

Transformation de code et parallélisation pour la génération de code de transfert de données dans Ardoise

Ronan Keryell

Gérald Ouvradou

—

Laboratoire d'Informatique des Télécommunications —

Département Informatique de

l'École Nationale Supérieure des Télécommunications de Bretagne

17 novembre 1999

Intérêt du reconfigurable dynamique

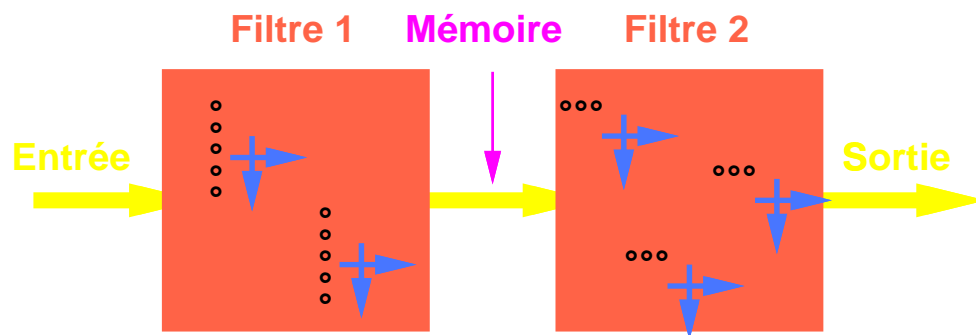
1

- Projeter une (grande) architecture virtuelle sur du (petit) matériel
- Architecture matérielle adaptative
- Optimisation directe du matériel avec spécialisation par logiciel
- Instanciation directe du concept de co-conception matériel/logiciel
- Compromis coût-performance : logiciel / matériel
- ⇔ Trouver l'« application qui tue »...

- Exemples d'applications du reconfigurable dynamique
- Modèle de programmation contre modèle d'exécution
- Expression des communications par transformation de programme
- Optimisation des transferts inter-opérateurs
- Compactage des mémoires temporaires

Filtrage

3



- Composition de différents filtres de taille et/ou forme différentes
- Calculs et communication en $\mathcal{O}(N^2)$
- Évaluation partielle : « tabulation » des opérateurs
- Cas général des traitements multirésolution

- Cas particulier : chaîne de segmentation

Transformée en ondelettes

5


- Localisation en fréquence *et* en espace
- Opérateurs de différentes tailles
- Compromis entre taille et nombre des opérateurs utilisables en parallèle
- Reconfiguration de la machine pour chaque taille d'opérateur

- Permet de simuler des phénomènes macroscopiques à partir de lois microscopiques et locales
 - ▶ Gaz
 - ▶ Réactions chimiques
 - ▶ Dépôts par plasma
 - ▶ ...
- Différents milieux hétérogènes
- \rightsquigarrow Différents modèles
- Reconfiguration dynamique des opérateurs sur les différents domaines en fonction des besoins



Filtrage adaptatif

7

- Changer les coefficients des filtres avec évaluation partielle
- Changer la configuration du filtre
- Faire tourner un nombre maximal de filtres en parallèle pour augmenter le débit
-  Débit mémoire et E/S ↗...



- Beaucoup de calculs simples
- Multiplexage temporel de différentes couches
- Apprentissage :
 - ▶ Changement *dynamique* des coefficients
 - ▶ Changement *dynamique* du nombre de neurones

Cryptographie


9

- Si algorithme asymétrique, 2 configurations distinctes :
 1. Chiffrement
 2. Déchiffrement
- Spécialisation du matériel en fonction de chaque clé
- Machine à casser les clés (EFF,...)

- Multiplexage temporel de fonctionnalités indépendantes ou pas
- Simulation d'un gros système ou circuit
- Chaîne de traitement d'image
- Coupleur ATM + cryptage
- Couplé au contexte Unix pour gérer différents utilisateurs :
Ardoise multi-utilisateur
↪ intérêt des FPGAs multi-contexte

Problématique

11

- Nombre de pattes des FPGAs limité
- Reconfiguration dynamique \implies des transferts de données ↗
entre configurations
-  Encore plus sensible au nombre de pattes limité...
- ↪ Optimisation de l'utilisation de la mémoire
- Utiliser si possible des transferts internes (registres, mémoires)
- Automatiser les transferts d'information dans la machine

Le rêve

- Langage de haut niveau
- Boucles imbriquées, séquentielles ou parallèles
- Accès à une mémoire globale
- Proche de la spécification de l'algorithme



Modèle d'exécution

13

Le dure réalité de la vie

- Programme de contrôle sur DSP, microprocesseur ou automate
- Chargement de configurations dans les FPGAs
- Lectures et écritures en mémoire locales
- Gestion du transfert d'information entre les opérateurs (registres, FIFO, mémoires)
- Contrôle des E/S
- Proche du matériel



- Cible : Ardoise (action ARP-ISIS, DSP+FPGA)
- But : aider le travail manuel
- Générer les codes de transfert mémoire, intra- ou inter-FPGA
- Chargement de configurations dans les FPGAs
- Gérer les transferts d'information entre les configurations ou opérateurs pour limiter les transferts mémoires
- Gérer les E/S
- Gérer les communications inter-FPGA
- Préchargement (données et configurations)
- Pipeline logiciel, recouvrement préchargements/calculs



Quelques Techniques de compilation utilisables

15

- Analyse fine du code à la compilation pour faire une orchestration précise
- Transformer & paralléliser le code (boucles,...)
 - ↪ Code équivalent correct par construction
 - Déroulage de boucle : ↗ unités fonctionnelles



```
do i = 0, 100
  c(i) = b(i) + a(2*i+4)
```

 ↪


```
do i = 0, 100, 4
  c(i) = b(i) + a(2*i+4)
  c(i+1) = b(i+1) + a(2*i+6)
  c(i+2) = b(i+2) + a(2*i+8)
  c(i+3) = b(i+3) + a(2*i+10)
```
 - Strip mining : découpage d'une boucle en 2 boucles



imbriquées \rightsquigarrow adaptation du code à la taille d'un opérateur
(réduction,...)

- Simplification du code (évaluation partielle,...)
- Extraire les macro-fonctions et sous-traiter à une chaîne de compilation FPGA
- Planifier l'exécution
- Préchargement des données et des reconfigurations, pavage multidimensionnel avec pipeline logiciel,...
- Générer le code DSP d'infrastructure
-  Comment choisir parmi toutes ces techniques ?

Exemple — filtrage

17

En FortranHDL :

```

program double_filtre
integer i,j,n,t
integer longtemps,tx,ty
parameter (tx = 512)
parameter (ty = 512)
integer op1x,op1y,op2x,op2y
parameter (op1x = 1)
parameter (op1y = 5)
parameter (op2x = 3)
parameter (op2y = 1)
integer image(1:tx,1:ty), image2(1:tx,1:ty), image3(1:tx,1:ty)
integer coeff1(-op1y/2:op1y/2), coeff2(-op2x/2:op2x/2)
do t = 1, longtemps
  do i = 1, tx

```

```

do j = 1, ty
  call entree(image(i,j))
enddo
enddo
do i = 1, tx
  do j = 1, ty
    image2(i,j) = 0
    if (j .ge. 1 + op1y/2 .and. j .le. ty - op1y/2) then
      do n = -op1y/2, op1y/2
        image2(i,j) = image2(i,j) + image(i,j+n)*coeff1(n)
      enddo
    endif
  enddo
enddo
do i = 1, tx

```

Exemple — filtrage

19

```

do j = 1, ty
  image3(i,j) = 0
  if (i.ge.1 + op2x/2 .and. i.le.tx - op2x/2) then
    do n = -op2x/2, op2x/2
      image3(i,j) = image3(i,j) + image2(i+n,j)*coeff2(n)
    enddo
  endif
enddo
enddo
do i = 1, tx
  do j = 1,ty
    call sortie(image3(i,j))
  enddo
enddo
enddo
end

```

Exemple de code à générer

21

Code tournant sur le DSP d'Ardoise avec 3 zones programmables
(par exemple 3 tranches FPGA + mémoires)
4 macro-fonctions implantées en matériel :

- ▶ entree
- ▶ sortie
- ▶ filtre_1
- ▶ filtre_2

```

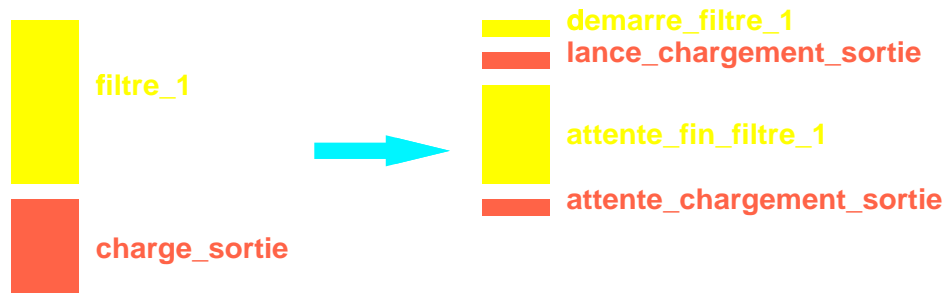
program double_filtre_DSP
integer t, longtemps
! Charge les configurations statiques :
call charge_entree
call charge_sortie
! Le temps qui passe :
do t = 1, longtemps
  call entree
  call charge_filtre_1
  call filtre_1
  call charge_filtre_2
  call filtre_2
  call sortie
enddo
end

```

Version pipelinée — principe

23

Ardoise avec 2 zones programmables



↪ Masquage des reconfigurations filtre_1 et sortie

Code tournant sur le DSP d'Ardoise avec 2 zones programmables
(entrée et filtre 2 sur une zone et sortie et filtre 1 sur l'autre)

```

program double_filtre_DSP
integer t
integer longtemps
!   Prologue du pipeline :
    call charge_entree

```

Version pipelinée — le code

25

```

!   Le temps qui passe avec pipeline :
do t = 1, longtemps
↓       call démarre_entree
        call charge_filtre_1
↑       call attente_fin_entree
↓       call démarre_filtre_1
        call charge_filtre_2
↑       call attente_filtre_1
↓       call démarre_filtre_2
        call chargement_sortie
↑       call attente_fin_filtre_2
↓       call démarre_sortie
        call charge_entree
↑       call attente_sortie
        enddo
end

```

`call fonction` : lance l'exécution synchrone d'une fonction de calcul

`call demarre_fonction` : démarre l'exécution asynchrone d'une fonction de calcul

`call attend_fonction` : attend la fin de l'exécution asynchrone d'une fonction de calcul

`call charge_fonction` : charge une zone de FPGA avec une fonction de calcul

`call lance_chargement_fonction` : lance le chargement asynchrone d'une fonction de calcul dans une zone de FPGA

`call attente_chargement_fonction` : attend la fin du chargement asynchrone d'une fonction de calcul dans une zone de FPGA

Code intermédiaire

27

- En entrée de l'outil
- Code partitionné et alloué
- Généré par d'autres phases du projet (cf Daniel Dours)
- Code de haut niveau (style C, Fortran,...)
- Transformations de source à source
- Mise au point plus facile
- Simulation possible en compilant classiquement et en liant avec un exécutif spécifique
- Rajout d'annotations par les phases précédentes pour exprimer
 - ▶ Une macro-fonction FPGA
 - ▶ Une information de placement
 - ▶ Exécution en parallèle possible

► ...

Utilisation possible d'un outil interactif (style PIPS) pour manipuler le code, optimiser, placer,...



Code intermédiaire typique

29

```
do i1 = ...
  ... = f1(...)
  ... = f2(...)
do i2 = ...
  ... = f3(...)
enddo
do i3 = ...
  ... = f4(...)
  ... = f5(...)
enddo
... = f6(...)
```

```
enddo
```

Définit le squelette d'exécution du séquençement (sur DSP par exemple).

Chaque exécution de fonction est déterminée par son vecteur d'itération I

$$I_{f_5} = (i_1, i_3)$$



```

do i = j, k
  a(i) = 3*b(2*i)*c(k)
  d(i) = a(i)+2

```

Devient

```

do i = j, k
  b'(2*i) = read(b(2*i))
  c'(k) = read(c(k))
  a'(i) = 3*b'(2*i)*c'(k)
  write(a'(i), a(i))
  a'(i) = read(a(i))
  d'(i) = a'(i)+2
  write(d'(i), d(i))

```



Expliciter les communications

31

- Code brut permettant une exécution sur Ardoise :
 - ▶ Les tableaux a, b et c résident en mémoire externe
 - ▶ Les tableaux a', b' et c' résident en mémoire dans le FPGA avant utilisation par l'opérateur *arbitrairement complexe*
 - ▶ Les fonctions read et write explicitent les communications (accès) entre la mémoire et le FPGA
- Code simpliste :
 - ▶ Alloue tous les tableaux en interne même si 1 seul élément utilisé
 - ▶ Communications redondantes
 - ▶ ⇔ Optimisations intensives pour aller vers un code efficace

```

do i = j, k
  b' = read(b(2*i))
  c' = read(c(k))

```




```
a' = 3*b'*c'  
write(a', a(i))  
d' = a'+2  
write(d', d(i))
```

Plus que du chemin de données !



Régions de PIPS

33

- Région *in*
 - ▶ Les éléments de tableaux nécessaires à un bloc d'instructions
- Région *out*
 - ▶ Les éléments de tableaux nécessaires pour la suite des calculs

Exprime exactement les données à transférer entre la mémoire et l'opérateur

- Génération des communications
- Privatisation implicite
- Élimination des transferts inutiles
- La taille de la région détermine la taille du temporaire à allouer en interne au FPGA



- Région *in* pour 2 itérations consécutives I et I'
- Si $\text{RegionIn}(I) \cap \text{RegionIn}(I') \neq \emptyset$ on peut réutiliser des données
 - ▶ Par une mémoire interne au FPGA
 - ▶ Par un registre à décalage si $\text{RegionIn}(I) = \text{RegionIn}(I') + \vec{c}$

Beaucoup d'optimisations sont possibles. Dans le pire des cas, la mémoire globale existe...



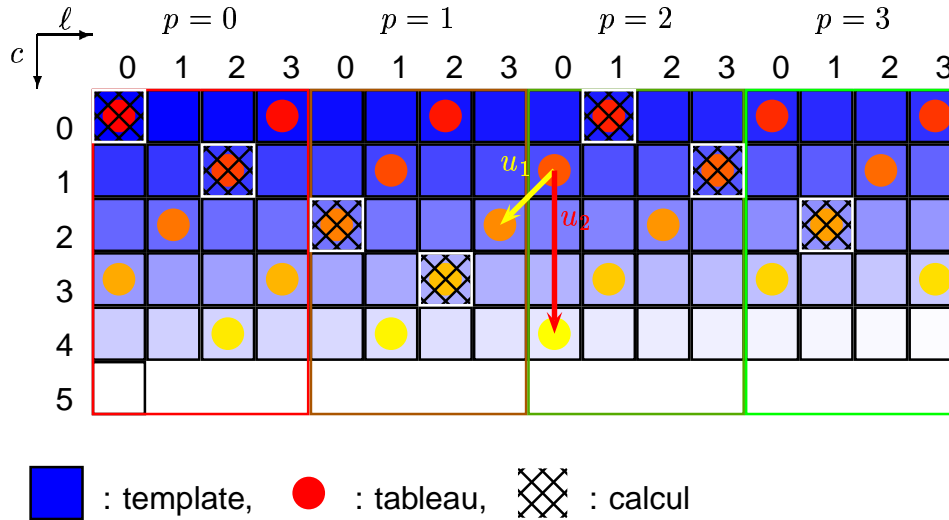
Gestion plus fine des bancs mémoires

35

- En réalité pas de mémoire unique...
- Chaque FPGA possède 2 bancs mémoire
- Utilisation possible d'annotations à la HPF pour distribuer les données
- 1 Ardoise à n tranche $\equiv 2 \times n$ bancs $\equiv 2 \times n$ processeurs virtuels
- Règle des écritures locales (*owner compute rule*) : les calculs ont lieu sur le FPGA possédant le membre gauche de l'assignation
- Placer les calculs intermédiaires \equiv placer des assignations intermédiaires
- Rajouter les équations de distribution et d'alignement dans le modèle



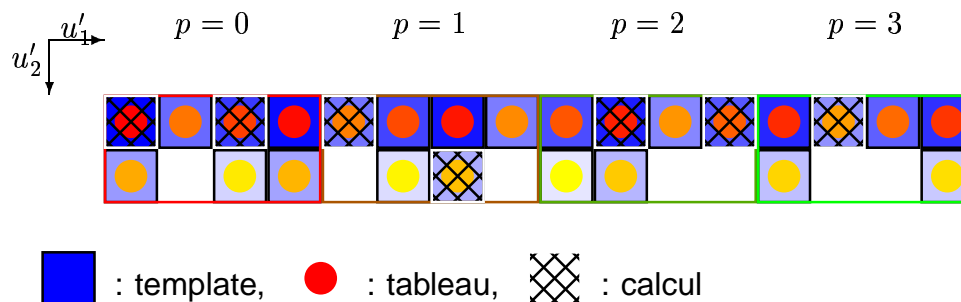
Extrait du schéma de compilation pour la machine Phénix de l'ETCA (1996) : perte de place sans optimisation



Allocation compacte

37

- HPF: autorise des trous
- Compromis temps-espace
- Méthode :
 - ▶ Base (c, ℓ)
 - ▶ Base du cristal (u_1, u_2) via
 - ▶ « Fortranisation » du tableau en (u'_1, u'_2) par division entière



- RAW au MIT
 - ▶ Cible : réseau sur 1 circuit de petits processeurs avec mémoire
 - ▶ Utilise aussi des techniques de compilation avancées (SUIF)
 - ▶ Distribution cyclique des données
- SynDEX à l'INRIA
 - ▶ Résoud le problème dans sa globalité
 - ▶ Qu'est-ce que cela donne appliqué à Ardoise ?

Conclusion

39

- Problématiques proches entre parallélisme et reconfigurable : threads et configurations
- Algèbre linéaire bon support pour manipuler les applications de traitement d'images
 - ▶ Grosse quantité de données
 - ▶ Bonne régularité globalement
 - ▶ Permet les domaines polyédriques quelconques
 - ▶ Outils mathématiques de manipulation
- Allocation mémoire et transferts de données : permet au concepteur de se recentrer sur choix algorithmiques et partitionnement (si non automatisé...)
- Continuer l'étude sur l'algorithme de segmentation d'Ardoise
- Préciser la technique de compilation

- Étudier l'exécutif de configuration et d'exécution et mise en œuvre
- Concrétiser dans un prototype logiciel...
- ...Retrouver un (bon) thésard et un finance
- Comme pour les ordinateurs //, la problématique est le nombre de pattes \rightsquigarrow limiter les E/S
- Avenir : FPGA de 10^9 portes à la RAW contenant de la mémoire distribuée ?

À faire (ailleurs)

41

- Partitionnement
- Choix de l'allocation dans les bancs mémoire

List of Slides

- | | | | |
|----|--|----|--------------------------------------|
| 1 | Intérêt du reconfigurable dynamique | 21 | Exemple de code à générer |
| 2 | Plan | 23 | Version pipelinée — principe |
| 3 | Filtrage | 24 | Version pipelinée — le code |
| 5 | Transformée en ondelettes | 26 | Exécutif pour la reconfiguration |
| 6 | Gaz sur réseau | 27 | Code intermédiaire |
| 7 | Filtrage adaptatif | 29 | Code intermédiaire typique |
| 8 | Réseaux de neurones | 30 | Expliciter les communications |
| 9 | Cryptographie | 33 | Régions de PIPS |
| 10 | Multitâche en général | 34 | Réutilisation des données |
| 11 | Problématique | 35 | Gestion plus fine des bancs mémoires |
| 12 | Modèle de programmation | 36 | Exemple d'accès mémoire creux |
| 13 | Modèle d'exécution | 37 | Allocation compacte |
| 14 | Objet de l'étude | 38 | Projets proches |
| 15 | Quelques Techniques de compilation utilisables | 39 | Conclusion |
| 17 | Exemple — filtrage | 41 | À faire (ailleurs) |
| | | 42 | Table des matières |