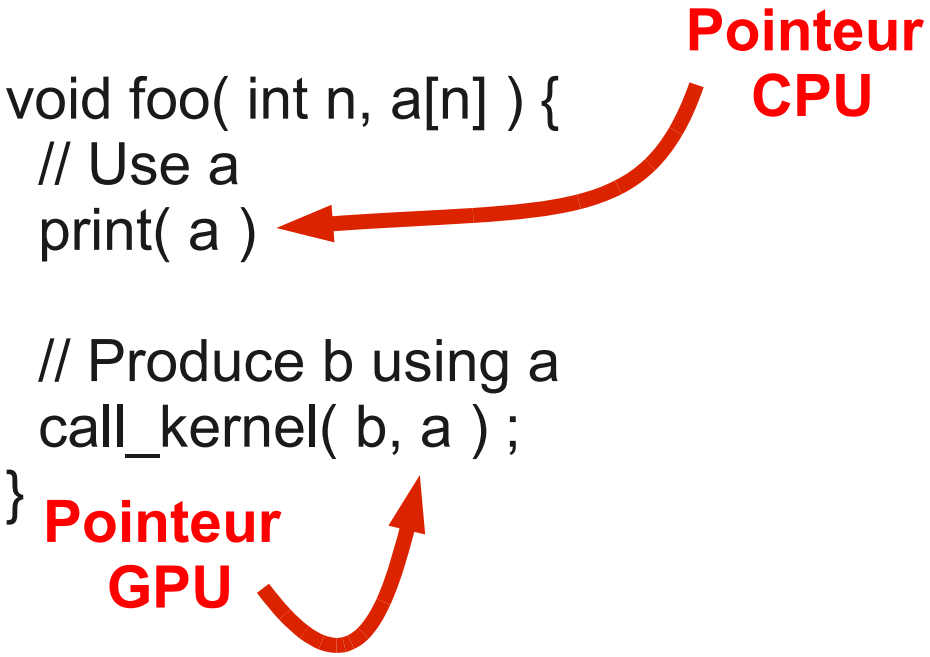
Comment avoir à la fois les adresses CPU et GPU ?

```
void foo( int n, a[n] ) {
  // Use a
  print( a )

  // Produce b using a
  call_kernel( b, a ) ;
}
```

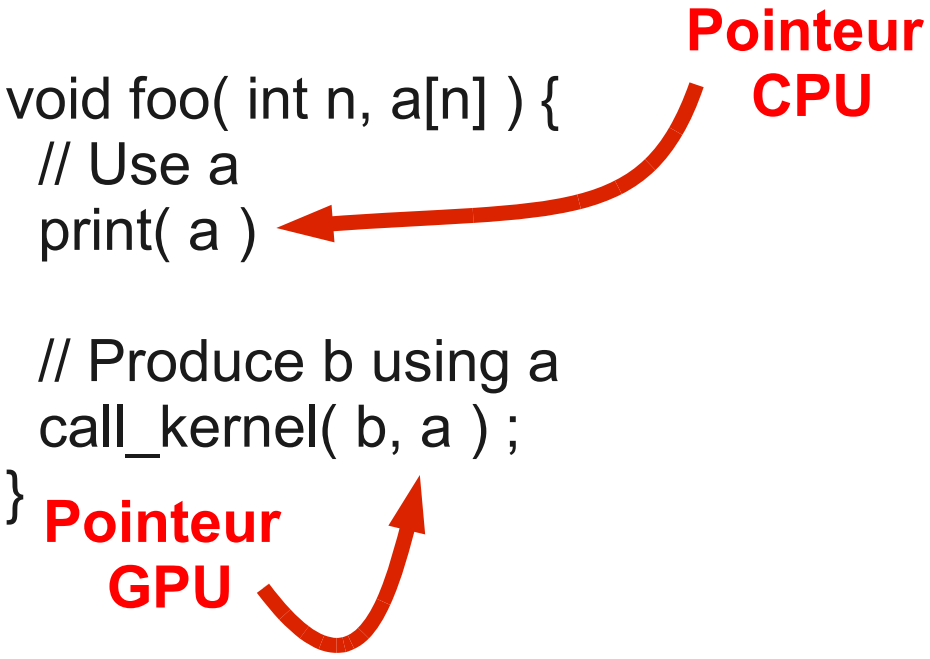
Runtime léger

Comment avoir à la fois les adresses CPU et GPU ?



Runtime léger

Comment avoir à la fois les adresses CPU et GPU ?



Runtime léger

Comment avoir à la fois les adresses CPU et GPU ?

```
void foo( int n, a[n] ) {
  // Use a
  print( a )

  // Produce b using a
  call_kernel( b, a ) ;
}
```

Pointeur CPU

Pointeur GPU

@CPU	@GPU
&a	…..
&b	…..
&c	…..
&d	…..
...	….
...	….

```
void foo( int n, a[n], d_a[n] )
{
  // Use a on CPU
  Print( a )

  // Produce b using a
  call_kernel( b, d_a ) ;
}
```

```
void foo( int n, a[n] ) {
  // Use a on CPU
  print( a )

  // Produce b using a
  call_kernel( b, get_dev(a) ) ;
}
```

Travaux préliminaires, plusieurs pistes :

- Évaluer le semi automatisé : PGI et HMPP
- Vers l'asynchrone et le parallélisme de tâche ? (intégrer Starpu?)
- Optimisation de communication et multi-GPU ?
- Collaboration CPU ↔ GPU
- Optimisation de communication et contrainte mémoire ?
- Plus de collaboration au runtime ?
- Plus de benchmarks ;-)

Bilan

- Le futur s'annonce hétérogène
- L'optimisation des communications est critique dans ce contexte ! (accélération x8 à x25 sur nos exemples)
- Approche statique (*compile time*)
- Entièrement automatisée

	Simulation cosmologique		Jacobi	
Séquentiel		98.4s		48.6s
OpenMP *(6 threads)*	x5.9	16.6s	x3.5	13.8s
Cuda de base	**x3.8**	**25.9s**	**x0.5**	**95.6s**
Cuda comm. opt.	**x46.9**	**2.1s**	**x12.8**	**3.8s**
Cuda manuel	x54.7	1.8s		N/A