# Expression et optimisation des réorganisations de données dans du parallélisme de flots
CEA LIST

Pablo de Oliveira Castro[1,3]
Encadrant : Stéphane Louise[1]
Directeur de thèse : Denis Barthou[2]

[1]CEA LIST
[2]Université de Bordeaux
[3]Université de Versailles St Quentin

# Introduction

- Contexte
- Modèle d'exécution : Flots de données
- État de l'Art
- Problématique

# Contexte (1)

- Pour augmenter la puissance de calcul les fonderies se tournent vers les multicœurs :
  - Intel annonce un chip avec 80 cœurs
- Le marché de l'embarqué ne fait pas exception : Picochip, Tilera, Ambric (> 100 cœurs).
- Programmation des architectures multicœur difficile.
  - Comment exprimer du parallélisme dans l'application ?
  - Comment trouver une parallélisation efficace pour la cible choisie ?

# Contexte (2)

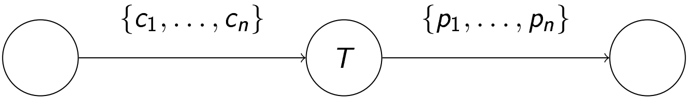
- Applications de traitement du signal embarquées :
  - dirigées par les données ;
  - routage de données statique ;
  - tâches s'exécutent sur un flot continu de données.
- Échange des données important pour obtenir une parallélisation efficace :
  - détermine les dépendances entre tâches ;
  - les patrons d'accès au données impactent les performances.
- Sur un grand nombre de cœurs l'adaptation manuelle à l'architecture des patrons d'accès est très couteuse.
- Comment exprimer et optimiser les échange de données dans ces applications ?

# Choix du modèle d'exécution

- Modèle de *Threads* non déterministe.
- Dans le contexte de l'embarqué [Jantsch05] identifie trois modèles d'exécutions adaptés :
  - Modèles temps-réel
  - Rendez-vous
  - Flots de données

# Modèle d'exécution Cyclo-Static DataFlow (CSDF)

- Modèle de flots de données : Cyclo-Static DataFlow (CSDF)[Bilsen 95]
  - déterminisme de l'application ;
  - ordonnancement en mémoire bornée ;
  - détection des interblocages.
- Graphe de tâches qui communiquent à travers des arcs
- Cycles de consommation et production fixes



# Filtres et Nœuds de routage

- On distinguera deux types de nœuds dans les graphes CSDF :
  - Les filtres qui effectuent des calculs arbitrairement complèxes sur les données.
  - Les nœuds de routage qui ne changent que l'ordre ou la multiplicité des éléments sur un flux.

- Dans cette thèse les calculs sur les données ne nous intéressent pas.

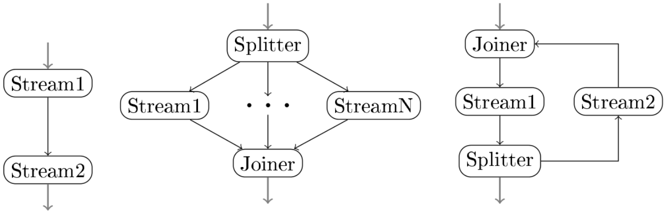
# Déclaration des dépendances entre filtres

- ArrayOL[Boulet 07], BlockParallel[Black-Schaffer 08] : réorganisation des données spécifiées par les dépendances d'entrée et de sortie des filtres.

- Les réorganisations de données ne sont pas programmées explicitement. Elles sont extraites des dépendances des filtres.

# Manipulation explicite des flots de données

- StreamIt[Gordon 02] : les données sur les flots sont réorganisées à l'aide de connecteurs,




Pipeline Split-Join Feedback Loop

- Brook[Liao 06] : les flots sont des objects de première classe manipulés par des fonctions (`streamStride`, `streamMerge`, etc.).

# Comparaison

| Langage | Expression | Optimisation |
|---|---|---|
| ArrayOL | haut-niveau | trans. pas de méthode automatique |
| BlockParallel | haut-niveau | optimisation sur cas simples |
| StreamIt | bas-niveau | transformation de graphes |
| Brook | bas-niveau | partitionnement affine |

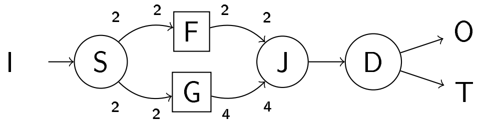
# Problématique

- Combiner l'expressivité d'un langage de haut-niveau avec des optimisations de flots de données efficaces.
- Deux niveaux de langages : principe *Separation of concerns*
  - SLICES haut-niveau pour l'expression
  - SJD bas-niveau pour l'optimisation
- Comment compiler un programme SLICES en SJD ?
- Comment optimiser un programme SJD : c'est-à-dire l'adapter à l'architecture ?

# Plan

# Présentation du langage SJD

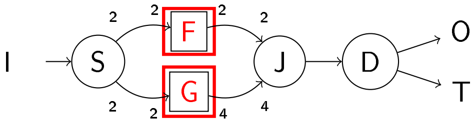
- SJD s'inscrit dans le modèle CSDF.




- Filtres : appliquent une fonction de calcul sur les entrées.
- Nœuds de routage :
  - Split : distribue les élements en entrée en tourniquet.
  - Join : rassemble les éléments sur les entrées en tourniquet.
  - Duplicate : duplique un flot.
  - Entrées & Sorties
  - Puits (T) : n'observe pas les éléments consommés.
  - Constante : produit un flot infini constant.
- Graphes arbitraires et cycliques.

# Présentation du langage SJD

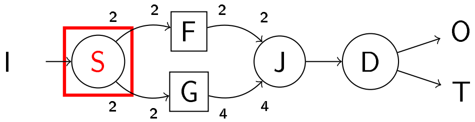
- SJD s'inscrit dans le modèle CSDF.




- Filtres : appliquent une fonction de calcul sur les entrées.
- Nœuds de routage :
  - Split : distribue les élements en entrée en tourniquet.
  - Join : rassemble les éléments sur les entrées en tourniquet.
  - Duplicate : duplique un flot.
  - Entrées & Sorties
  - Puits (T) : n'observe pas les éléments consommés.
  - Constante : produit un flot infini constant.
- Graphes arbitraires et cycliques.

# Présentation du langage SJD

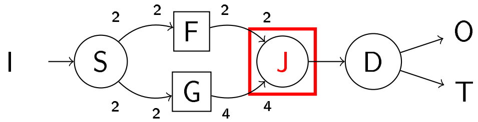
- SJD s'inscrit dans le modèle CSDF.




- Filtres : appliquent une fonction de calcul sur les entrées.
- Nœuds de routage :
  - Split : distribue les élements en entrée en tourniquet.
  - Join : rassemble les éléments sur les entrées en tourniquet.
  - Duplicate : duplique un flot.
  - Entrées & Sorties
  - Puits (T) : n'observe pas les éléments consommés.
  - Constante : produit un flot infini constant.
- Graphes arbitraires et cycliques.

# Présentation du langage SJD

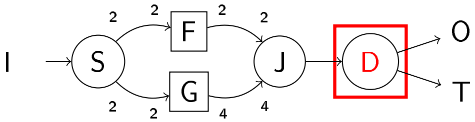
- SJD s'inscrit dans le modèle CSDF.




- Filtres : appliquent une fonction de calcul sur les entrées.
- Nœuds de routage :
  - Split : distribue les élements en entrée en tourniquet.
  - Join : rassemble les éléments sur les entrées en tourniquet.
  - Duplicate : duplique un flot.
  - Entrées & Sorties
  - Puits (T) : n'observe pas les éléments consommés.
  - Constante : produit un flot infini constant.
- Graphes arbitraires et cycliques.

# Présentation du langage SJD

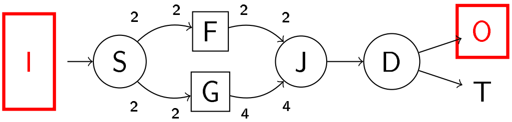
- SJD s'inscrit dans le modèle CSDF.




- Filtres : appliquent une fonction de calcul sur les entrées.
- Nœuds de routage :
  - Split : distribue les élements en entrée en tourniquet.
  - Join : rassemble les éléments sur les entrées en tourniquet.
  - Duplicate : duplique un flot.
  - Entrées & Sorties
  - Puits (T) : n'observe pas les éléments consommés.
  - Constante : produit un flot infini constant.
- Graphes arbitraires et cycliques.

# Présentation du langage SJD

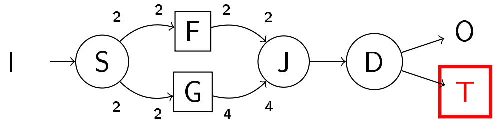
- SJD s'inscrit dans le modèle CSDF.




- Filtres : appliquent une fonction de calcul sur les entrées.
- Nœuds de routage :
  - Split : distribue les élements en entrée en tourniquet.
  - Join : rassemble les éléments sur les entrées en tourniquet.
  - Duplicate : duplique un flot.
  - Entrées & Sorties
  - Puits (T) : n'observe pas les éléments consommés.
  - Constante : produit un flot infini constant.
- Graphes arbitraires et cycliques.

# Présentation du langage SJD

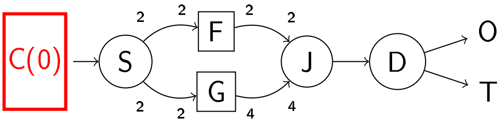
- SJD s'inscrit dans le modèle CSDF.




- Filtres : appliquent une fonction de calcul sur les entrées.
- Nœuds de routage :
  - Split : distribue les élements en entrée en tourniquet.
  - Join : rassemble les éléments sur les entrées en tourniquet.
  - Duplicate : duplique un flot.
  - Entrées & Sorties
  - Puits (T) : n'observe pas les éléments consommés.
  - Constante : produit un flot infini constant.
- Graphes arbitraires et cycliques.

# Présentation du langage SJD

- SJD s'inscrit dans le modèle CSDF.




- Filtres : appliquent une fonction de calcul sur les entrées.
- Nœuds de routage :
  - Split : distribue les élements en entrée en tourniquet.
  - Join : rassemble les éléments sur les entrées en tourniquet.
  - Duplicate : duplique un flot.
  - Entrées & Sorties
  - Puits (T) : n'observe pas les éléments consommés.
  - Constante : produit un flot infini constant.
- Graphes arbitraires et cycliques.

# Accès aux données : Multiplication de matrices

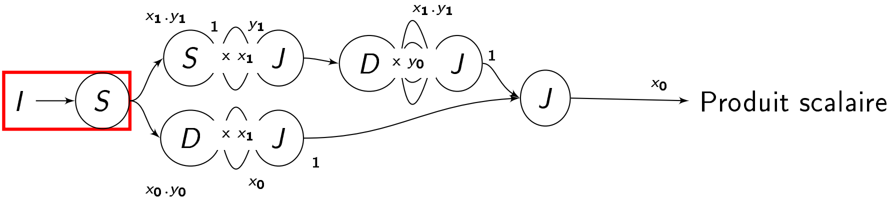
$C_{y_0,x_1} = A_{y_0,x_0} \times B_{y_1,x_1}$ et $x_0 = y_1$

```
for (j = 0; j < y0; j++)
  for (i = 0; i < x1; i++)
    for (k = 0; k < x0; k++)
      C[j,i] += A[j,k]*B[k,i]
```

- Pour effectuer le produit scalaire on accède aux données dans l'ordre :
  - $A_{0,0}, B_{0,0}, A_{0,1}, B_{1,0}, A_{0,2}, B_{2,0}, \ldots$
- Comment exprimer ce patron d'accès aux données en SJD ?

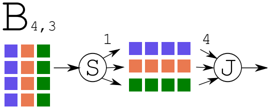
# Multiplication Matricielle en SJD

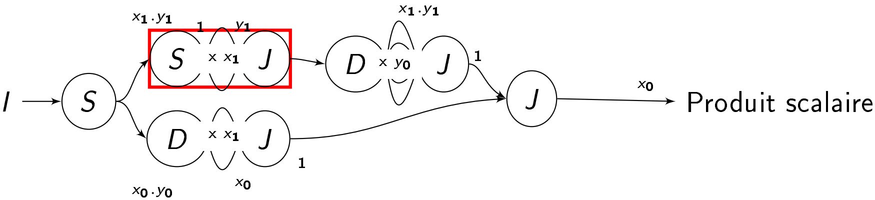
$$C_{y_0,x_1} = A_{y_0,x_0} \times B_{y_1,x_1}$$



# Multiplication Matricielle en SJD

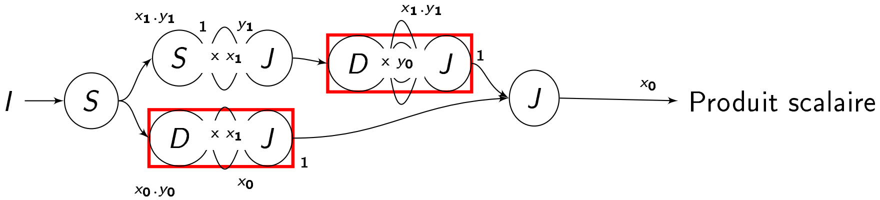
$$C_{y_0,x_1} = A_{y_0,x_0} \times B_{y_1,x_1}$$



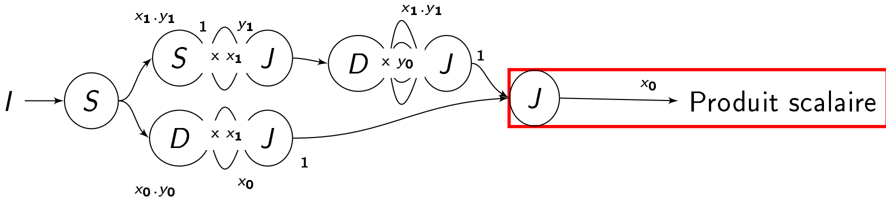
# Multiplication Matricielle en SJD

$$C_{y_0,x_1} = A_{y_0,x_0} \times B_{y_1,x_1}$$



# Multiplication Matricielle en SJD

$$C_{y_0,x_1} = A_{y_0,x_0} \times B_{y_1,x_1}$$



# Expressivité du langage SJD

- Les filtres peuvent exprimer des programmes Turing-complets
- ... mais les réorganisations de données exprimées dans les filtres sont opaques pour notre compilateur.
- Quelle est l'expressivité des nœuds de routage ?

### Théorème

Les graphes SJD composés uniquement de nœuds de routage réalisent exactement tous les arrangements avec répétition des données entrantes.

- StreamIt ne peut modéliser tous les arrangements, par exemple, impossible d'inverser l'ordre des éléments d'un vecteur.

# Objets du langage SLICES

Cinq concepts :

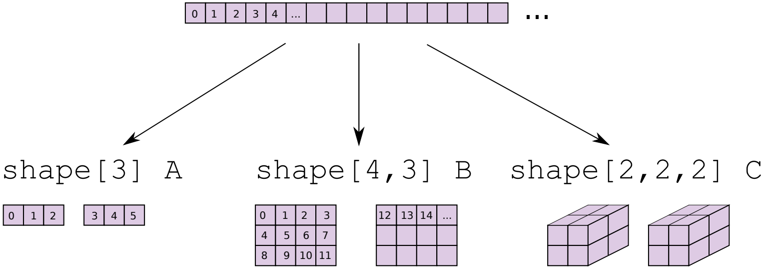
- Formes
- Grille
- Blocs
- Iterateur
- Zip

Un système de types garantit la correction des programmes à la compilation.

# Formes

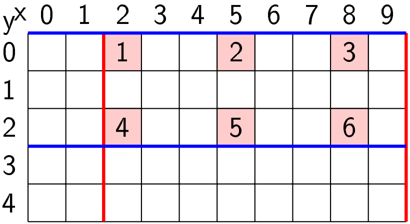
- Vue multidimensionnelle des données.

```
shape[a,b,c, ...] I = flot.input
```



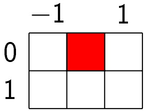
# Grilles

- Sélectionne des points régulièrement espacés sur une shape.
- `shape[10,5] I = flot input`
- `I[2:10:3, 0:3:2]`

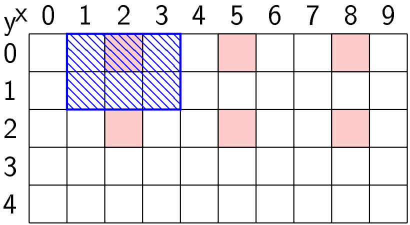


# Blocs

- Un bloc c'est une boîte de points à extraire : `(-1:1,0:1)`

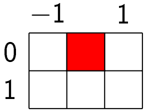



- Les blocs sont positionnés sur les points d'une grille.
- `I[2:10:3, 0:3:2]`$\times$(-1 :1,0 :1)

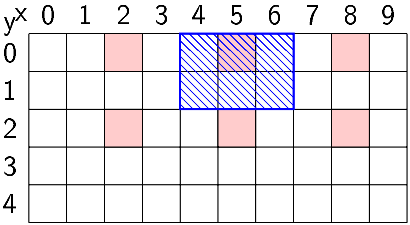


# Blocs

- Un bloc c'est une boîte de points à extraire : `(-1:1,0:1)`

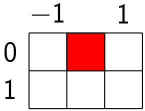



- Les blocs sont positionnés sur les points d'une grille.
- `I[2:10:3, 0:3:2]`×(-1 :1,0 :1)

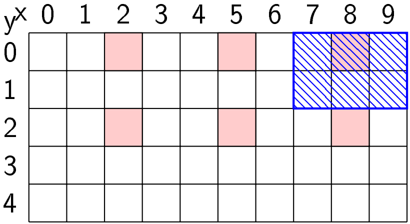


# Blocs

- Un bloc c'est une boîte de points à extraire : `(-1:1,0:1)`

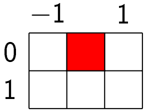



- Les blocs sont positionnés sur les points d'une grille.
- `I[2:10:3, 0:3:2]`×(-1 :1,0 :1)

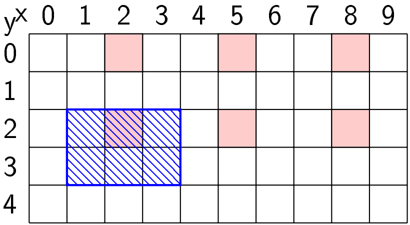


# Blocs

- Un bloc c'est une boîte de points à extraire : `(-1:1,0:1)`

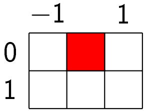



- Les blocs sont positionnés sur les points d'une grille.
- `I[2:10:3, 0:3:2]`$\times$(-1 :1,0 :1)

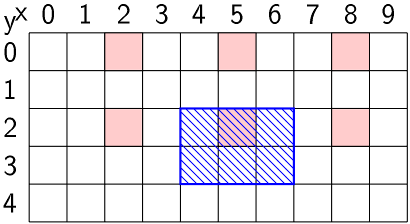


# Blocs

- Un bloc c'est une boîte de points à extraire : `(-1:1,0:1)`

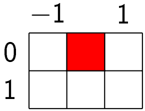



- Les blocs sont positionnés sur les points d'une grille.
- `I[2:10:3, 0:3:2]`$\times$(-1 :1,0 :1)

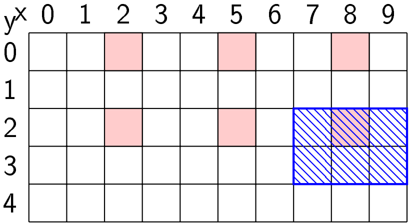


# Blocs

- Un bloc c'est une boîte de points à extraire : `(-1:1,0:1)`



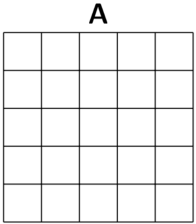

- Les blocs sont positionnés sur les points d'une grille.
- `I[2:10:3, 0:3:2]`$\times$(-1 :1,0 :1)



# Multiplication matricielle en SLICES

- Les itérateurs permettent de combiner les patrons d'accès précédents.
- On associe chaque ligne de A avec toutes les colonnes de B.

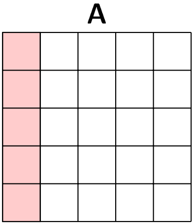
```
shape[5,5] A = A.input
shape[5,5] B = B.input
for l in A[0:1:1,0:5:1] x (0:4,0:0):
  for c in B[0:5:1,0:1:1] x (0:0,0:4):
    push l
    push c
```



# Multiplication matricielle en SLICES

- Les itérateurs permettent de combiner les patrons d'accès précédents.
- On associe chaque ligne de A avec toutes les colonnes de B.

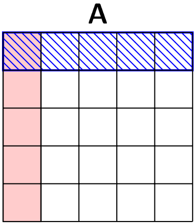
```
shape[5,5] A = A.input
shape[5,5] B = B.input
for l in A[0:1:1,0:5:1] x (0:4,0:0):
  for c in B[0:5:1,0:1:1] x (0:0,0:4):
    push l
    push c
```



# Multiplication matricielle en SLICES

- Les itérateurs permettent de combiner les patrons d'accès précédents.
- On associe chaque ligne de A avec toutes les colonnes de B.

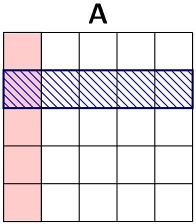
```
shape[5,5] A = A.input
shape[5,5] B = B.input
for l in A[0:1:1,0:5:1] x (0:4,0:0):
  for c in B[0:5:1,0:1:1] x (0:0,0:4):
    push l
    push c
```



# Multiplication matricielle en SLICES

- Les itérateurs permettent de combiner les patrons d'accès précédents.
- On associe chaque ligne de A avec toutes les colonnes de B.

```
shape[5,5] A = A.input
shape[5,5] B = B.input
for l in A[0:1:1,0:5:1] x (0:4,0:0):
  for c in B[0:5:1,0:1:1] x (0:0,0:4):
    push l
    push c
```



# Compilation de SLICES vers SJD

- On a implémenté un compilateur SLICES vers SJD.
- Pour la multiplication matricielle : le graphe produit est identique au graphe SJD écrit à la main.
- Le compilateur minimise le nombre de copies (nœud Duplicate) utilisés.

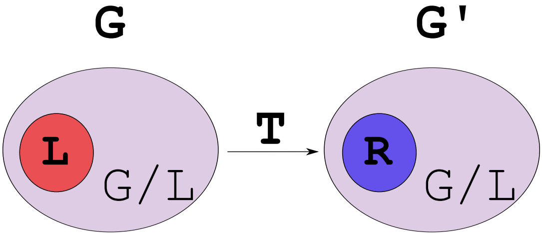
# Où en est-on ?

- Ce qu'on a vu :
  - Deux langages (SLICES, SJD) pour exprimer des réorganisations de données.
  - On peut compiler SLICES vers SJD.
- Ce qu'on va voir :
  - Comment optimiser un programme SJD avec des transformations de graphe.

# Optimisation par transformations

- Introduire un ensemble de transformations qui préservent la sémantique des programme.
- Ces transformations engendrent un espace de programmes équivalents.
- Choisir dans cet ensemble le programme le plus adapté à l'architecture.

# Correction d'une transformation CSDF




## Propriété

Une transformation $G \xrightarrow{T} G'$ d'un graphe CSDF est *correcte* si

- Les traces en sortie de $G$ et $G'$ sont identiques pour les mêmes entrées.
- La transformation n'introduit pas d'interblocages (vivacité).
- Après transformation le graphe reste ordonnançable en mémoire bornée (consistance).

# Condition suffisante de correction

$$\mathbf{L} \xrightarrow{\mathbf{T}} \mathbf{R}$$

### Lemme : Correction locale

Si une transformation $T$ qui remplace le sous-graphe $L$ par le sous-graphe $R$ vérifie

$$\begin{array}{rcr}
\exists b \in \mathbb{N}, & & \\
\forall I(L) = I(R) & \Rightarrow & O(L) \leq O(R) \\
\forall I(L) = I(R) & \Rightarrow & \max(\overline{O(R)} - \overline{O(L)}) \leq b \\
L \text{ est consistant} & \Rightarrow & R \text{ est consistant}
\end{array}$$

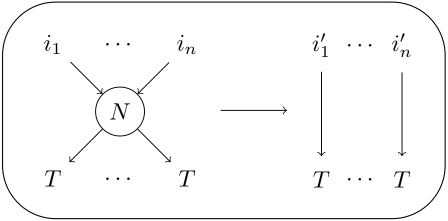
alors la transformation est *correcte* sur tout ordonnancement stationnaire independamment du contexte.

# Classification

- Deux familles de transformations :
  - Simplificatrices : enlèvent des nœuds ou des arcs inutiles.
  - Restructurantes : modifient la manière dont le routage de données est réalisé.

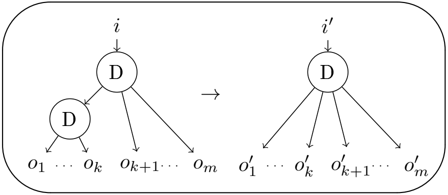
# Simplifications

- Suppressions :
  - élimination de code mort
  - réduction des identités
- Compactions :
  - repliage des hiérarchies de nœuds
  - fusion des canaux
- Suppression de synchronisation :
  - suppression de jonctions $J - S$
  - propagation de constantes



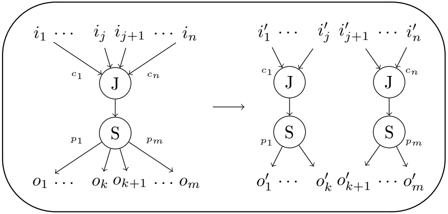
# Simplifications

- Suppressions :
  - élimination de code mort
  - réduction des identités
- Compactions :
  - repliage des hiérarchies de nœuds
  - fusion des canaux
- Suppression de synchronisation :
  - suppression de jonctions $J - S$
  - propagation de constantes



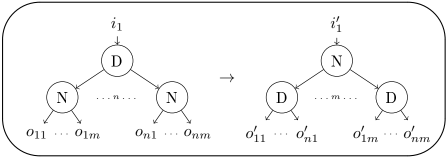
# Simplifications

- Suppressions :
  - élimination de code mort
  - réduction des identités
- Compactions :
  - repliage des hiérarchies de nœuds
  - fusion des canaux
- Suppression de synchronisation :
  - suppression de jonctions $J - S$
  - propagation de constantes



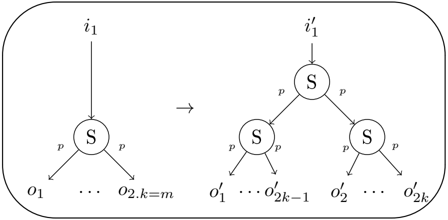
# Restructurations

- Factorisations : élimination des recalculs
  - inversion des Duplicate
  - inversion par uniformisation
- Regroupements : regroupement des $S$ et $J$ selon une congruence modulo $d$
- Déroulage : déroulage des exécutions successives d'un nœud pur



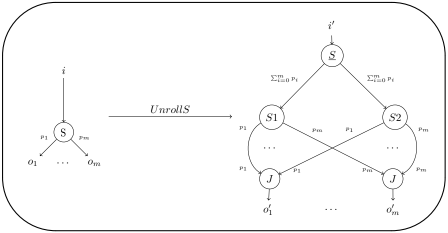
# Restructurations

- Factorisations : élimination des recalculs
  - inversion des Duplicate
  - inversion par uniformisation
- Regroupements : regroupement des $S$ et $J$ selon une congruence modulo $d$
- Déroulage : déroulage des exécutions successives d'un nœud pur



# Restructurations

- Factorisations : élimination des recalculs
  - inversion des Duplicate
  - inversion par uniformisation
- Regroupements : regroupement des $S$ et $J$ selon une congruence modulo $d$
- Déroulage : déroulage des exécutions successives d'un nœud pur



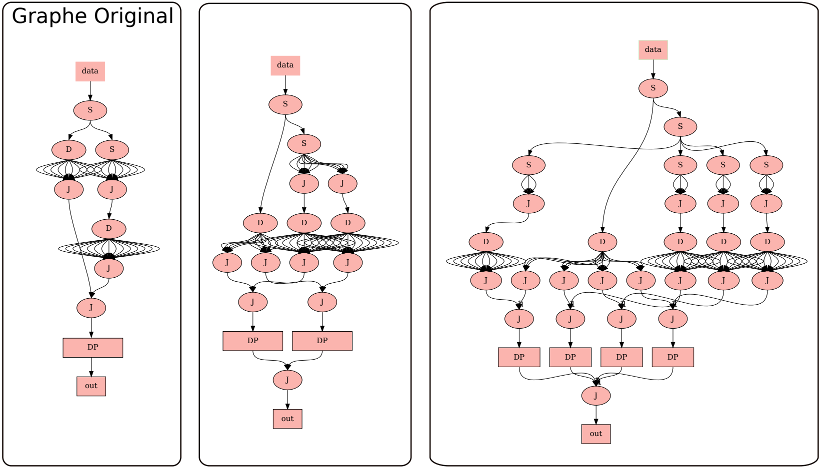
# Espace engendré

- Les dérivations de ces transformations engendrent un espace de programmes équivalents au programme original.
- Les transformations de déroulage rendent l'espace engendré infini...
- ... mais si on les compose avec des transformations simplificatrices :
  - Déroulage $\circ$ Suppression
  - Déroulage $\circ$ Sup. de synchronisation
  - Déroulage $\circ$ Déroulage$^{-1}$
- ... on montre que l'espace engendré reste fini.

# Exploration exhaustive (1/2)

- L'espace étant fini, il peut-être exploré indépendemment de la fonction objectif choisie.
- On part d'un graphe initial, chaque transformation possible va créer une nouvelle branche d'exploration.
- On itère jusqu'à ce que tous les graphes équivalents aient été produits.
- Complexité au moins exponentielle en fonction du nombre de nœuds du graphe initial.

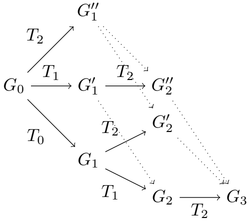
# Exploration exhaustive : Multiplication de matrices (2/2)




- On sélectionne le graphe qui optimise la fonction objectif $\phi(Graphes)$.

# Optimisations

- Couper les branches confluentes (transformations qui commutent)




- Appliquer les transformations simplificatrices systematiquement

# Heuristique : Recherche par faisceau

- Recherche par faisceau (Beamsearch) [Lowerre 76].
- Heuristique gloutonne : à chaque branchement on conserve uniquement les $n$ meilleurs candidats (selon la métrique $\phi$).
- Plus $n$ est large plus la portion d'espace explorée est large.

# Où en est-on ?

- Ce qu'on a vu :
  - On définit un ensemble de transformations sur les programmes SJD.
  - On sait explorer l'espace engendré par ces transformations.
- Ce qu'on va voir :
  - Comment réduire le coût des communications entre différents cœurs avec cette méthode.

# Backend

- Compile un graphe SJD vers du code C :
  - partitionnement
  - ordonnancement
  - fusion des tâches et des communications
  - génération de code
- Implémenté en Python ( 10000 lignes de code).

# Benchmarks

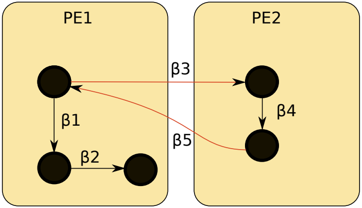
- Programmes SLICES compilés :
  - MM-COARSE et MM-FINE : multiplication de matrices (12x12)
  - GAUSS : Filtre de Gauss (sur des images 10x10 et 100x100)
  - HOUGH : Filtre de Hough (sur des images 10x10 et 100x100)
- Programmes SJD tirés de StreamIt :
  - FFT : transformée rapide de Fourier (vecteurs de taille 16)
  - DES : chiffrage (vecteurs de taille 16)
  - BITONIC : tri (vecteurs de taille 8)
  - DCT : transformée en cosinus discrète
  - FM : démodulateur FM
  - CHANNEL : vocodeur

# Réduction des communications

- Diminuer le coût des communications entre processeurs.
- Les communications au sein d'un même processeur ont un coût nul.
- Coût des communications sur un arc $e$ entre deux processeurs :

$$c = c_0 + \frac{\beta(e)}{bw}$$

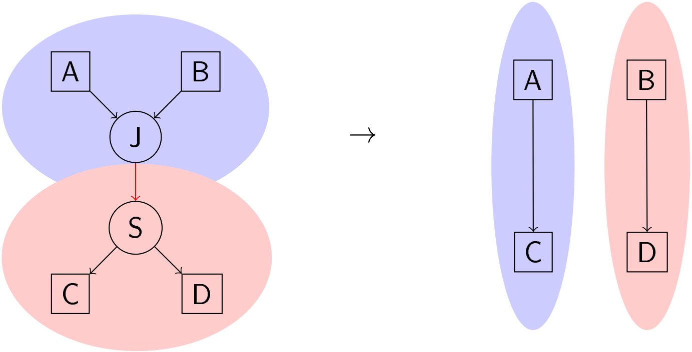
- Architecture faible latence : $c_0$ négligeable.



coût des communications = c(β3) +c(β5)

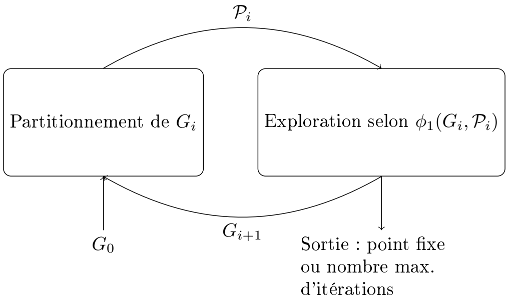
# Fonction objectif indirecte

- Lors de l'exploration le partitionnement n'est pas connu ...
- ... comment mesurer le coût inter-processeurs ?
- Métrique indirecte $\phi_0$ : réduire les points de synchronisation.

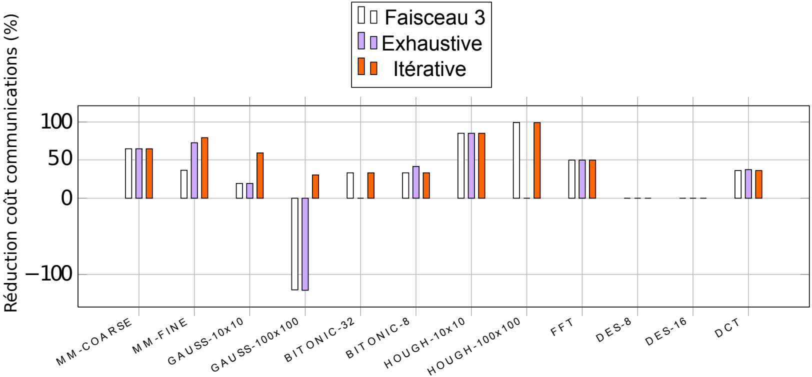


# Méthode itérative

- Métrique directe $\phi_1$ : coût de communications inter-processeur.
- Il faut connaître le partitionnement ...
- ... exploration itérative.



# Résultats sur une architecture à faible latence (1)

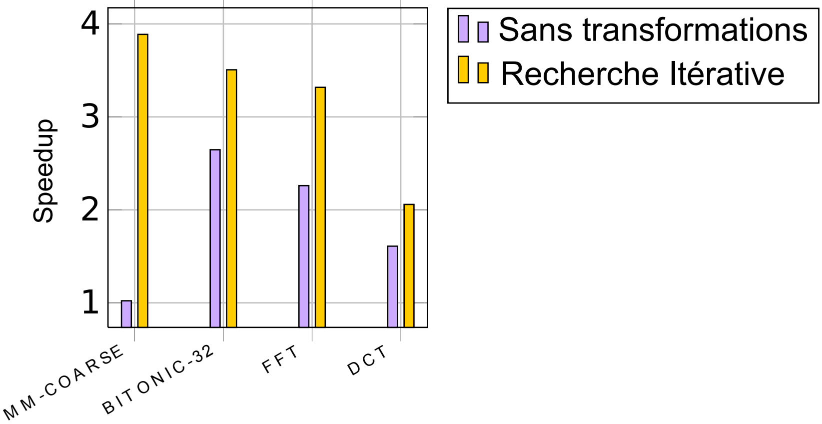



- Réduction du coût des communications sur une architecture à faible latence avec 4 cœurs.

# Résultats sur une architecture à faible latence (2)

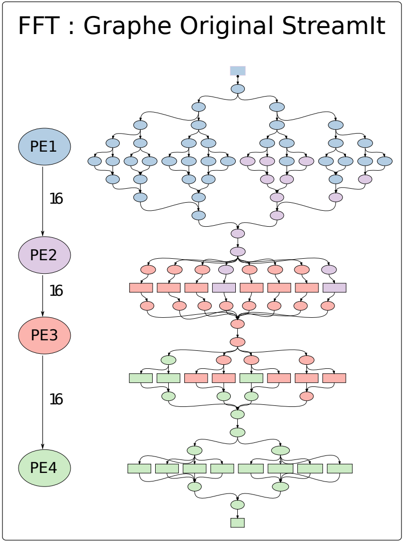
| | Exhaustive $\phi_0$ | | Faisceau 3 $\phi_0$ | | Itérative $\phi_1$ | |
|---|---|---|---|---|---|---|
| Programme | R(%) | T(s) | R(%) | T(s) | R(%) | T(s) |
| MM-COARSE | 64.9 | 2.2 | 64.9 | 2.5 | 64.9 | 4.5 |
| MM-FINE | 72.8 | 9.4 | 36.7 | 2.5 | 79.4 | 5.8 |
| GAUSS-10x10 | 19.4 | 230.9 | 19.4 | 51.3 | 59.5 | 9.9 |
| GAUSS-100x100 | -120.8 | 2558.1 | -120.2 | 18.1 | 30.5 | 18.4 |
| BITONIC-32 | - | stop | 33.3 | 575.9 | 33.3 | 925.4 |
| BITONIC-8 | 41.7 | 105.1 | 33.3 | 5.5 | 33.3 | 8.9 |
| HOUGH-10x10 | 85.2 | 1.3 | 85.2 | 1.6 | 85.2 | 3.9 |
| HOUGH-100x100 | - | stop | 99.3 | 91.4 | 99.2 | 411.7 |
| DES-16 | - | stop | 0.0 | 63.1 | 0.0 | 65.5 |
| DES-8 | 0.0 | 77.2 | 0.0 | 11.7 | 0.0 | 17.3 |
| FFT | 50.0 | 1148.6 | 50.0 | 18.8 | 50.0 | 10.0 |
| DCT | 37.5 | 0.5 | 36.3 | 0.5 | 36.3 | 10.3 |

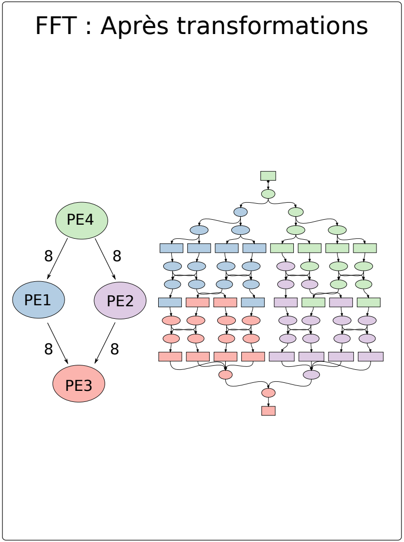
# Impact sur le speed-up (1/2)




- Quadricœur Nehalem Xeon W3520 cadencé à 2.67GHz avec 8MB de cache L3
- Code séquentiel de référence compilé avec `streamit -O3`
- MM-COARSE : `Backend SJD + clang-1.1 -O2`
- AUTRES : `Backend SJD + gcc-4.3 -O3`

# Impact sur le speed-up : FFT (2/2)



# Conclusion

- Problématique : optimiser l'accès aux données.
- On sépare les problèmes de l'expression (SLICES) et de l'optimisation (SJD)
- Expression SLICES :
  - Langage haut-niveau et multidimensionnel.
  - Presque aussi expressif que ArrayOL (itérations non parallèle aux axes).
  - Efficacement compilé.
- Optimisation SJD :
  - On étend les transformations de StreamIt aux graphes arbitraires.
  - Formalisation robuste des transformations.
  - On peut utiliser des métriques arbitrairement complexes pour l'exploration.
- Implémenté un prototype de cette chaîne de compilation.

# Perspectives

- Métriques plus complexes :
  - combiner plusieurs objectifs dans l'exploration
  - adapter les patrons d'accès aux méchanismes architecturaux (SIMD)
- Étendre les transformations à SJD avec contrôle
- Entrées et Transformations paramétriques
- SLICES déclaration des libertés d'ordre (réductions, etc.)

# Bibliographie


G. Bilsen, M. Engels, R. Lauwereins & JA Peperstraete.
*Cyclo-static data flow.*
In Acoustics, Speech, and Signal Processing, 1995.
ICASSP-95., 1995 International Conference on, volume 5, 1995.

D. Black-Schaffer.
*Block Parallel Programming for Real-Time Applications on Multi-core Processors.*
PhD thesis, Stanford University, 2008.

P. Boulet.
*Array-OL revisited, multidimensional intensive signal processing specification.*
Research Report RR-6113, INRIA, 2007.

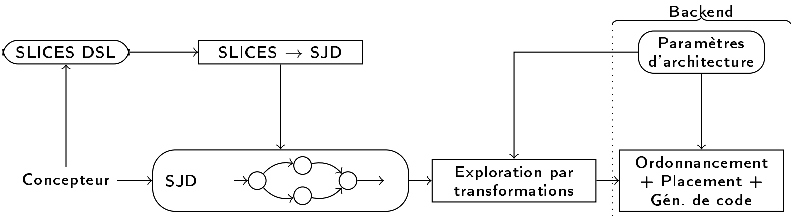
Yoonseo Choi, Yuan Lin, Nathan Chong, Scott Mahlke & Trevor Mudge.
*Stream Compilation for Real-Time Embedded Multicore Systems.*
In Int. Symp. on Code Generation and Optimization, pages 210–220, Washington, DC, USA, 2009. IEEE Computer Society.

Michael I. Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S. Meli, Andrew A. Lamb, Chris Leger, Jeremy Wong, Henry Hoffmann, David Maze & Saman Amarasinghe.
*"A Stream Compiler for Communication-Exposed Architectures".*
In Int. Conf. on Architectural Support for Programming Languages and Operating Systems, pages 291–303. ACM, 2002.

Shih-wei Liao, Zhaohui Du, Gansha Wu & Guei-Yuan Lueh.
*Data and Computation Transformations for Brook Streaming Applications on Multiprocessors.*
In Int. Symp. on Code Generation and Optimization, Washington, DC, USA, 2006. IEEE Computer Society.

Bruce T. Lowerre.
*The Harpy Speech Recognition System.*
PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1976.

# Vue globale du compilateur

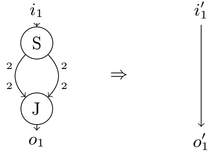


# Complexité des graphes

| *Programme* | *Nœuds* | *Arcs* | *Routage* | *Filtres* | *E/S* | *Cycles* |
|---|---|---|---|---|---|---|
| MM-COARSE | 16 | 52 | 10 | 4 | 2 | 0 |
| MM-FINE | 19 | 56 | 12 | 4 | 3 | 1 |
| GAUSS-10x10 | 32 | 50 | 24 | 2 | 6 | 0 |
| GAUSS-100x100 | 32 | 50 | 24 | 2 | 6 | 0 |
| HOUGH-100x100 | 16 | 10018 | 9 | 3 | 4 | 1 |
| HOUGH-10x10 | 16 | 118 | 9 | 3 | 4 | 1 |
| FFT | 110 | 145 | 82 | 24 | 4 | 0 |
| DES-8 | 215 | 286 | 96 | 101 | 18 | 0 |
| DES-16 | 423 | 566 | 192 | 197 | 34 | 0 |
| BITONIC-8 | 52 | 69 | 24 | 24 | 4 | 0 |
| BITONIC-32 | 374 | 598 | 130 | 240 | 4 | 0 |
| CHANNEL | 57 | 72 | 4 | 51 | 2 | 0 |
| FM | 43 | 53 | 14 | 27 | 2 | 0 |
| DCT | 40 | 69 | 4 | 34 | 2 | 0 |

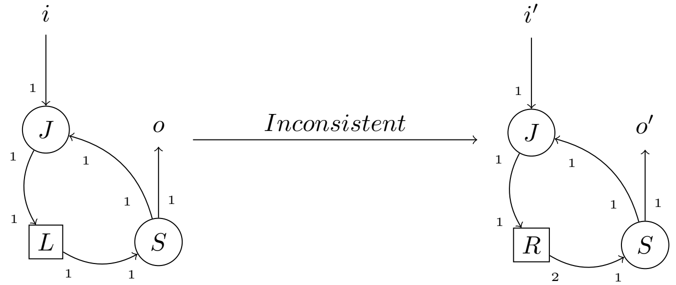
# Justification de la propriété préfixe de correction

$$I(G) = \{a, b, c, d, e\}$$




$$O(G) = \{a, b, c, d\} \leq O(G') = \{a, b, c, d, e\}$$

# Inconsistance

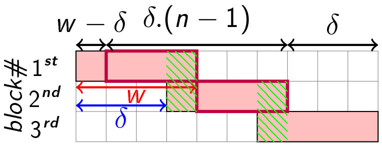


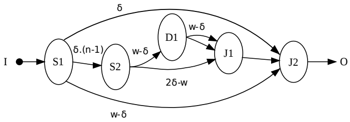

$f_L = \{pop(); push(1); \}$

$f_R = \{pop(); push(1); push(1); \}$

# Compilation patrons 1 dimension

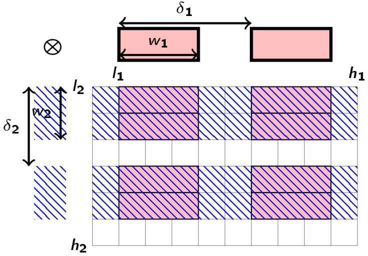
- block[1], trois cas possibles :
  1. $\frac{w}{\delta} \leq 1$, pas de chevauchement.
  2. $1 < \frac{w}{\delta} < 2$, chevauchement partiel.
  3. $2 \leq \frac{w}{\delta}$, chevauchement total.
- Pour chaque cas on produit un graphe SJD adéquat.



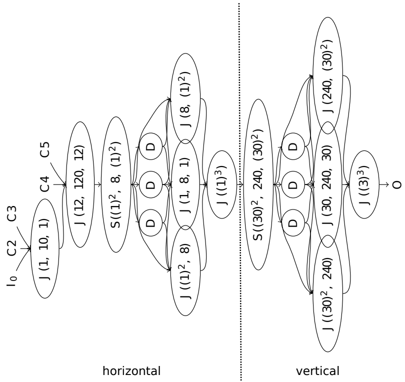



# Compilation patrons n dimensions

- block[n] produit cartésien de block[1]
- Graphe SJD block[n] reconstruit avec les graphes obtenus par projection sur chacune des dimensions (modulo redimensionnement et réordonnement).

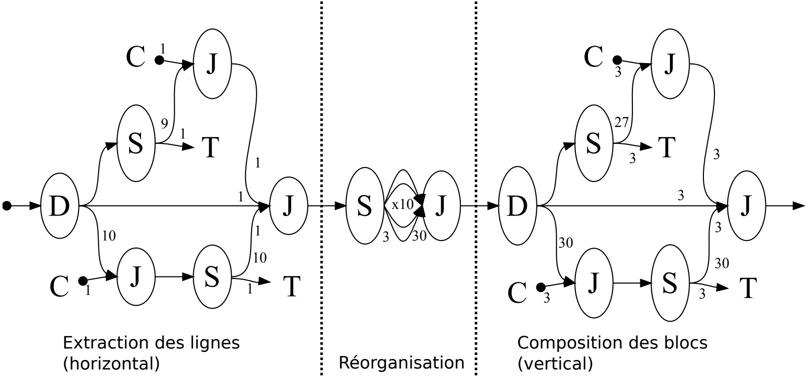


# Compilation du filtre Gaussien



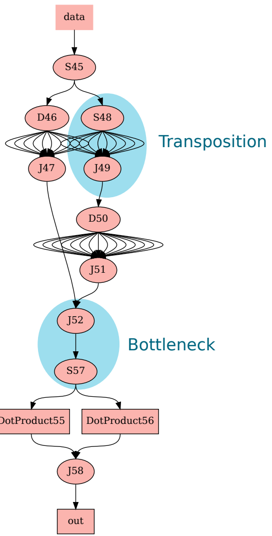

- Toutes les données dupliquées sont consommées.
- On duplique le plus tard possible : Réduire la bande passante des nœuds en aval.
- La nombre d'arcs ne dépend que de la taille des blocs extraits.

# Compilation du filtre Gaussien

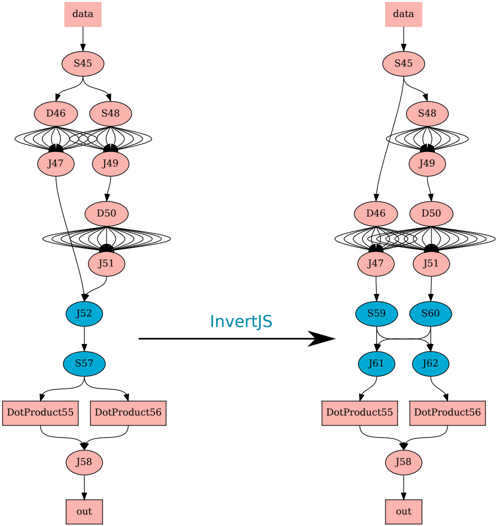



- Toutes les données dupliquées sont consommées.
- On duplique le plus tard possible : Réduire la bande passante des nœuds en aval.
- La nombre d'arcs ne dépend que de la taille des blocs extraits.

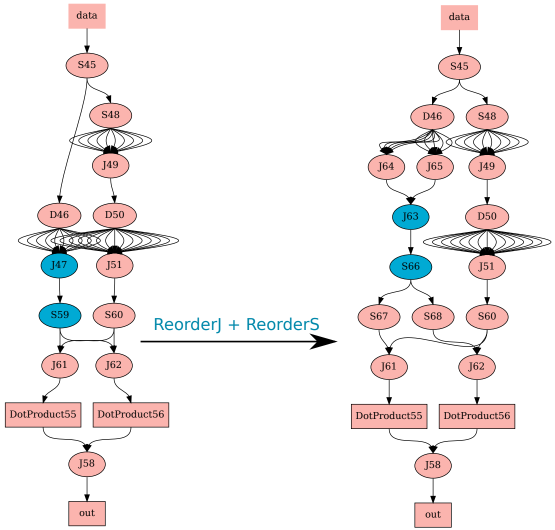
# Dérivation : Multiplication de matrices



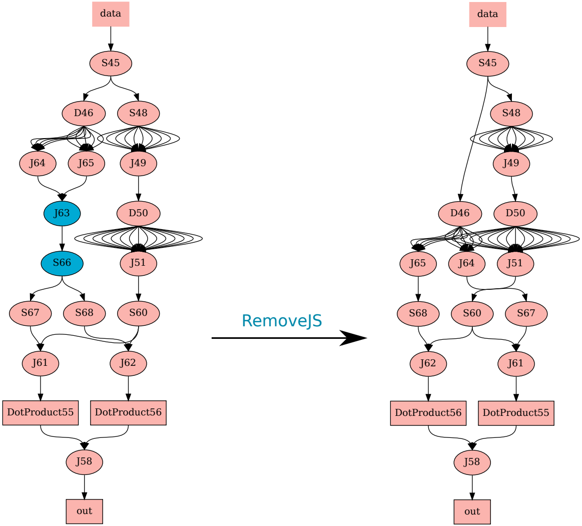
# Dérivation : Multiplication de matrices



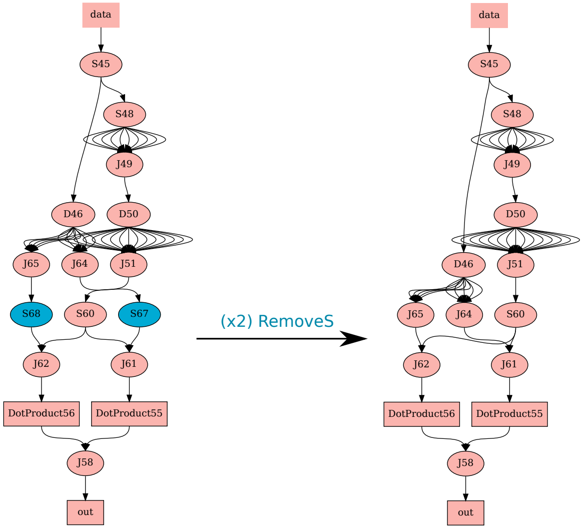
# Dérivation : Multiplication de matrices



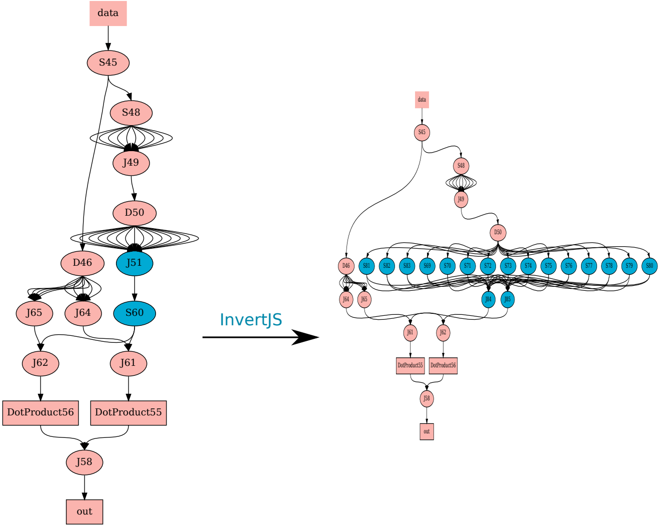
# Dérivation : Multiplication de matrices



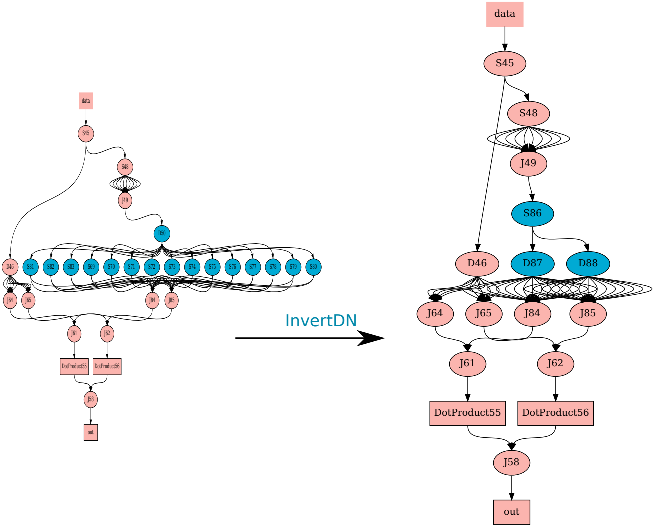
# Dérivation : Multiplication de matrices



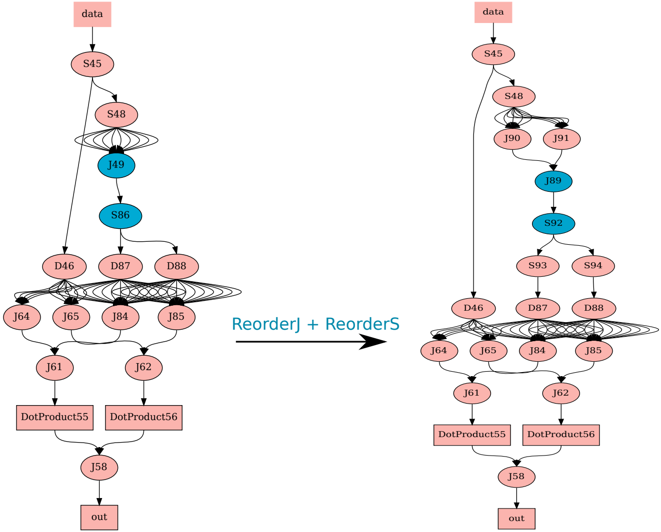
# Dérivation : Multiplication de matrices



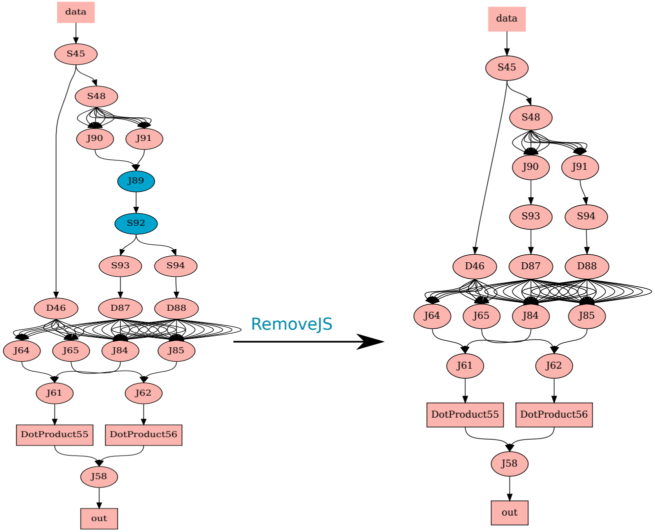
# Dérivation : Multiplication de matrices



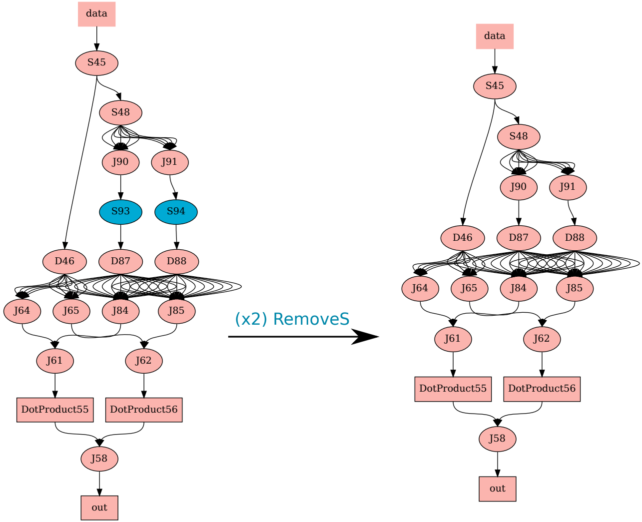
# Dérivation : Multiplication de matrices



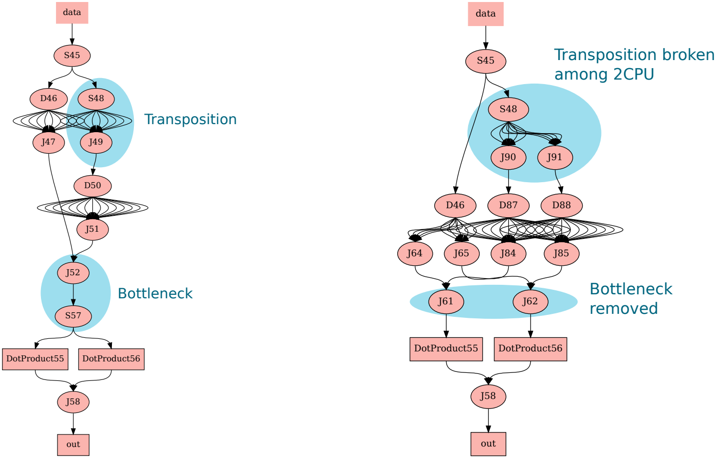
# Dérivation : Multiplication de matrices



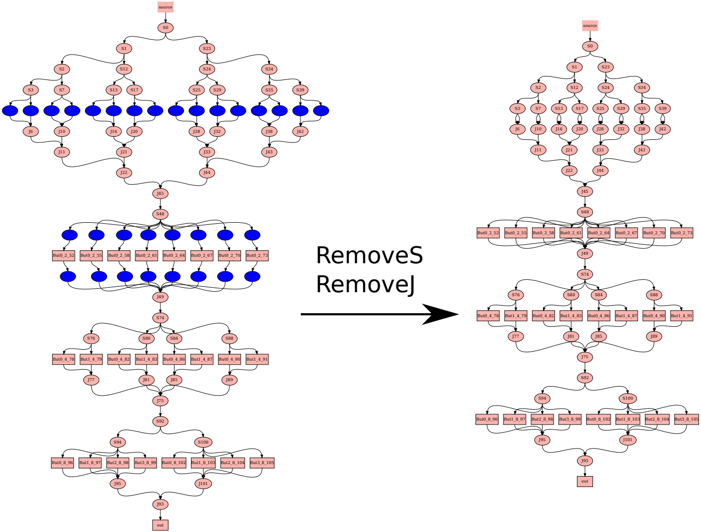
# Dérivation : Multiplication de matrices



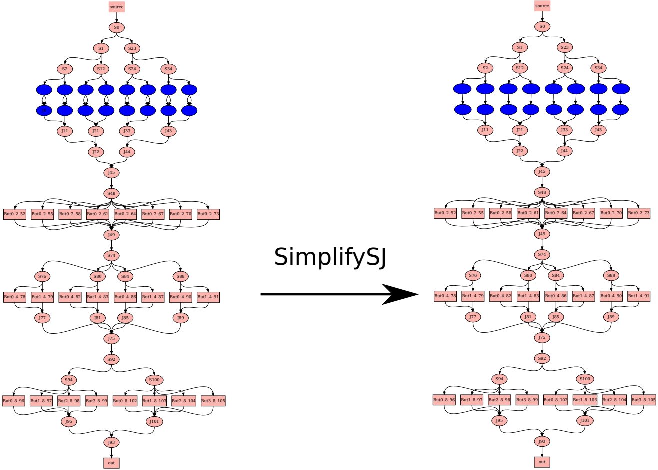
# Dérivation : Multiplication de matrices



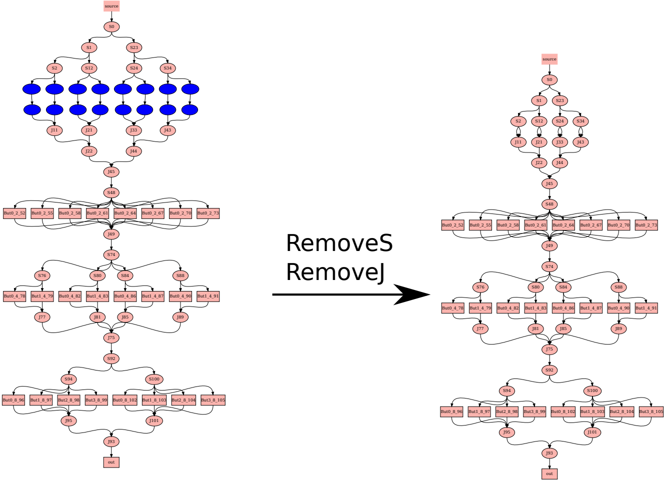
# Dérivation : FFT



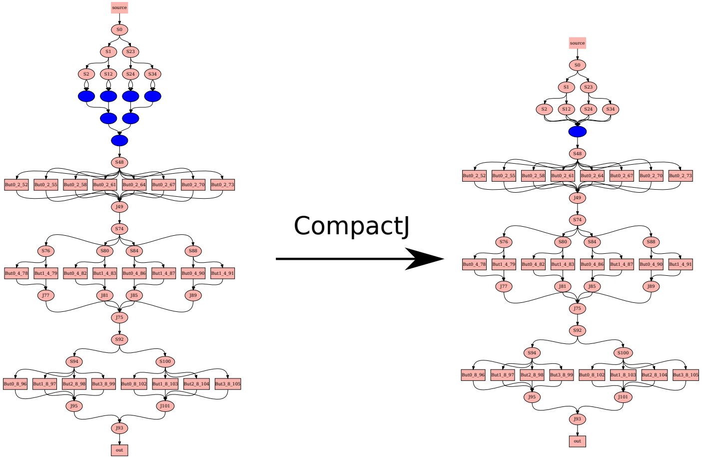
# Dérivation : FFT



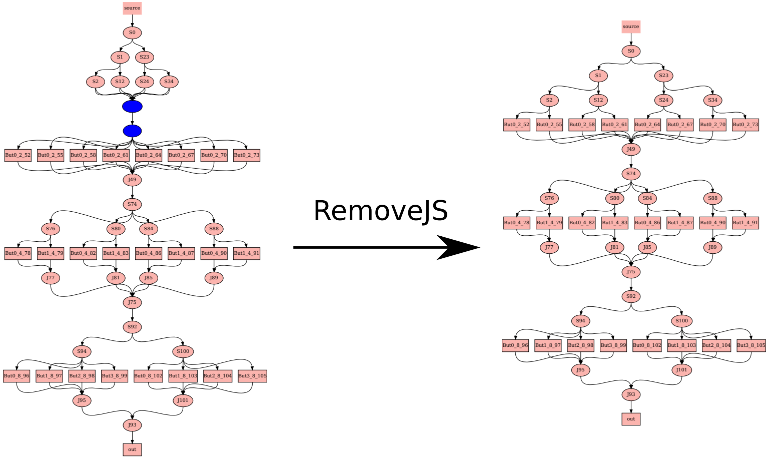
# Dérivation : FFT



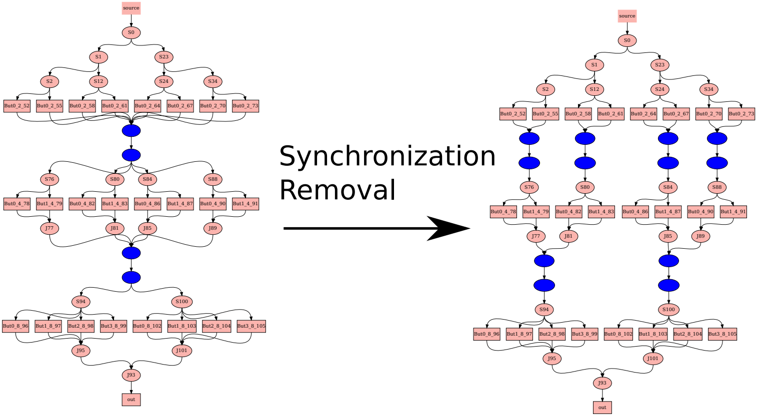
# Dérivation : FFT



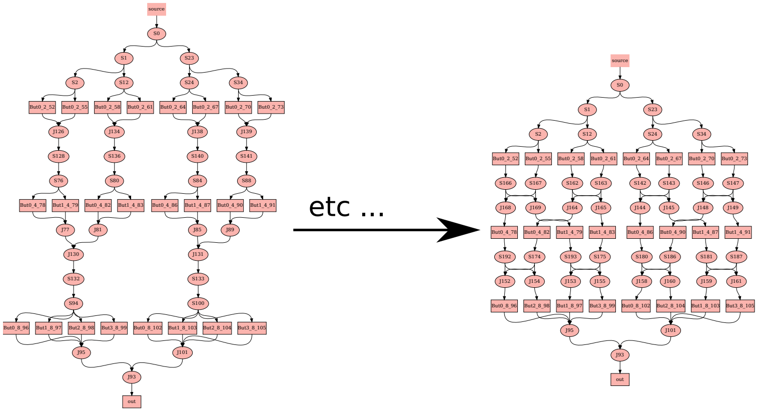
# Dérivation : FFT



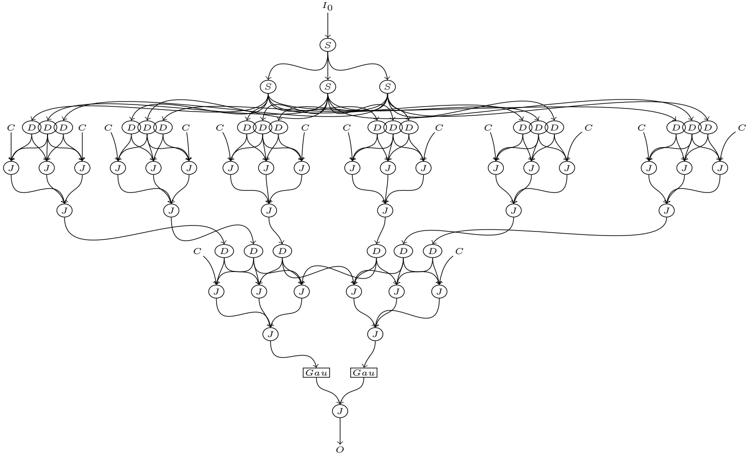
# Dérivation : FFT



# Dérivation : FFT

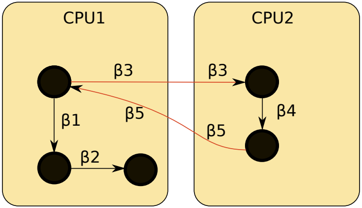


# Dérivation : Gauss



# Réduction de la mémoire

- Pas de mémoire partagée.
- On ne prends pas en compte la mémoire d'instructions.




mem1=β1+β2+β3+β5 mem2=β3+β4+β5

- On veut minimiser la mémoire locale nécessaire par processeur :

$$minP = max(mem_i)$$

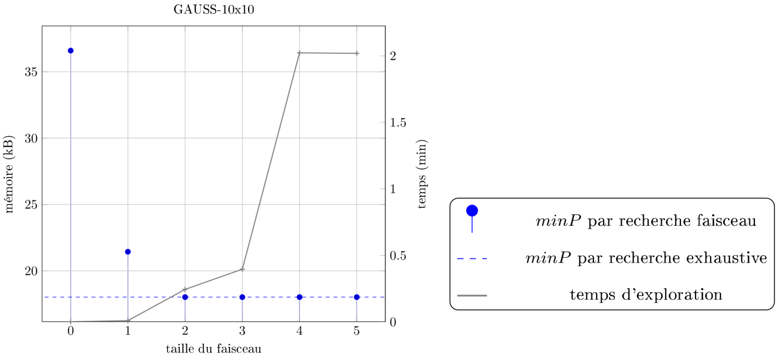
# Fonction objectif

- Fonction objectif : réduit la mémoire maximale par arc, puis la mémoire totale du graphe,

$$\phi_{mem}(G) = (max_{e \in G} \beta(e), \sum_{e \in G} \beta(e))$$

- On mesure *minP* avec la partitionnement [Choi 09] :
  - sur le graphe original
  - sur le graphe qui minimise $\phi_{mem}$ obtenu par transformations

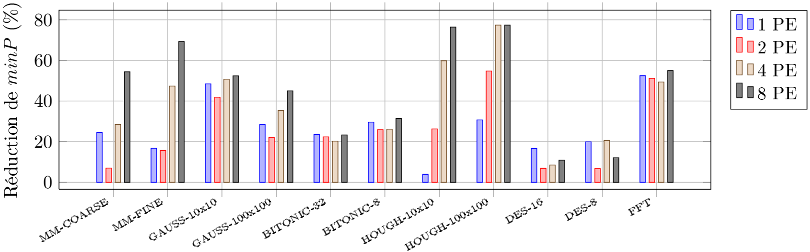
# Réduction sur 4 cœurs pour Gauss-10x10



# Faisceau contre Exhaustive

| | Réduction mémoire (%) | | Temps d'exploration ($s$) | |
|---|---|---|---|---|
| Programme | Exhaustive | Faisceau-3 | Exhaustive | Faisceau-3 |
| MM-COARSE | 28.4 | 28.4 | 1.0 | 1.1 |
| MM-FINE | 47.4 | 47.4 | 4.3 | 2.2 |
| GAUSS-10x10 | 50.8 | 50.8 | 107.9 | 23.7 |
| GAUSS-100x100 | 39.9 | 35.3 | 1140.2 | 59.3 |
| BITONIC-32 | - | 20.3 | interrompu | 418.5 |
| BITONIC-8 | 29.9 | 26.2 | 43.0 | 3.7 |
| HOUGH-10x10 | 59.6 | 59.8 | 0.6 | 0.6 |
| HOUGH-100x100 | - | 77.4 | interrompu | 40.8 |
| DES-16 | - | 8.5 | interrompu | 61.2 |
| DES-8 | 24.6 | 20.6 | 34.3 | 6.7 |
| FFT | 50.6 | 49.4 | 511.4 | 12.6 |
| CHANNEL | 2.9 | 2.9 | 0.0 | 0.1 |
| DCT | 7.9 | 7.9 | 0.4 | 0.4 |
| FM | 12.5 | 12.5 | 0.0 | 0.0 |

# Réduction de la mémoire en fonction du nombre de cœurs




- Réductions négligeables pour DCT, FM et CHANNEL : utilisation faible des nœuds de routage.