UNIVERSITE DE VERSAILLES SAINT-QUENTIN-EN-YVELINES

Ecole Doctorale Sciences et Technologies de Versailles - STV

Laboratoire Parallélisme, Réseaux, Systèmes et Modélisation

THÈSE DE DOCTORAT

DE L'UNIVERSITE DE VERSAILLES SAINT–QUENTIN–EN YVELINES

Spécialité : Informatique

Présentée par : Pablo de Oliveira Castro Herrero

Pour obtenir le grade de Docteur de l'Université de Versailles Saint-Quentin-en-Yvelines

Expression et optimisation des réorganisations de données dans du parallélisme de flots

soutenue le 14 décembre 2010

Devant le jury composé de :

Rapporteurs :	Albert Cohen, Directeur de recherche, INRIA
	François Irigoin, Maître de recherche, Mines ParisTech
Examinateurs :	Michel Harrand, Directeur Technique, Kalray
	William Jalby, Professeur de l'université de Versailles
Directeur de thèse :	Denis Barthou, Professeur de l'université de Bordeaux
Co-directeur :	Stéphane Louise, Ingénieur de recherche, CEA

Numéro national d'enregistrement :

Résumé

Pour permettre une plus grande capacité de calcul les concepteurs de systèmes embarqués se tournent aujourd'hui vers les MPSoC. Malheureusement, ces systèmes sont difficiles à programmer. Un des problèmes durs est l'expression et l'optimisation des réorganisations de données au sein d'un programme. Dans cette thèse nous souhaitons proposer une chaîne de compilation qui : 1) propose une syntaxe simple et haut-niveau pour exprimer le découpage et la réorganisation des données d'un programme parallèle ; 2) définisse une exécution déterministe du programme (critique dans le cadre des systèmes embarqués) ; 3) optimise et adapte les programmes aux contraintes de l'architecture.

Pour répondre au point 1) nous proposons un langage haut-niveau, SLICES, qui permet de décrire les réorganisation de données à travers des découpages multidimensionnels.

Pour répondre au point 2) nous montrons qu'il est possible de compiler SLICES vers un langage de flots de données, SJD, qui s'inscrit dans le modèle des *Cyclostatic Data-Flow* et donc admet une exécution déterministe.

Pour répondre au point 3) nous définissons un ensemble de transformations qui préservent la sémantique des programmes SJD. Nous montrons qu'il existe un sous-ensemble de ces transformations qui génère un espace de programmes équivalents fini. Nous proposons une heuristique pour explorer cet espace de manière à choisir la variante la plus adaptée à notre architecture. Enfin nous évaluons cette méthode sur deux problèmes classiques : la réduction de la mémoire consommée et la réduction des communications d'une application parallèle.

Abstract

Embedded systems designers are moving to multicores to increase the performance of their applications. Yet multicore systems are difficult to program. One hard problem is the expression and the optimization of data reorganizations. We would like to propose a compilation chain that : 1) uses a simple high-level syntax to express data reorganizations within a parallel application ; 2) ensures the deterministic execution of the program (critical in an embedded context) ; 3) optimizes and adapts the program to the target's constraints.

To address point 1) we propose a high-level language, SLICES, describing data reorganizations through multidimensional slicings.

To address point 2) we show that it is possible to compile SLICES to a data-flow language, SJD, that is built upon the Cyclostatic Data-Flow formalism and therefore ensures determinism.

To address point 3) we define a set of transformations that preserve the semantics of SJD programs. We show that a subset of these transformations generates a finite space of equivalent programs. This space can be efficiently explored with an heuristic that selects the program variant more fit to the target's constraints. Finally, we evaluate this method on two classic problems : reducing memory and reducing communication costs in a parallel application.

Remerciements

Tout d'abord, je tiens à remercier vivement Denis Barthou et Stéphane Louise pour leur encadrement. Durant ces trois ans, ils ont su me conseiller dans les moment difficiles et m'ont temoigné leur confiance ainsi que leur sympathie. Merci !

Je remercie mes rapporteurs, Albert Cohen et François Irigoin, pour leur relecture minutieuse de ce manuscrit ainsi que pour leurs retours précieux.

Je remercie également Michel Harrand et William Jalby de m'avoir fait l'honneur de participer à mon jury de thèse.

Je tiens à remercier tous les membres du laboratoire LaSTRE du CEA, qui m'ont accueilli chaleureusement durant ces trois ans. Merci pour leur conseils, pour les discussions scientifiques enrichissantes que nous avons eues et pour les bons moments que nous avons passés à la pause café.

Merci à Alexandre Guerre, Ilias Garnier, Julien Hervé, Marie Clermontelle, Matthieu Lemerre, Nicolas Benoît, Pierre-Aimé Agnel, Renaud Sirdey, Sergiu Carpov, Thomas Megel et Vincent David qui ont relu mes articles ou ce manuscrit et y ont apporté de nombreuses corrections.

Je tiens à remercier aussi Samuel Tardieu et Alexis Polti, anciens professeurs et surtout amis, pour tout ce qu'ils m'ont appris et continuent à me faire découvrir dans l'informatique et l'électronique. Merci de m'avoir aidé à préparer ma soutenance.

Enfin, je remercie tous mes amis : sans leur présence, je ne serais jamais venu au bout de ma thèse et la vie serait bien moins drôle ! Merci à Alexandre, Ali, Alice, Alvaro, Ander, André, Antoine, Asier, Baptiste, Bruno, Claire, Corentin, Daniel, Erwan, Fanny, Ilias, Ingrid, Laure, Loulou, Marie et Marie et Marie, Marion, Mathieu, Miguel, Nancy, Nicolas, Prisca, Kikedao, Stéph, Thomas, Yann, YanYan, Xor, et tous les autres.

Merci à mon grand-père Robert qui m'a donné, petit, le goût de l'informatique.

Au delà des mots, merci à Pierre Alain, Patricia, Manuel et Pierre-Aimé d'être toujours là quand il faut.

Table des matières

Chapitre 1

Introduction

Sommaire

Parce qu'il devient de plus en plus difficile d'augmenter la puissance des processeurs avec un seul cœur de calcul, depuis quelques années les fonderies produisent des processeurs à plusieurs cœurs de calcul. Pour qu'un programme puisse tirer parti de ces nouvelles architectures, il faut le paralléliser, c'est-à-dire le découper en tâches s'exécutant de manière concurrente sur l'ensemble des cœurs.

On peut décomposer la parallélisation d'un programme en deux étapes. La première consiste à extraire le parallélisme en partitionnant le programme en tâches exécutables de manière concurrente. La deuxième consiste à adapter le parallélisme à l'architecture en répartissant les tâches sur les différents cœurs et en générerant leur code.

Dans cette thèse nous nous intéressons à la parallélisation efficace d'applications de traitement du signal sur systèmes embarqués. Les applications de traitement du signal sont souvent dirigées par les données : les flux de données traités par l'application contrôlent l'ordre des opérations. Dans ce type d'application les opérations de routage et de réorganisation de données vont influencer de manière critique la manière de paralléliser le programme. D'une part, elles déterminent l'indépendance de deux tâches et par conséquent le fait qu'elles soient exécutables de manière concurrente. D'autre part, elles déterminent la quantité de données que chaque tâche est amenée à traiter. Leur analyse permet d'estimer les besoins en puissance de calcul, en mémoire et en communications pour chaque tâche. Cette analyse permet un équilibrage de charge des tâches sur les différentes ressources de l'architecture de manière à obtenir les meilleures performances.

Forts de ce constat, dans cette thèse nous nous sommes intéressés à l'étude des opérations de communication et de réorganisation de données dans la parallélisation de programmes de traitement du signal.

1.1 Contexte

En 1965 G. Moore remarque que le nombre de transistors des processeurs produits par l'industrie double tous les deux ans [Moo75]. Cette loi empirique s'est traduite par une augmentation régulière de la capacité de traitement des processeurs monocœur durant les trente dernières années. Cette progression s'est néanmoins ralentie. Le parallélisme d'instructions et l'augmentation de la taille des pipelines d'instructions ont permis de repousser le déclin de croissance de quelques années encore. Cependant ces techniques atteignent aujourd'hui leurs limites[OH05], les fabriquants de microprocesseurs se sont donc tournés vers les multicœurs pour augmenter encore plus la puissance de calcul des ordinateurs. Certains [ABC+06] reformulent donc la loi de Moore pour les années à venir : "Le nombre de cœurs à l'intérieur d'une puce double tous les 18 mois".

On retrouve également cette tendance dans le marché de l'embarqué où l'on voit apparaître des processeurs multicœurs comme Ambric, Tilera ou PicoChip. Ces processeurs présentent plusieurs avantages dont nous donnons quelques exemples. Par leur plus grande puissance, ils permettent d'exécuter des application lourdes en calcul (p. ex. encodage ou traitement d'un flux vidéo temps réel) sur une plateforme embarquée. En remplaçant des applications traditionnellement implémentées sur un ASIC (p. ex. modulation de signal téléphonie radio) par un code tournant sur un processeur générique multicœur on diminue les coûts de développement et on gagne en modularité puisque le programme peut être modifié à la volée. Enfin, la consommation d'un processeur grandit de manière exponentielle avec sa fréquence ; en répartissant une application sur plusieurs processeurs moins rapides, nous diminuons la consommation énergétique du système, ce qui est critique pour des cibles embarquées sur batterie.

Pour tirer parti de ces avantages, il faut néanmoins paralléliser efficacement les applications, ce qui n'est possible que si l'on trouve un "bon" découpage du programme en tâches. Tous les découpages ne permettent pas de réaliser un gain de performances : par exemple, si les tâches ne sont pas indépendantes elles ne pourront pas être exécutées en même temps par les processeurs. Outre l'indépendance des tâches, il faut s'assurer également que la répartition du travail parmi les cœurs est équilibrée. Si un cœur est trop chargé il devient le goulet d'étranglement de la puce : les autres cœurs finiront de traiter leurs tâches bien avant lui et passeront leur temps à l'attendre sans pouvoir contribuer à la complétion du programme. Cet équilibrage de charge ne doit d'ailleurs pas se limiter à la charge en travail des processeurs, mais doit prendre en compte toutes les ressources qui influent sur la performance (*ie.* bus de communications, caches, mémoire) en particulier dans un contexte embarqué où les capacités de la cible sont contraintes.

Description du parallélisme Traditionnelement, les programmes sont développés dans les langages de programmation impératifs classiques (p. ex. C ou Java) basés sur une architecture séquentielle de Von Neumann. Malheureusement paralléliser un tel programme est difficile car :

- Dans le modèle de Von Neumann les programmes sont une suite d'instructions, chaque instruction pouvant modifier l'état de la machine et entraîner des effets de bords nécessaires à la bonne exécution des instructions à venir. Pour paralléliser un tel programme, il nous faut analyser les interactions entre instructions pour identifier les éventuels blocs qui sont indépendants entre eux. Cette analyse est un problème difficile.
- Dans le modèle de Von Neumann les différents blocs d'instructions d'un programme s'échangent des données de façon arbitraire à travers une mémoire globale. Ceci rend difficile l'analyse des communications au sein d'un programme et donc leur prise en compte lors de la répartition des tâches parmi les différents cœurs.

Pour toutes ces raisons nous pensons que pour exploiter efficacement les architectures multicœurs, il faut abandonner les modèles de programmations basés sur une architecture Von Neumann, et se tourner vers des langages plus structurés qui facilitent l'extraction et l'adaptation du parallélisme. Dans le cas des programmes dirigés par les données, puisque les dépendances sont souvent

conditionnées par les flux de données au sein du programme, nous pensons qu'une bonne parallélisation passe par une description fine des opérations de routage.

Formalisme Avant de pouvoir étudier les différents moyens pour décrire du parallélisme, il faut définir un modèle sous-jacent nous permettant de raisonner sur l'exécution concurrente des programmes. Dans le contexte de l'embarqué, les applications doivent pouvoir tourner très longtemps sans intervention humaine, parfois dans des domaines critiques. Il est donc important de garantir le déterminisme et la prédictabilité de nos programmes ; nous nous limiterons donc au formalismes d'exécution parallèle qui garantissent une exécution déterministe des programmes.

Les *threads* sont le modèle classique d'exécution d'un programme parallèle. Le concepteur divise son code en différents blocs dont l'exécution est concurrente, la cohérence des échanges entre différents fils d'exécution est obtenue par l'utilisation de primitives de synchronisation (p. ex. sémaphores ou mutex). Dans un modèle de threads garantir le déterminisme est très difficile car il n'y a pas de moyen simple pour vérifier l'absence d'interblocages. Par ailleurs, le comportement du programme est lié à l'ordre d'exécution des threads ; ce qui peut créer des situations de compétition (*race conditions*) et rendre difficile la reproductibilité d'une erreur. Les problèmes de la programmation avec threads sont discutés en détail dans [Lee06]. Dans cette thèse nous ne nous intéressons pas au problème de la parallélisation avec threads, puisque de toutes façons ce modèle ne nous fournit pas les garanties de déterminisme suffisantes pour une exécution reproductible dans un contexte embarqué.

Jantsch sépare dans [JS05] les modèles adaptés aux systèmes embarqués non temps-réel en deux catégories :

- Les modèles à *rendez-vous* où les tâches sont modélisées par des processus séquentiels qui s'exécutent en concurrence. L'information n'est échangée entre tâches qu'à certains points de synchronisation des programmes, appelés *rendez-vous*. Les CSP (*Communicating Sequential Processes*) [Hoa78], les CCS (*Calculus of Communicating Systems*) [Mil80], le Π-calcul [MPW92] et les mécanismes de synchronisation basés sur les primitives `entry` et `accept` du langage Ada sont des exemples de modèles à rendez-vous.

- Les modèles de flots de données où les tâches sont modélisées par des processus séquentiels qui s'échangent des données entre eux à travers des canaux de communications. On peut ainsi représenter un programme comme un graphe dont les nœuds sont les processus et les arcs sont les canaux. Il existe de nombreux représentants du modèle de flots de données [Kah74, LP95, BELP95, KM66].

Contrairement aux flots de données, les modèles à *rendez-vous* sont construits autour du concept de synchronisation et ne modélisent pas explicitement la communication entre deux tâches. Ils sont adaptés aux applications concurrentes complexes mais ne capturent pas directement les flots de données au sein d'une application de traîtement du signal. C'est pourquoi nous avons choisi de travailler sur le modèle de flots de données qui nous assure un exécution déterministe et qui modélise explicitement les communications au sein d'une application.

1.2 Le modèle de flots de données

Il existe plusieurs formalismes (cf. figure 1.1) dans la famille des modèles de flots de données qui représentent différents compromis entre l'expressivité et les garanties offertes.

Le modèle le moins contraint, les Processus de Kahn (KPN, *Kahn Process Networks*) [Kah74], impose une règle simple : les processus communiquent uniquement à travers les canaux, chaque canal est muni d'une file (abstraite) où les données en transit sont stockées, l'écriture de données est asynchrone mais la lecture est bloquante. Ce modèle garantit le déterminisme d'exécution (les données produites par un KPN sont uniquement une fonction des entrées du KPN), mais ne

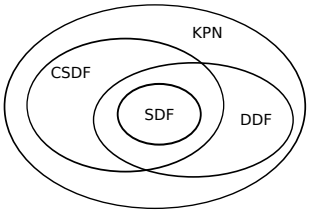


FIGURE 1.1 – Les différents formalismes de flots de données.

permet pas de détecter statiquement les interblocages ni de borner statiquement la taille maximale atteinte par les files des canaux durant l'exécution (c'est-à-dire la mémoire nécessaire pour garantir la cohérence des données).

Une restriction des KPN très intéressante est le modèle de *Synchronous Dataflow* (SDF)[LM87] où la quantité de données émise à chaque exécution d'un processus est fixe. Ce modèle est souvent utilisé dans la conception de systèmes embarqués et systèmes temps-réel parce qu'il garantit un certain nombre de propriétés intéressantes, en particulier :

- le déterminisme d'exécution, la détection d'interblocages et l'exécution en mémoire bornée qui nous permettent de garantir un comportement correct et reproductible de nos applications ;
- une expression naturelle de tâches concurrentes qui nous permet de détecter facilement le parallélisme propre à l'application et d'extraire les dépendances entre tâches ;
- la connaissance à la compilation du coût en mémoire et en communication pour chaque tâche qui nous permet d'estimer le coût en mémoire, communication et calcul et ainsi de faire un équilibrage de charge des tâches sur les différentes ressources disponibles sur la cible.

Ces garanties très fortes ont un prix : la quantité de données envoyés par un nœud est toujours fixe, elle ne peut pas dépendre des données. Il existe une extension des SDF qui conserve toutes les garanties précédemment cités tout en permettant un peu plus de flexibilité : les graphes *Cyclo-Static Dataflow* (CSDF) [BELP95]. Les CSDF imposent aux processus de fixer non plus une valeur de production, mais un nombre fini de valeurs qui constituent le cycle de ses productions. À l'exécution chaque acteur produira en tourniquet autant de valeurs qu'imposées par la production courante du cycle. Les CSDF permettent ainsi aux acteurs d'avoir des productions qui varient cycliquement, mais dont le cycle doit être, comme pour les SDF, connu à la compilation.

Les SDF et CSDF ne permettent pas de modéliser du contrôle au niveau des branchements du graphe. Néanmoins il est possible moyennant un certain nombre d'hypothèses de modéliser du contrôle de manière déterministe ; le lecteur intéressé pourra consulter les travaux de Buck[Buc93] sur les *Dynamic Dataflow* (DDF).

1.2.1 Différence avec les CDFG

Il ne faut pas confondre le formalisme CSDF avec les *Control Data Flow Graph* qui sont par exemple décrits dans [RCHP91]. Les CDFG sont une réprésentation qui modélise les dépendances de contrôle et de données entre les tâches d'un programme. Les tâches sont représentées par des nœuds qui sont connectés par des arcs de contrôles ou de données. Néanmoins dans les CDFG, chaque tâche n'est exécuté qu'une seule fois ; alors que dans les CSDF (ou SDF) il y a un flux continu de données dans le graphe : on peut voir cela comme une boucle implicite qui entourerait tout le graphe CSDF. Les techniques de compilation d'un CDFG et d'un CSDF sont donc différentes. Pour les CDFG chaque tâche ne peut-être exécutée qu'une seule fois ; on essayera donc de choisir un ordre d'exécution qui minimise le chemin critique dans le graphe. Pour les CSDF, on peut executer

les tâches plusieurs fois : on peut donc « pipeliner » leurs executions successives ; on s'intéressera donc plutôt à augmenter le débit de productions de données du graphe.

Nous avons choisi de construire notre modèle de programmation sur le formalisme de flots de données qui assure le déterminisme en mémoire borné indispensable dans le contexte de l'embarqué. Dans les sections qui suivent nous discuterons des solutions existantes pour décrire et paralléliser des applications dirigées par les données en utilisant des flots de données. Nous montrerons les avantages et les limites de ces approches pour la description et l'optimisation du routage de données, puis nous présenterons la problématique et les contributions de cette thèse.

1.3 Les premiers langages de flots de données

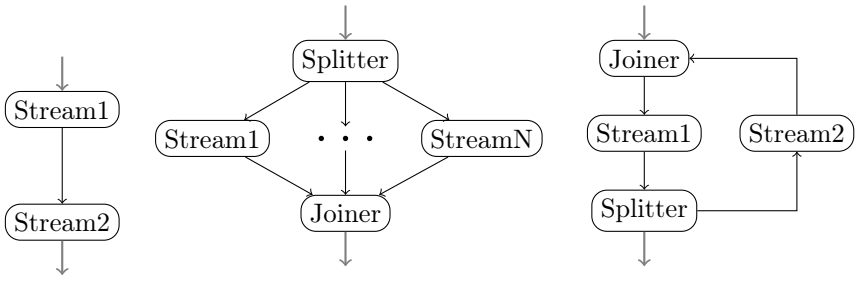
Un des premiers langages à considérer des flots de données, Lucid [AW77], s'appuie sur les flots pour raisonner mathématiquement sur l'exécution d'un programme. Dans Lucid les variables ne représentent pas une valeur en mémoire mais des séquences infinies de valeurs ; de la même manière les opérateurs classiques, comme `+`, sont étendus aux séquences infinies. Pour instancier ou assembler des séquences, Lucid introduit des primitives spéciales comme `first`, `next` ou `as soon as`. Ainsi le programme suivant :

```
first I = 0
next  I = I + 1
```

construit un flot I infini décrivant les entiers naturels. Chaque ligne dans un programme Lucid correspond à une équation entre différents flots de valeurs, ce qui permet de faire des preuves efficaces de correction. La description équationnelle du programme peut-être représenté comme un réseau de Kahn [AW77]. Un des successeurs plus récents de Lucid est SISAL [MSA+85] qui permet une description plus riche du contrôle (utilisation de boucles for par exemple). Le modèle de flots sous-jacent est préservé dans SISAL grâce à la propriété d'affectation unique (*Single Assignment*) : chaque variable ne peut être affectée qu'une seule fois à l'intérieur d'une fonction, ceci permet d'analyser facilement les dépendances entre variables qui peuvent être reformulées dans un formalisme de flots de données.

Les langages synchrones possèdent un formalisme proche de celui des flots de données. Ces langages modélisent explicitement la concurrence et la synchronicité (l'exécution du programme est cadencée par des réactions à des évènements ou à une horloge). Lustre et Signal utilisent une syntaxe equationelle proche de celle de Lucid et permettent de cadencer la production des éléments sur des flots de données avec des horloges logiques. Esterel adopte un style impératif pour la description de programme synchrones : différents acteurs sont exécutés en parallèle et communiquent en s'échangeant des signaux. Il faut tout de même faire attention : seuls les programmes synchrones déterministes peuvent être modélisés comme des flots de données. Or Esterel et Signal permettent l'écriture de programmes synchrones non déterministes[Hal98]. Actuellement, aucun de ces langages synchrones ne permet de programmer directement des architectures parallèles. Néanmoins, le successeur Shim [ET06], basé sur un modèle de réseaux de Kahn avec *rendez-vous*, peut être compilé vers un modèle de *threads* executable sur des multiprocesseurs classiques [EVT08]. Même si les langages synchrones sont proches du modèle de flots de données, ils sont surtout adaptés à la modélisation de systèmes réactifs temps-réel ; alors que les flots de données présupposent des communications régulières entre consommateur et producteur, les langages synchrones permettent des patrons de communications complexes en réponse à un événement. Dans [BCE+03] on trouvera une discussion plus approfondie des langages synchrones.

Aujourd'hui, les nouvelles architectures parallèles ont suscité un regain d'intérêt pour les langages de flots de données et leur facilité à exprimer la concurrence. De nombreux langages flots de données ont été proposés pour paralléliser des applications. Nous allons présenter les plus aboutis, puis montrer quelles sont les différences entre eux, leurs forces et leurs faiblesses.



(a) Pipeline (b) Split-Join (c) Feedback Loop

FIGURE 1.2 – Les différentes primitives de StreamIt pour assembler des filtres.

1.4 Langages de Flots pour le parallélisme

1.4.1 StreamIt

StreamIt[AGK+05] est un langage de flots de données et un compilateur optimisant. Le compilateur cible l'architecture RAW et les architectures SMP avec cache. Un simulateur fonctionnel Java est également disponible.

StreamIt propose de nombreuses transformations permettant d'ajuster au mieux la granularité d'un programme parallèle à l'architecture cible, en exploitant différents types de parallélisme : de données, de tâche et de pipeline.

StreamIt s'articule autour de la notion de filtre. Un filtre prend un seul flux en entrée et produit un seul flux en sortie après avoir effectué un calcul sur les données. Le type de données d'un flux est paramétrable, par contre les flux sont considérés infinis et unidimensionnels.

Les filtres StreamIt sont assemblés dans un graphe de flots à l'aide d'un ensemble de connecteurs : les tuyaux (pipes) qui permettent de connecter deux sous-graphes, le split-join qui permet de distribuer puis de réunir un flot sur plusieurs sous-graphes, et les boucles de retour (feedback loops) qui permettent d'avoir des cycles. Ces connecteurs sont représentés sur la figure 1.2. L'utilisation de ces squelettes contraint la structure des graphes StreamIt qui ont une forme hiérarchique où les filtres forment un réseau série-parallèle. La structure hiérarchique des graphes permet une description textuelle plus simple du graphe, où l'on décrit l'emboîtement des différentes primitives et des filtres. Par exemple, sur l'extrait de code 1.1, la multiplication de deux matrices est exprimée à l'aide des connecteurs StreamIt.

Les taux de consommation et de production sont fixés par le concepteur. En effet les graphes StreamIt s'appuient sur le modèle CSDF pour dimensionner et ordonnancer le système. Le concepteur doit donc pour chaque filtre préciser le nombre d'éléments consommés et produits par exécution. Il existe un troisième mode d'accès aux données offert par StreamIt : le `peek`. Un filtre StreamIt peut ainsi consulter (sans consommer) un certain nombre de données en avance sur le canal entrant : cela est particulièrement pratique pour écrire des filtres qui travaillent sur une fenêtre glissante de données. On évite de devoir confiner à l'intérieur du filtre l'état interne de la fenêtre glissante ce qui cacherait le parallélisme de données. L'utilisation de `peek` est compatible avec le modèle CSDF à condition de précharger les données lues en avance pour chaque filtre.

StreamIt propose un certain nombre de transformations permettant d'obtenir une granularité de parallélisme adaptée à l'architecture. On peut les distinguer en trois groupes :

- **Les transformations de Fusion** sont utiles lorsque l'on veut réduire le parallélisme existant ; elles regroupent des filtres adjacents. Ces transformations sont intéressantes pour des filtres légers : en regroupant ces filtres on obtient un filtre plus gros pouvant amortir les

sur-coûts d'activation et synchronisation.

- **Les transformations de Fission**, au contraire, permettent d'augmenter le parallélisme d'une application lorsque celui-ci n'est pas suffisant pour occuper tous les processeurs de l'architecture. Elles divisent un filtre en deux ou plusieurs filtres plus petits, qui peuvent être alors répartis sur des processeurs différents.
- **Les transformations de Réorganisation**, changent l'ordre et le routage des données au sein de l'application pour optimiser les communications entre les différents filtres.

Pour choisir quelles transformations appliquer, le compilateur StreamIt utilise un modèle de coût qui prend en compte la répartition du temps de calcul, des transferts de données et du coût de synchronisation sur RAW. L'espace de transformation est exploré avec la technique de recuit simulé.

Des transformations complémentaires ont également été proposées. Pour les filtres linéaires, StreamIt propose des transformations qui exploitent des simplifications algébriques entre des filtres linéaires consécutifs. Pour les architectures avec cache, StreamIt propose des transformations de fusion qui prennent en compte la taille des caches d'instruction et de données.

Plus récemment une nouvelle primitive de communication, les *teleport messages*, a été incorporée au langage pour permettre l'expression de contrôle dynamique dans certains programmes.

1.4.2 StreamC et KernelC

StreamC et KernelC sont les deux composantes d'un langage de programmation pour Imagine[KDK+01], une architecture spécialisée dans le traitement de flots de données. Imagine assemble plusieurs unités de calcul autour d'un SRF (Stream Register File). Le SRF est une unité mémoire partagée composée de registres permettant d'échanger des données entres les unités de calcul.

Pour programmer la cible Imagine, le concepteur décrit séparément le code pour faire des calculs et le code pour assembler des Streams :

- KernelC est le langage utilisé pour l'écriture des noyaux de calcul. C'est un langage avec une syntaxe semblable au C muni d'extensions permettant d'exprimer du parallélisme vectoriel (SIMD).
- StreamC est le langage utilisé pour décrire l'assemblage des streams à travers lesquels les noyaux s'échangent des données.

Le but de cette séparation est d'isoler dans le code StreamC toutes les opérations de contrôle et de communication ; ce qui facilite l'analyse et la compilation sur Imagine. Pour éviter que du code de contrôle ou de communication se retrouve à l'intérieur des noyaux de calcul, le langage KernelC n'autorise qu'une seule structure simple de contrôle, les boucles, et les seuls accès permis à la mémoire globale passent à travers des streams.

Les données échangées dans les streams sont typées : types scalaires ou types *record*. Les streams sont unidimensionnels et de taille finie. StreamC introduit la notion de stream dérivé ; si on ne s'intéresse qu'à certains éléments à l'intérieur d'un stream, on peut définir un stream dérivé qui ne contiendra que les éléments en question. Pour les sélectionner, on pourra utiliser un vecteur de déplacement (*stride*) s'ils sont régulièrement espacés ou alors passer une liste contenant les positions des éléments qui nous intéressent.

Pour compiler efficacement sur Imagine, plusieurs optimisations sont proposées. En particulier, la latence est réduite en superposant les calculs et les accès mémoire. Du strip-mining des boucles des noyaux est réalisé pour faire tenir les streams échangés entre producteurs et consommateurs à l'intérieur du SRF (et éviter des recopies en mémoire globale plus coûteux). Il faut noter que l'ordonnancement réalisé est à un niveau de granularité très fin, celui de l'opération élémentaire arithmétique.

```
float->float pipeline
MatrixMultiply (int x0, int y0, int x1, int y1) {
  add RearrangeDuplicateBoth(x0, y0, x1, y1);
  add MultiplyAccParallel(x0, x0);
}
float->float splitjoin
RearrangeDuplicateBoth (int x0, int y0, int x1, int y1) {
  split roundrobin(x0 * y0, x1 * y1);
  // on duplique les lignes de la premiere matrice
  add DuplicateRows(x1, x0);

  // on transpose et duplique la deuxieme matrice
  add RearrangeDuplicate(x0, y0, x1, y1);
  join roundrobin;
}
float->float pipeline
RearrangeDuplicate(int x0, int y0, int x1, int y1) {
  add Transpose(x1, y1);
  add DuplicateRows(y0, x1*y1);
}
float->float splitjoin
Transpose(int x, int y) {
  split roundrobin;
  for (int i = 0; i < x; i++) add Identity<float>();
  join roundrobin(y);
}
float->float pipeline
DuplicateRows(int times, int length) {
  split duplicate;
  for (int i = 0; i < times; i++) add Identity<float>();
  join roundrobin(length);
}
float->float splitjoin MultiplyAccParallel(int x, int n) {
  split roundrobin(x*2);
  for (int i = 0; i < n; i++) add MultiplyAcc(x);
  join roundrobin(1);
}
float->float filter MultiplyAcc(int rowLength) {
  work pop rowLength*2 push 1 {
    float result = 0;
    for (int x = 0; x < rowLength; x++) {
      result += (peek(0) * peek(1));
      pop();
      pop();
    }
    push(result);
  }
}
```

Extrait de code 1.1 – Multiplication de matrices en StreamIt (extrait du fichier source `matrix.str` distribué avec StreamIt)

1.4.3 Cg

Cg[MGAK03] est un langage de programmation dérivé du C pour programmer des processeurs graphiques de manière portable. Traditionnellement les processeurs graphiques possédaient un pipeline à plusieurs étages parmis lequels on trouve deux étages programmables : le *Vertex processor* et le *Pixel (ou Fragment) processor*. Cg est un langage spécialisé en la programmation de ces deux étages. Contrairement à StreamIt ou à StreamC, dans Cg l'interconnexion entre tâches est implicite ; en effet les graphes Cg sont limités à deux tâches : une tâche qui s'exécute sur le *Vertex processor* et qui alimente une deuxième tâche s'exécutant sur le *Pixel processor*.

Le langage Cg permet de décrire facilement les noyaux de ces deux tâches. Il présente un ensemble d'opérateurs haut-niveau qui sont transformés dans des opérations vectorielles par le compilateur. Le passage des données entre les deux étages est également pris en charge par le compilateur.

1.4.4 Brook

Brook est un langage de programmation de flots qui peut-être compilé sur des architectures généralistes (Merrimac, Imagine) mais qui est surtout utilisé pour programmer des accélérateurs graphiques. Il est intéressant de noter que Brook était déjà capable de compiler des programmes sur accélérateurs parallèles avant la popularisation des GPU et des librairies spécialisées (*e.g.* CUDA) par l'utilisation de Shaders à travers OpenGL ou DirectX.

La syntaxe de Brook est inspirée du C, mais propose de nombreuses extensions pour travailler sur des streams. Les streams sont typés et de dimension arbitraire. Les types pour les streams peuvent être librement définis par le concepteur [1]. Néanmoins les streams sont de taille finie qui est spécifiée par une *shape* qui indique la taille de chaque dimension .

Les noyaux de calcul sont des fonctions C précédées du mot clé `kernel` et pouvant recevoir des paramètres de type stream. Ces derniers sont au choix en lecture seule, en écriture seule, ou en lecture seule par accès arbitraire (on peut lire les élements dans le désordre). Un stream peut être utilisé dans le même noyau en lecture et écriture à condition que l'accès aux données soit régulier. Le fait de distinguer les accès réguliers et les accès aléatoires permet d'optimiser la gestion de la mémoire ; en effet les streams à accès aléatoire seront à priori stockés dans une zone de cache (puisque les données vont être lues plusieurs fois) alors que les streams à lecture régulière seront stockés dans une zone de mémoire transiente. Bien sûr et comme pour les autres langages, les effets de bord entre noyaux doivent être strictement limités aux communications par stream, sous peine de violer la sémantique lors de la parallélisation des noyaux.

Lorsque les tailles des streams en entrée ne sont pas compatibles, le compilateur va automatiquement ajuster la taille des streams (en dupliquant ou décimant des éléments).

Une des possibilités intéressantes de Brook est d'introduire des noyaux qui font une réduction sur les éléments des streams en entrée (la même opération est appliquée récursivement sur tous les éléments). Pour que la sémantique de la réduction ait un sens, les noyaux de réduction doivent être associatifs (cela n'est pas vérifié par le compilateur).

Finalement, Brook propose un ensemble de fonctions réordonnant les élements dans un stream, par exemple :

- `streamStencil` qui permet d'extraire des blocs de données d'un stream en déplaçant un patron ou *stencil* sur l'espace des données. Par exemple pour extraire des noyaux lors d'une convolution ;
- `streamStride` qui permet de sélectionner des éléments régulièrement espacés à l'intérieur d'un stream ;

1. Cependant, l'implémentation actuelle de Brook est limitée aux types primitifs (pas de types tableau, ou de types composites.)

- `streamRepeat` qui permet de dupliquer des éléments dans un stream ;
- `streamMerge` qui permet de mélanger les éléments de plusieurs streams.

Le compilateur Brook pour GPU est un compilateur source vers source : il ne génère pas directement du code pour les accélérateurs graphiques, mais génère du code Cg. La communication des streams a lieu à travers les mémoires de texture du GPU.

Un ensemble de transformations[LDWL06] basées sur une représentation des dépendances sur les stream dans le modèle polyédrique a été proposé pour optimiser les programmes Brook. On en discutera plus en détail en § 3.8.

1.4.5 ArrayOL

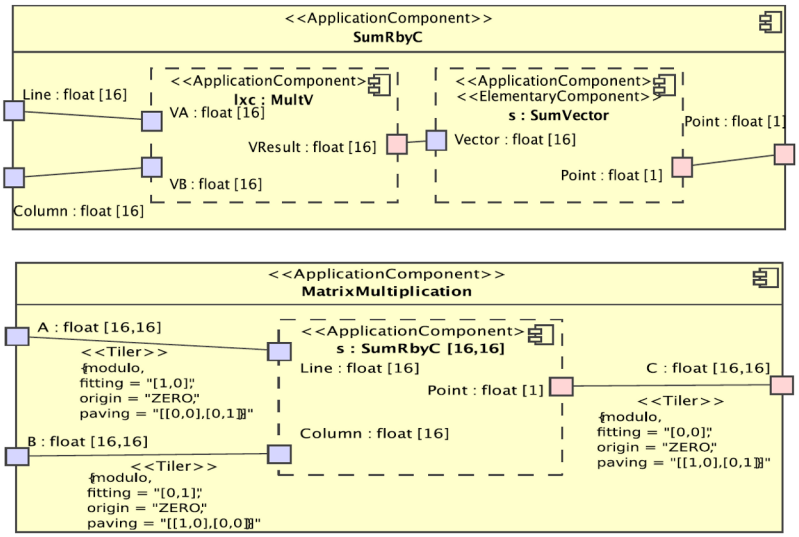


FIGURE 1.3 – La multiplication de matrices dans le modèle ArrayOL dans l'environment de développement Gaspard2. La figure a été empruntée à l'article [CGH+08].

ArrayOL [Bou07] est un langage spécialisé dans le traitement du signal. Les données sont représentées par des tableaux multidimensionnels, dont une des dimensions peut-être infinie (ce qui permet de représenter le temps par exemple). Les tableaux sont toriques : en se déplaçant le long d'une dimension on revient au début du tableau. Ces caractéristiques permettent une expression simple de certains algorithmes de traitement du signal (comme le traitement de données sonar).

Dans ArrayOL les programmes sont composés de filtres qui manipulent les tableaux de données et qui s'échangeant des tableaux à travers des flots ; la descript ion du programme se fait à deux niveaux :

- *le niveau global* décrit à l'aide d'un graphe acyclique orienté les interconnexions entre filtres. Un filtre peut prendre plusieurs flots en entrée. L'absence de cycles dans le graphe interdit les boucles de retour dans les calculs mais simplifie l'ordonnancement. La précédence des tâches dans le graphe orienté introduit un ordre partiel d'exécution en laissant toute liberté au compilateur pour choisir l'ordonnancement le plus adapté.
- *le niveau local* décrit les dépendances entre les entrées et les sorties d'une tâche. À chaque entrée est associé un *tiler* : un bloc qui décrit un sens de parcours des données. Un tiler

Langage	*Dim.*	*Graphe*	*Statique*	*Réorganisations*
StreamIt	1	hiérarchique	oui (sauf téléport)	connecteurs (p. ex. `split/join`)
KernelC/StreamC	1	arbitraire	non	primitives (p. ex. `strided`)
ArrayOL	n	acyclique	oui	DSL
Brook	n	arbitraire	non	primitives (p. ex. `StreamStencil`)
Cg	1	2 étages	oui	non
Block Parallel	2	acyclique	oui	DSL

TABLE 1.1 – Expression du routage de données dans les langages de flots de données.

est composé d'un point d'origine, d'une forme de motif, d'une matrice de pavage et d'une matrice d'ajustage. À chaque exécution, la tâche va consommer un motif, c'est-à-dire un sous ensemble d'éléments dans le tableau en entrée, puis le motif va être déplacé avant l'exécution suivante. Le déplacement du motif dépend du type de recouvrement, celui-ci est déterminé par le point d'origine et les matrices de pavage et d'ajustage. De la même manière que l'on associe un tiler à chaque entrée pour décrire un parcours de lecture, on associe à chaque sortie un tiler pour décrire un parcours en écriture. Les dépendances de données sont complètement déterminées par la correspondance des parcours en entrée avec les parcours en sortie.

Les travaux les plus récents pour compiler des programmes ArrayOL, font une passe d'optimisations où certaines tâches ArrayOL sont fusionnées pour obtenir du parallélisme à gros grain et factoriser les dépendances entre deux filtres successifs pour avoir un pipeline efficace. Puis le graphe ArrayOL optimisé est ensuite transformé dans un KPN, ce qui permet une exécution concurrente des tâches. [Sou01, ABD05].

Les programmes ArrayOL sont développés dans un environnement de développement visuel appelé Gaspard[DBDM02] qui permet de visualiser les différents filtres, les inter-connexions au niveau global, et les matrices d'ajustage, de pavage et d'origine, au niveau local. Sur la figure 1.3 on a repris la multiplication matricielle telle qu'implémentée dans Gaspard, sur les entrées A et B, les matrices de pavage et d'ajustage permettent d'extraire respectivement les lignes et les colonnes.

1.4.6 Block Parallel

Block Parallel[BS08] est un langage proposé par Black-Schaffer pour l'expression d'applications de traîtement d'images. L'auteur constate que les expressions multidimensionnelles proposées, par exemple par ArrayOL, sont difficiles à optimiser car chaque nouvelle dimension augmente le nombre de parcours possibles des données qu'il faut prendre en compte. Un compromis est donc fait entre expressivité et facilité d'optimisation : Block Parallel se restreint aux graphes de flots de données acycliques sur des espaces bi-dimensionnels. Le modèle de programmation, basé sur un graphe de flots formé de filtres, combine différents idiomes que l'on a vu précédemment.

La réorganisation des données consommées par les filtres est exprimée à l'aide de nœuds *Input* et *Output* qui spécifient la taille des blocs de données traités (les blocs doivent être rectangulaires) et l'espacement entre les blocs consécutifs. Le parallélisme dans l'application est décrit à l'aide de *splitters* et *joiners* comme dans StreamIt. Les dépendances spécifiées à l'aide des nœuds *Input* et *Output* sont propagées dans le graphe de manière à trouver un parcours des données qui favorise la réutilisation de données par des filtres consécutifs.

Le langage Block Parallel est cependant limité car il ne permet pas d'exprimer[BS08] une multiplication matricielle ou une FFT. En effet ces applications consomment les données de manière trop irrégulière, et ne peuvent être exprimées facilement avec les nœuds *Input* et *Output*, ce qui rend difficile la propagation des dépendances de données dans le graphe.

Langage	*Optimisations*	*Mémoire bornée*	*Ordonnancement*
StreamIT	Trans. de graphe(p. ex. fission, fusion, etc.), filtres linéaires	oui	Statique(CSDF)
KernelC	Trans. de boucles (p. ex. strip mining), Vectorisation	non	Dynamique
ArrayOL	Trans. graphe (p. ex. fusion, repavage)	oui	Statique(KPN)
Brook	Trans. polyédriques	non	Dynamique
Cg	Vectorisation	non	Dynamique
Block Parallel	Fission, Combine dépendances	oui(vérif. hors ligne)	Dynamique

TABLE 1.2 – Optimisation et exécution dans les langages de flots de données.

1.4.7 Expressivité de ces langages

Tous les langages présentés ci-dessus ont le modèle de flots de données en commun. On a comparé l'expressivité de ces langages selon certains critères sur la table 1.1. On peut remarquer qu'à l'exception de Cg, tous séparent la description du graphe des flots et la description des noyaux de calcul. Dans le cas de Cg, le graphe de flots est implicitement un pipeline à deux étages qui mime le pipeline matériel offert par le GPU, il n'y a donc nul besoin de le décrire. Pour les autres langages cette séparation semble raisonnable puisqu'elle permet d'extraire facilement les dépendances entre noyaux de calculs, nécessaires avant toute parallélisation. Dans cette thèse nous ne nous intéressons pas à l'expression des noyaux de calcul, nous nous limitons donc à une comparaison sur l'expression des flots.

Forme et type des flots Un premier critère est le type et la forme des élements échangés sur les flux. La plupart des langages permettent d'échanger des données typées, bien que la taille des types doit être connue à la compilation (les tableaux à taille dynamique sont donc proscrits). Ainsi l'utilisateur peut définir ses propres types record et échanger entre tâches des données structurées complexes.

StreamIt et KernelC considèrent que les flots n'ont pas de forme : ce sont des suites unidimensionnelles de données. Cela complique l'expression d'algorithmes de traitement d'image ou de vidéo, en effet on ne peut pas exprimer les dépendances de données en 2 dimensions : on est obligé de projeter les dépendances sur un modèle unidimensionnel, ce qui entraîne des calculs d'index fastidieux. ArrayOL et Brook permettent de déclarer des flux avec un nombre de dimensions arbitraire, ce qui autorise une plus grande flexibilité lors de l'écriture de programmes qui travaillent sur des données multidimensionneles. Block Parallel implémente des flots de dimensions 1 et 2.

Structure du graphe Un deuxième critère est les restrictions sur le graphe d'interconnexion des filtres. Cg est bien sûr le plus contraint, puisqu'il ne permet que des pipelines à deux niveaux ; l'avantage est que le mapping du graphe sur l'architecture est trivial, puisqu'il y a une bijection entre le graphe de filtres et la topologie.

StreamIt impose une structure de graphe hiérarchique. Selon ses auteurs cette structure rendrait plus simple la description du graphe. Il est vrai que la structure série-parallèle imposée permet de décrire le graphe comme un emboîtement de primitives de communication, ce qui est facilement transposable sous une forme textuelle. Cependant cette structure impose des restrictions sur l'expression des réorganisations de données. Certains types de réorganisations comme l'inversion de l'ordre de parcours d'un vecteur sont impossible à exprimer sans passer par un filtre dédié (ce qui prive cette réorganisation des optimisations du compilateur). D'autres, comme le parcours des

éléments dans une FFT Butterfly, imposent l'utilisation de cascades de Split et Join peu naturelles.

Finalement ArrayOL et Block Parallel interdisent les cycles dans la composition des filtres. Ceci rend l'analyse du programme plus simple mais interdit l'expression des programmes avec une boucle de retour. On peut contourner le problème en exprimant les boucles de retour à l'intérieur d'un filtre ; le désavantage est que l'on ne peut plus exploiter le parallélisme potentiel dans la boucle de retour.

Expression des réorganisations de données Comment exprimer des réorganisations de données, par exemple une transposition ou une décimation, dans ces langages ? On a identifié trois approches : soit le langage ne permet pas de modéliser les réorganisations complexes de données, celles-ci sont donc faites par des noyaux de calcul et ne bénéficient pas d'optimisations spécifiques ; soit le langage propose un ensemble de primitives de transformation qui peuvent être assemblées pour créer des réorganisations plus complexes ; soit le langage propose un langage dédié (Domain Specific Language) pour l'écriture de réorganisations de données.

KernelC/StreamC et Cg ne permettent pas l'expression de réorganisations complexes, comme les transpositions, il faut passer par un noyau de calcul. Dans ces langages on ne peut exprimer que du routage simple de données entre filtres, mais il n'y a pas de mécanisme spécifique pour exprimer les arrangements de données.

StreamIt et Brook appartiennent à la deuxième famille : ils utilisent une bibliothèque de primitives. StreamIt propose trois primitives qui permettent de faire de la réorganisation de données : Split Round Robin, Join Round Robin, Duplicate Round Robin. Par exemple pour écrire une transposition en StreamIt il suffit d'assembler un Split et un Join aux productions bien choisies. Brook propose également des primitives mais elles prennent la forme de fonctions paramétrables. Ainsi la fonction `StreamStencil` permet d'extraire successivement les colonnes, ce qui revient à faire une transposition de la matrice.

ArrayOL et Block Parallel définissent des blocs de restructuration de données. La transformation associée à chaque bloc est spécifiés en utilisant un langage dédié. Cette dernière famille offre de notre point de vue la plus grande facilité et flexibilité pour l'expression de réorganisation de données.

Productions statiques/dynamiques Pour StreamIt, ArrayOL, Block Parallel, les productions et consommations des filtres doivent être connues et donc fixées à la compilation. KernelC/StreamC, Brook et Cg autorisent des productions et consommations dynamiques. Le deuxième groupe de langages permet donc une plus grande flexibilité lors de l'écriture de programmes. Cependant le fait de travailler avec des flots de taille inconnue ne permet pas de dimensionner l'application à la compilation, ce qui peut-être problématique dans le contexte de l'embarqué où les ressources mémoires sont limitées.

1.4.8 Compilation de ces langages

Exécution en mémoire bornée sans interblocages En observant la table 1.2 un lien apparaît entre l'ordonnançabilité en mémoire bornée des programmes et le type d'ordonnancement. Ainsi, à l'exception de Block Parallel, les langages qui sont ordonnancés statiquement permettent de calculer la mémoire nécessaire statiquement et ceux qui sont ordonnancés dynamiquement ne le sont pas. Ceci se comprend bien car l'ordre d'exécution détermine la quantité de mémoire consommée par les échanges entre filtres. Les ordonnanceurs dynamiques gagnent en flexibilité en différant le choix sur l'ordre des tâches jusqu'à l'exécution, au prix de ne plus pouvoir assurer une exécution en mémoire bornée. Block Parallel constitue une exception : il ordonnance les filtres dynamiquement, mais s'assure qu'ils peuvent être exécutées en mémoire bornée et sans interblocage pendant la compilation (il utilise une technique similaire à celle employée pour prouver la consistance d'un SDF, cf. § 2.4.1).

Nous notons également que les langages permettant une execution en mémoire bornée sont ceux dont les productions et les consommations sont connues et fixées. Les langages dont les productions sont connues permettent également de détecter les interblocages statiquement.

Optimisations du compilateur Les langages dynamiques proposent quelques optimisations mais celles-ci sont limitées puisque beaucoup sont faites en ligne ; leur coût de mise en place doit donc être faible. KernelC/StreamC propose néanmoins une transformation de boucles hors pour faire tenir les noyaux importants dans le SRF. Cette transformation est en fait une variante du *Strip Mining* appliquée aux filtres de KernelC : elle permet de fractionner l'exécution d'un kernel en plusieurs exécutions qui travaillent chacune sur une partie différente du flot. Cette transformation est intéressante lorsque plusieurs filtres se succèdent sur un même pipeline : en diminuant la taille des échantillons produits, les échanges sur le pipeline se font à travers le SRF sans le sur-coût d'un passage par la mémoire globale.

Les langages statiques pour lesquels les dépendances de données sont connues à l'exécution permettent des transformations plus riches qui prennent en compte ces informations supplémentaires. ArrayOL propose des transformations de fusion permettant de factoriser les dépendances de deux filtres sur un même pipeline. Dans certains cas ArrayOL est capable de "repaver", c'est-à-dire de changer l'ordre de parcours d'un tableau lorsque cela facilite la fusion de deux filtres. Les transformations d'ArrayOL sont très prometteuses ; néanmoins, leur efficacité dans une chaîne de compilation n'a pas été montrée expérimentalement.

StreamIt au contraire propose de nombreuses transformations au sein d'un compilateur qui affiche des gains en performances très intéressants sur RAW mais aussi sur des architectures SMP avec cache.

1.5 Problématique

Compte tenu de l'analyse de l'état de l'art de la précédente section, nous pouvons maintenant reformuler plus précisément notre problématique et faire apparaître les contributions de cette thèse.

La problématique de cette thèse s'articule autour de trois objectifs :

Limites d'expression StreamIt propose une méthode de compilation efficace, et affiche des performances remarquables sur l'architecture RAW. Cependant, en se limitant aux graphes série-parallèle, il interdit l'expression d'un certain nombre de patrons de communications ; comme par exemple l'inversion d'un tableau de données. De plus comme nous avons vu sur l'exemple précedent de la multiplication matricielle par blocs ; l'expression en StreamIt peut-être très verbeuse et peu lisible pour le non spécialiste. Une représentation plus haut-niveau, multidimensionnelle, comme par exemple celle proposée par ArrayOL, est bien plus accessible.

Ce qui nous amène au premier objectif de cette thèse : **Combiner l'expressivité d'un langage de haut-niveau (p. ex. ArrayOL) avec des optimisations de flots de données efficaces (p. ex. StreamIt).**

Adaptation à l'architecture Pour un même programme il existe de nombreuses implémentations différentes. Chaque implémentation affiche des communications et des réorganisations de données différentes qui conduisent toutes au même résultat final.

Lors de la compilation, nous cherchons l'implémentation qui affiche les meilleures performances, mais nous sommes contraints par les limites matérielles de la cible. Nous ne pouvons pas par exemple dépasser la mémoire disponible sur chaque processeur.

Puisque nous avons fait le choix de séparer l'expression du parallélisme de son implémentation ; il nous faut proposer une méthode automatique pour adapter le programme à la cible, c'est-à-dire choisir la "meilleure" implémentation. Puisque nous considérons les programmes dirigés par les

données, dans cette thèse nous étudions l'adaptation des réorganisations de données en fonction de critères architecturaux (p. ex. la mémoire disponible) avec des objectifs pour améliorer les performances (p. ex. réduction des communications).

En bref, nous souhaitons **proposer une méthode d'exploration de l'espace d'implémentation guidée par les contraintes architecturales et par des critères d'optimisation.**

Mais avant de pouvoir explorer l'espace d'implémentation il faut d'abord le connaître (ou en tout cas une partie). Pour générer les différentes implémentations équivalentes à partir d'un programme nous allons considérer un ensemble de transformations qui préservent la sémantique et les appliquer successivement pour générer des programmes sémantiquement équivalents mais correspondant à des implémentations différentes.

Étendre le modèle de Transformations StreamIt, Array-Ol, StreamC et Blockparallel proposent différentes optimisations du graphe de flots. Dans StreamIt, les transformations de graphe ne sont définies que dans le cadre des graphe série-parallèle. Pourtant dans certains cas, il est possible de trouver des implémentations plus simples à condition de considérer des graphes arbitraires. On verra, par exemple qu'il est possible d'optimiser les communications au sein d'une FFT Butterfly en considérant des graphes arbitraires (cf. figure 4.25 en § 4.4).

De la même manière, les transformations de StreamIt et d'ArrayOL ne considèrent pas des transformations sur des cycles ; et ne peuvent donc pas paralléliser certaines boucles de retour dont les itérations sont indépendantes. C'est le cas par exemple du filtre de Hough (cf. figure 4.15 en § 4.3.4). StreamC définit des transformations simples (tiling) au niveau des noyaux de calcul, mais ne s'intéresse pas aux réorganisations de données.

Dans cette thèse nous souhaitons étendre la méthode d'optimisation par transformation de graphes au graphes arbitraires avec cycles. Le fait de traiter les cycles soulève une nouvelle difficulté. En effet lorsque l'on transforme des cycles il est possible d'introduire des interblocages, puisque l'on est amené à changer l'ordre de traîtement des données au sein du cycle. Il nous faut donc garantir formellement que les transformations, non seulement préservent la sémantique du graphe mais n'introduisent pas d'interblocages. Pour cela nous introduirons un critère formel de légalité sur les transformations de graphes CSDF.

Ce qui nous amène au troisième et dernier objectif de cette thèse : **Étudier les transformations sur les réorganisations de données de stream pour des graphes arbitraires avec boucles.**

1.6 Contributions

Nous proposons de distinguer les opérations de réorganisation/routage de données et la description des opérations de calcul sur les données. Les opérations de calculs seront décrites dans un langage familier au concepteur comme C, alors que les opérations de réorganisation/routage seront décrites dans un langage spécialisé qui permet d'extraite facilement les dépendances entre tâches et donc de faire un découpage du programme en tâches parallèles. Pour la description des réorganisations de données, une analyse de l'existant, révèle deux types d'approches :

- soit une bibliothèque de primitives à composer comme les splitters, joiners de StreamIt ;
- soit un DSL spécialisé comme les tilers d'ArrayOL.

L'approche DSL brille lors de l'expression de réorganisations de données complexes (comme le parcours des données tableau lors d'une convolution). Un DSL multidimensionnel est d'ailleurs bien mieux adapté pour les application de traitement du signal qui considèrent des domaines à plusieurs dimensions, puisque l'expression est proche du modèle dans lequel le développeur conçoit et spécifie ses besoins.

Cependant l'approche DSL multidimensionnelle a deux inconvénients :

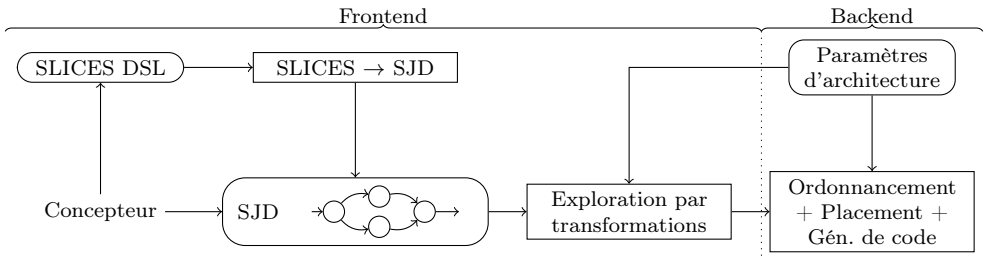


FIGURE 1.4 – Architecture de compilation proposée

- L'analyse d'un modèle multidimensionnel de flots est bien plus complexe qu'en une dimension comme le souligne Black-Schaffer[BS08], ce qui rend difficile les optimisations du compilateur. C'est pour cette raison que Block Parallel se limite à la dimension 2, de manière à faire un compromis entre puissance d'expression et capacité d'analyse.

- L'approche par primitives, plus bas niveau, peut-être utile pour le spécialiste qui veut avoir le contrôle complet sur les interconnexions entre filtres.

Nous proposons donc pour notre langage d'adopter une approche hybride avec deux niveaux de langages. Un premier langage que l'on appellera SJD, de bas niveau, unidimensionnel ; permettra au programmeur spécialiste de décrire lui même les réseaux interconnexions entre filtres et sera le langage sur lequel les optimisations du compilateur seront faites. Un deuxièmes langage, SLICES, de haut-niveau permettra de décrire le routage de données entre filtres de façon simple. Pour combiner les deux langages nous montrerons qu'il est possible de compiler le langage SLICES vers le langage SJD. La chaîne de compilation proposée est représenté en figure 1.4.

Dans un deuxième temps nous nous intéressons au problème de l'adaptation du parallélisme, et plus particulièrement au problème de l'optimisation des opérations de réorganisation de données. Nous proposons un ensemble de transformations qui permettent d'explorer différentes implémentations du parallélisme et du routage de données et nous choisissons celle qui convient le mieux aux contraintes de notre cible par un ensemble de métriques.

Pour faire cela nous introduisons tout d'abord un formalisme pour décider de la légalité d'une transformation : préserve-t-elle les traces du programme sans introduire des interblocages ou des inconsistances ? Puis nous présentons un ensemble de transformations légales : certaines sont les transformations proposées par StreamIt généralisées au cas des graphes non hiérarchiques avec cycles ; d'autres sont originales. Nous montrons que cet ensemble de transformations engendre un espace d'implémentation fini qui peut être exploré exhaustivement ou par une heuristique.

Finalement pour valider expérimentalement cette approche, nous nous intéressons à deux problème différents : la réduction de l'empreinte mémoire d'une application parallèle et la réduction des communications entre processeurs. Nous montront expérimentalement que la réduction des communications permet d'améliorer les performances (le débit du programme) lorsque celui-ci est borné par les communications.

Le langage SLICES a été décrit dans [dOCLB10a]. L'ensemble des transformations sur des graphes SJD ainsi que leur application à la réduction mémoire sur multiprocesseur ont été présentés dans [dOCLB10b] et [dOCLB09]. Enfin, l'utilisation des transformations pour réduire la communication entre processeurs a été présentée dans[dOCLB10c].

Pour résumer, les contributions de cette thèse sont :

- Chapitre 2 :

  – Un langage de flots bas niveau, SJD, basé sur des nœuds Split, Join et Dup inspiré de StreamIt mais plus expressif.

  - Un nouveau langage haut-niveau multidimensionnel, SLICES, permettant de décrire des réorganisations de données de manière plus simple et accessible.
  - Une approche hybride permettant de combiner ces deux niveaux de langages.

- Chapitre 3 :
  - Un modèle formel pour l'analyse de transformations sémantiquement valables dans les CSDF.
  - Un ensemble de transformations sur le langage de flots SJD que produit notre front-end permettant de générer différentes impléméntations d'un même programme.
  - Une méthode d'exploration efficace qui permet de choisir automatiquement quelles transformations il faut appliquer pour optimiser le programme.

- Chapitre 4 :
  - L'implémentation d'un backend pour SJD.
  - Une validation expérimentale de notre approche sur les problématiques de réduction de la mémoire et du coût des communications.

Chapitre 2

Expression du routage

Sommaire

Pour exprimer le routage et la réorganisation de données statiques, nous proposons deux langages complémentaires. Le premier langage, SJD, est un langage basé sur des graphes composés de Filtres, Splitters et Joiners. C'est un langage de bas niveau qui permet au spécialiste de décrire lui même les réorganisations de données ; c'est également un langage formé de primitives simples sur lequel on effectuera des transformations optimisantes qui seront présentées au chapitre 3. SJD est un langage outil qui reprend les nœuds de StreamIt mais permet de les assembler de manière arbitraire.

Le deuxième langage, SLICES, est un langage déclaratif qui permet une expression simple des réorganisations de données multidimensionnelles. Les réorganisations de données sont décrites dans un modèle haut-niveau plus accessible. On montre qu'il est possible de compiler SLICES vers SJD, ce qui permet de mélanger les deux langages au sein du même programme, et d'appliquer les transformations optimisantes de SJD sur des programmes SLICES.

Dans la section suivante nous allons présenter quelques applications caractéristiques du traîtement du signal que nous reprendrons pour illustrer et comparer ces deux langages.

2.1 Quelques applications exemples

2.1.1 Multiplication de matrices

Le premier programme est la multiplication matricielle. Soient deux matrices $\mathbf{A} \in \mathcal{M}_{x_0,y_0}$ et $\mathbf{B} \in \mathcal{M}_{x_1,y_1}$. On veut calculer la matrice $\mathbf{C} = \mathbf{A} \times \mathbf{B}$ qui est définie lorsque $x_0 = y_1$.

Décomposons $\mathbf{A}$ en lignes, $A = \begin{bmatrix} A_1 \\ \dots \\ A_{y_0} \end{bmatrix}$ et décomposons $\mathbf{B}$ en colonnes, $B = \begin{bmatrix} B_1 & \dots & B_{x_0} \end{bmatrix}$ alors $\mathbf{C}_{ij} = \mathbf{A}_i.\mathbf{B}_i$, c'est-à-dire le produit scalaire de la i^{e} ligne de $\mathbf{A}$ avec la j^{e} colonne de $\mathbf{B}$.

2.1.2 Transformée rapide de Fourier

Le deuxième programme est une transformée rapide de Fourier (FFT) implémentée selon l'algorithme de Cooley-Tukey[CT65] en Radix-2. L'algorithme utilise une stratégie diviser-pour-regner pour calculer la transformée de Fourier sur une fenêtre de taille 2^N. Pour cela on utilise récursivement la décomposition suivante du k^{e} coefficient de la FFT :

$$\begin{aligned} X_k &= \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N} nk} \\ &= \sum_{m=0}^{N/2-1} x_{2m} e^{-\frac{2\pi i}{N}(2m)k} + \sum_{m=0}^{N/2-1} x_{2m+1} e^{-\frac{2\pi i}{N}(2m+1)k} \\ &= \underbrace{\sum_{m=0}^{N/2-1} x_{2m} e^{-\frac{2\pi i}{N/2} mk}}_{E_k} + e^{-\frac{2\pi i}{N}k} \underbrace{\sum_{m=0}^{N/2-1} x_{2m+1} e^{-\frac{2\pi i}{N/2} mk}}_{O_k} \\ &= E_k + e^{-\frac{2\pi i}{N}k} O_k \end{aligned}$$

Cette décomposition va être appliquée récursivement pour se ramener à des vecteurs de taille 1 pour lesquels la FFT se réduit à l'identité. L'implémentation peut être séparée en deux phases : une première phase de réorganisation des données qui va séparer les valeurs paires et impaires en cascade ; suivie d'une deuxième phase de calcul qui va effectuer les FFT-2 sur chaque paire de données.

2.1.3 Détection de panneaux ferroviaires.

La troisième application est une chaîne de traitement du signal pour détecter les panneaux ferroviaires dans un flux vidéo provenant d'une caméra placée à l'avant d'un train. Ceux-ci sont alors signalés au conducteur par réalité augmentée. Ici nous considérons la première partie de la chaîne qui se charge d'identifier toutes les formes carrées dans un flux d'images.

Pour extraire les formes carrées, nous proposons d'abord d'extraire les droites de l'image puis de sélectionner celles dont l'intersection forme un carré. Il existe diverses méthodes pour extraire les droites d'une image ; nous avons choisi d'utiliser l'algorithme de Hough [DH72]. Chaque point de l'image va voter pour l'ensemble des droites du plan qui pourraient le traverser. À la fin du vote, les droites du plan dont le score est le plus grand correspondent aux droites contenant le plus de points dans l'image. Pour que l'algorithme de Hough soit efficace on n'exécute celui-ci que sur les points appartenant aux contours des objets dans l'image, c'est-à-dire les points à gradient maximal qui sont identifiés avec un filtre de Sobel (1968). Finalement, pour éliminer les contours des petits objets (galets sur les voies, feuilles des arbres) qui ne nous intéressent pas ; nous utilisons en début de chaîne un filtre gaussien qui gomme les petits détails.

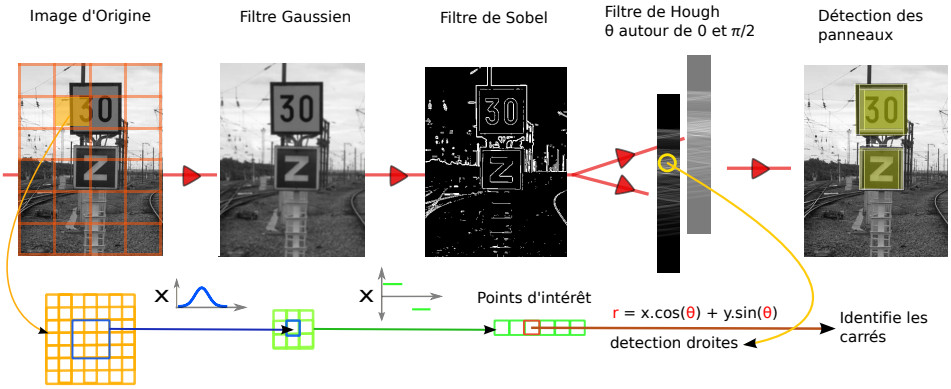


FIGURE 2.1 – Chaîne pour la détection de de panneaux ferroviaires (photographie d'Aurélien Braida)

Il existe des méthodes plus sophistiquées pour détecter des formes carrées dans une image. Néanmoins nous avons choisi cette méthode car elle implémente une chaîne de traitement d'image suffisamment simple pour servir d'exemple et suffisamment complexe pour être représentative.

Filtre Gaussien Le filtre gaussien permet de réduire le contraste des objets de petite taille en convoluant l'image avec le noyau 3×3 suivant :

$$\mathbf{G} = \begin{bmatrix} 1 & 2 & 1 \\ 2 & 3 & 2 \\ 1 & 2 & 1 \end{bmatrix} \otimes \mathbf{image}$$

Filtre de Sobel Une fois les détails gommés, nous appliquons à l'image un filtre de Sobel qui détecte les contours des objets de l'image. Le filtre approxime les dérivées sur l'axe horizontal et vertical à l'aide de deux convolutions : une convolution pour l'axe horizontal,

$$\mathbf{S}_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \otimes \mathbf{G}$$

et une convolution pour l'axe vertical,

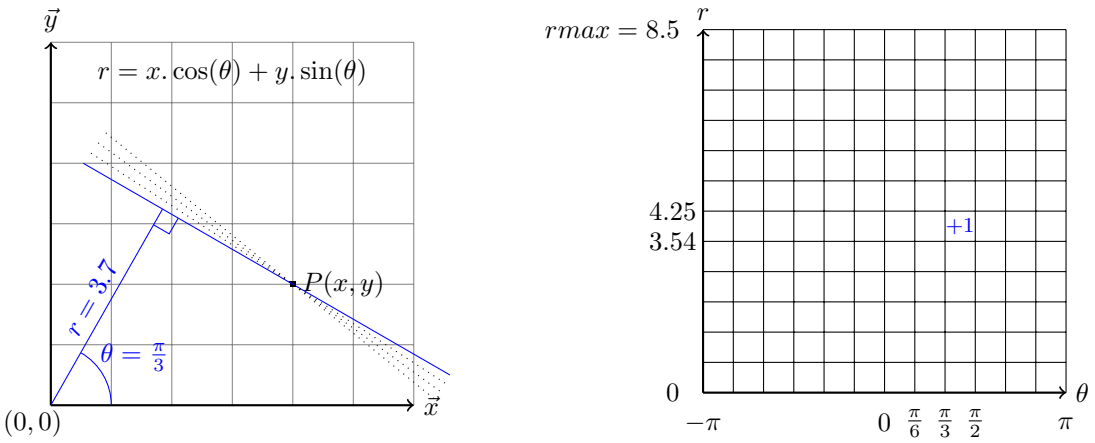
$$\mathbf{S}_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \otimes \mathbf{G}$$

Le filtre de Sobel est séparable, il peut être implémenté avec deux convolution en une dimension en décomposant le noyau en produit de vecteurs :

$$\mathbf{S}_x = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} * \left(\begin{bmatrix} 1 & 0 & -1 \end{bmatrix} * \mathbf{G} \right) \quad \text{et} \quad \mathbf{S}_y = \begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix} * \left(\begin{bmatrix} 1 & 2 & 1 \end{bmatrix} * \mathbf{G} \right)$$

Les points de gradient maximal, correspondant aux bords des objets, sont les points d'intérêt (x, y), qui vérifient l'inéquation :

$$\mathbf{S}_x(x, y)^2 + \mathbf{S}_y(x, y)^2 > \mathbf{seuil}$$



(a) On considère toutes les droites traversant chaque point d'intérêt.

(b) Chaque droite vote pour ses coordonnées dans l'espace transformé.

FIGURE 2.2 – Transformée de Hough.

Filtre de Hough Le dernier filtre utilisé, le filtre de Hough, permet de détecter les droites dans une image. Nous allons expliquer son fonctionnement à l'aide d'un exemple. Considérons la figure 2.2(a), on y a représenté le point d'intérêt P de coordonnées $(4, 2)$. Chaque droite du plan peut être représentée dans un système de coordonnées polaires par un angle θ et une distance à l'origine r. Nous discrétisons ce système de coordonnées comme sur le tableau de la figure 2.2(b) dont nous initialisons les cases à 0.

L'algorithme considère, selon un pas donné, toutes les droites du plan qui traversent P ; on en a représenté quelques unes sur la figure. Pour chaque droite (θ, r) on incrémente de 1 la valeur de la case lui correspondant dans le tableau de la figure 2.2(b). L'algorithme se termine lorsque toutes les droites traversant les points d'intérêt de l'image ont voté pour une des cases du tableau. Les cases qui contiennent les valeurs les plus grandes signalent les droites les plus probables de l'image (puisque beaucoup de points ont voté pour ces droites).

Dans la suite, nous allons présenter les langages SJD et SLICES. Nous utiliserons les exemples précédents pour illustrer les deux langages et comparer leur expressivité.

2.2 Comment exprimer le routage de données de ces différents programmes ?

Nous souhaitons construire un langage qui permette d'exprimer le routage des données des programmes de la section précédente. Pour cela on commence par décomposer les programmes en deux parties : une partie qui se charge du routage des données et une partie qui se charge du calcul sur les données. Par exemple le filtre de Gauss peut être décomposé comme sur la figure 2.3 en :

- un bloc de routage qui extrait un bloc de 9 pixels centré autour de chaque pixel de l'image d'origine ;
- un bloc de calcul qui effectue le produit point à point du bloc extrait avec un noyau Gaussien.

On peut faire cette même séparation sur les autres exemples, ce qui nous permet d'isoler le routage des données pour chacun de ces programmes à l'intérieur de blocs de routage.

Dans cette thèse on se limite aux routages de données statiques. Dans ce cas on peut modéliser la fonction de réorganisation de données d'un bloc de routage comme une application ϕ qui associe

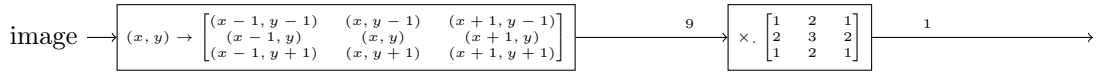


FIGURE 2.3 – Le filtre de Gauss peut être décomposé en un premier bloc de routage de données et un deuxième bloc de calcul.

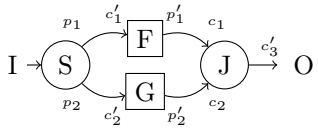


FIGURE 2.4 – Exemple de notations graphiques : le nœud `Split` consomme alternativement p_1 et p_2 éléments de l'entrée I. Ces éléments sont routés vers les filtres F et G respectivement et leurs sorties sont combinées par le nœud `Join` et produites sur la sortie O.

à chaque coordonnée en sortie ($[1, \ldots, m]$) une coordonnée des données en entrée ($[1, \ldots, n]$). La fonction ϕ est une suite à support fini également appelée arrangement avec répétition.

Nous souhaitons construire un langage de routage qui permette d'exprimer tous les arrangements avec répétitions possibles. StreamIt ne le permet pas : par exemple il est impossible d'inverser l'ordre des éléments d'un vecteur. Ceci nous a motivé à construire un langage outil SJD, dérivé de StreamIt, mais permettant de décrire tous les arrangements avec répétition (§ 2.4.2).

Le langage SJD construit, bien que très expressif, est difficile d'accès : pour décrire un réarrangement de données le concepteur est amené à connecter des joiners et splitters de manière complexe et peu intuitive. C'est pourquoi nous proposons un langage de haut-niveau SLICES qui facilite l'expression des blocs de routage.

2.3 Langage SJD

Le langage SJD est proche de ΣC [GBL+08] et de StreamIt. Il modélise les applications parallèles comme des graphes de flots : les nœuds opèrent des calculs et des réorganisations sur les données des flux et les arcs permettent aux nœuds de s'échanger des données.

À chaque exécution d'un nœud, celui-ci consomme un nombre fixe de données sur les arcs entrants, et une fois que son traitement est fini, il produit un nombre fixe de données sur les arcs sortants. Dans le langage SJD nous distinguons plusieurs catégories de nœuds : les nœuds `Filter`, et les nœuds de routage `Split`, `Join`, `Duplicate`.

2.3.1 Représentation graphique

Nous représentons les graphes SJD graphiquement : les nœuds `Filter` sont représentés avec des rectangles ; les nœuds de routage, représentés avec des cercles, sont munis d'une étiquette qui indique leur type. Le type sera parfois suivi d'un index pour distinguer les nœuds entre eux. Les arcs sont décorés avec les consommations et les productions des nœuds. L'exemple de la figure 2.4 récapitule ces notations.

Dans la suite nous présentons la sémantique des différents nœuds dont nous avons représenté le comportement sur la figure 2.5.

2.3.2 Filtres

Les filtres sont des nœuds de calcul. Un filtre $F(c_1, p_1)$ a une entrée et une sortie. À chaque exécution du filtre c_1 éléments sont consommées sur l'entrée et p_1 éléments produits sur la sortie.

Les éléments produits à chaque exécution sont le résultat d'un calcul réalisé sur les éléments consommés. On notera f, la fonction qui a partir des éléments consommés calcule les éléments produits. Un filtre *pur* possède une fonction f *pure*, c'est-à-dire qui n'a pas d'effets de bords et ne conserve pas d'état interne entre deux exécutions du nœud. Si la fonction f a un état interne, on dira que le filtre est *impur*.

Dans cette thèse les fonctions seront décrites en pseudo-code avec une syntaxe proche du langage `C` à laquelle on a adjoint les primitives `push(x)` et `x = pop()` permettant respectivement de consommer un élément sur le canal entrant et de produire un élément sur le canal sortant. Cette syntaxe est empruntée au langage StreamIt.

2.3.3 Nœuds de routage

Les nœuds de routage agissent sur l'ordre des données mais ne font pas de calculs. On y trouve les nœuds `Input` et `Constant` qui sont les flux entrants du programme, les nœuds `Output` et `Sink` qui sont les flux sortants du programme, les nœuds `Join` qui regroupent plusieurs flux et enfin les nœuds `Split` et `Duplicate` qui distribuent les données sur plusieurs flux.

Input et Constant

Les nœuds `Input` représentent les entrées d'un programme, on les notera I. On considérera que les nœuds `Input` produisent une infinité de données. En pratique, si un nombre fini de données est produit, le nœud `Input` finira par bloquer, ce qui conduira à l'arrêt progressif du programme.

Parfois, un noyau de calcul doit travailler avec des blocs de données d'une taille bien particulière (par exemple la transformation rapide de Fourier travaille sur des tailles puissance de 2) ; si les données de l'utilisateur ne sont pas au bon format on doit ajouter autant de zéros (par exemple) que de données manquantes. Dans ce genre de situations nous souhaitons disposer d'une source produisant de manière infinie une valeur donnée. Nous appellerons une telle source, `Constant`, et nous la noterons $C(v)$ où v est la valeur de la source.

Output et Sink

Les nœuds `Output` représentent les sorties d'un programme, on les notera O. Un nœud `Output` consomme toutes les données qui lui sont envoyées. Parfois dans un programme nous souhaitons décimer certaines données, pour effectuer cette opération nous pouvons les envoyer dans un puits qui détruit ces dernières ; il sera appelé `Sink` et noté T. Nous considérons que les données reçues par un `Sink` ne sont jamais observées par l'utilisateur.

Join

Les nœuds `Join` $J(c_1 \dots c_n)$ ont n entrées et une sortie. Chaque entrée i est associée à un taux de consommation c_i.

Le nœud `Join` a un comportement cyclo-statique. La première fois qu'il est exécuté il consomme c_1 données sur sa première entrée et les écrit sur sa sortie. Puis il consomme c_2 entrées sur la seconde entrée qu'il écrit sur sa sortie, et ainsi de suite. Formellement lors de sa k^{e} exécution le nœud `Join` consomme c_u éléments sur sa u^{e} sortie, avec $u = ((k-1) \mod n) + 1$. Ce nœud permet comme nous venons de le voir d'assembler plusieurs flux de données différents.

Split

À l'opposé on trouve le nœud `Split` $S(p_1 \dots p_m)$, qui permet de séparer un flot de données en plusieurs parties. Le nœud `Split` possède une entrée et m sorties. Chaque sortie j possède un taux de production p_j. Lors de sa k^{e} exécution un nœud `Split` consomme p_v éléments sur sa v^{e} entrée, avec $v = ((k-1) \mod m) + 1$.

Duplicate

Le nœud `Duplicate` $D(m)$ copie les éléments reçus sur son unique entrée sur chacune des m sorties.

2.3.4 Sucre syntaxique

On propose deux extensions au modèle SJD de base pour prendre en compte les filtres à plusieurs entrées/sorties et les nœuds de routage sans-délai. Ces extensions, par construction, peuvent se réecrire en utilisant le modèle de base. Elles n'augmentent pas l'expressivité du langage mais apportent un "sucre syntaxique" permettant d'écrire de manière plus commode certains patrons.

Filtres à plusieurs entrées et plusieurs sorties

Les filtres à plusieurs entrées et plusieurs sorties sont définis par l'expansion de la figure 2.6. Pour représenter un filtre à plusieurs entrées et sorties, on assemble les entrées avec un nœud `Join` et les sorties avec un nœud `Split`. On se retrouve alors avec une seule entrée et une seule sortie.

Cette modélisation impose un rendez-vous sur les entrées : le filtre doit attendre d'avoir reçu toutes ses entrées avant de pouvoir s'exécuter. Bien que nous n'ayons pas exploré cette possibilité, on pourrait imaginer d'autres sémantiques dans lesquelles le filtre peut commencer à travailler avant d'avoir reçu l'intégralité des données entrantes.

Nœuds de routage sans-délai

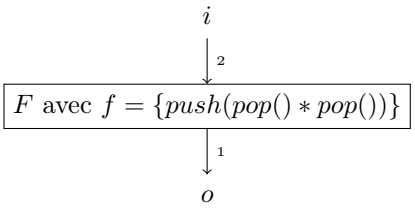
On propose deux variantes, qualifiées *sans-délai*, des nœuds `Split` et `Join`. Elles sont notées respectivement $\underline{S}$ et $\underline{J}$. Les nœuds sans-délai réalisent la même organisation de données que leurs variantes normales ; la différence réside dans la sémantique de déclenchement d'un nœud. Prenons comme exemple un nœud $J(1, 4)$. La première exécution sera déclenchée après réception de 1 élément en entrée et la deuxième exécution sera déclenchée après réception de 4 éléments, en réalisant ainsi un cycle d'exécution. Puis la troisième exécution sera déclenchée après réception de 1 élément, et ainsi de suite alternativement.

Ceci peut-être problématique dans le cas de programmes avec des cycles comme celui de la figure 2.7 dont nous allons dérouler l'exécution. J s'exécute une première fois en produisant une valeur sur S. À ce moment, S qui n'a pas encore 4 valeurs en entrée ne peut pas s'exécuter et ne peut donc pas produire de valeurs. Le programme est bloqué.

Si S et J laissaient passer les valeurs au fur à mesure de leur réception, le programme pourrait s'exécuter normalement. C'est exactement ce que font les variantes sans-délai des nœuds : dès réception d'un élément il est envoyé sur la sortie adéquate. Avec les variantes sans-délai le graphe s'exécute sans interblocage.

Les variantes sans-délai sont modélisables dans le formalisme SJD puisqu'elles sont exprimées par à une réécriture équivalente qui n'utilise que des nœuds normaux. Nous remplaçons, pour un nœud `Join` sans-délai, chaque canal entrant de consommation c par c canaux de consommation unitaire. Puis nous faisons de même avec les canaux sortants (cf. figure 2.8).

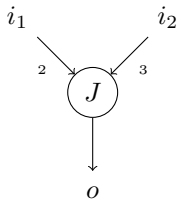
$t(i) = \{0, 1, 2, 3, 4, 5, 6 \ldots\}$



$t(o) = \{0, 6, 20, \ldots\}$

(a) Le nœud `Filter` applique une fonction f aux données en entrée. Dans le corps de f, la primitive *pop*() consomme une donnée en entrée et la primitive *push*() produit une donnée en sortie.

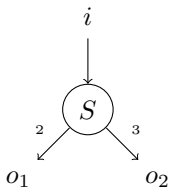
$t(i_1) = \{0, 1, 5, 6, \ldots\}$ $t(i_2) = \{2, 3, 4 \ldots\}$



$t(o) = \{0, 1, 2, 3, 4, 5, 6 \ldots\}$

(b) Le nœud `Join` réunit plusieurs flux de données.

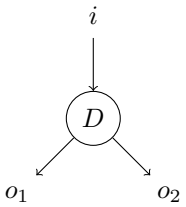
$t(i) = \{0, 1, 2, 3, 4, 5, 6 \ldots\}$



$t(o_1) = \{0, 1, 5, 6, \ldots\}$ $t(o_2) = \{2, 3, 4 \ldots\}$

(c) Le nœud `Split` sépare les données en plusieurs flux.

$t(i) = \{0, 1, 2, 3, 4, 5, 6 \ldots\}$



$t(o_1) = t(o_2) = \{0, 1, 2, 3, 4, 5, 6 \ldots\}$

(d) Le nœud `Duplicate` fait plusieurs copies d'un même flux.

FIGURE 2.5 – Comportement des nœuds SJD. Pour chaque nœud on a figuré les éléments du flux sur les entrées, les *traces entrantes* $t(i)$, et les éléments produits sur les sorties, les *traces sortantes* $t(o)$.

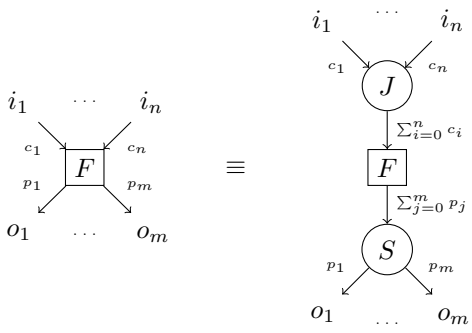


FIGURE 2.6 – Les filtres à plusieurs entrées et sorties peuvent être incorporés au modèle SJD avec l'expansion ci-dessus.

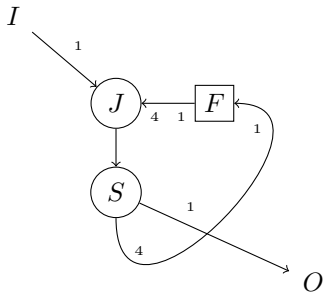


FIGURE 2.7 – Le cycle ci-dessus présente un interblocage ; il ne produira aucune valeur sur la sortie. L'interblocage peut-être évité en utilisant les variantes sans-délai des nœuds de routage.

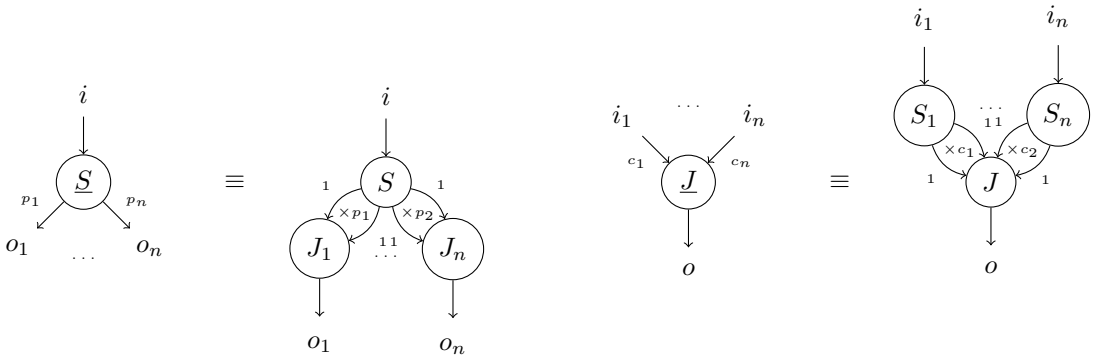


FIGURE 2.8 – Les nœuds sans-délai peuvent être exprimés en utilisant les variantes normales.

2.3.5 Représentation des graphes SJD en ΣC

La représentation graphique des programmes SJD utilisée précédemment est pratique pour visualiser les programmes et sera utilisée dans la suite. Néanmoins, pour développer un programme nous lui préférerons une représentation textuelle. Même si des IDE graphiques (Gaspard [DBDM02] ou Ptolemy [BHLM94]) existent, une représentation textuelle permet l'utilisation des nombreux outils de développement (IDE, Gestionnaire de Versions, Patch) auxquels les concepteurs sont habitués. Dans cette section nous présenterons très succinctement les bases de ΣC, un langage pour les flots de données que nous pouvons utiliser pour décrire les graphes SJD textuellement.

ΣC [GBL+08] étend le langage C avec des primitives adaptées à l'expression de graphes de flots de données complexes avec du contrôle. Une application ΣC est composée d'agents (appelés nœuds dans le formalisme SJD) qui communiquent à travers des canaux. Le point d'entrée d'une application est l'agent unique racine.

Structure d'un agent Un agent est déclaré avec le mot clé éponyme, suivi du nom de l'agent et d'éventuels paramètres d'instanciation. La déclaration d'un agent définit un prototype d'agent (à l'instar du constructeur `class` des langages objets). Pour obtenir une instance concrète d'un agent, on utilisera la primitive `new`.

Un agent est composé de trois sections :

- La section `interface` qui rassemble les déclarations des canaux entrants et sortants de l'agent, ainsi que des types de données qui y transitent.
- La section `map` qui permet d'instancier d'autres agents et de les connecter. Cette section définit le graphe d'interconnexion de l'agent au reste de l'application. Deux agents seront reliés grâce à la primitive `connect`.
- Le corps de l'agent, où le comportement de l'agent est défini à l'aide de fonctions `exchange` qui consomment et produisent des donnés sur les canaux de l'agent.

Agent systèmes Un ensemble d'agents standards, appelés agent systèmes, sont prédéfinis dans l'environnement d'exécution. Parmis ces agents on retrouve les nœuds `Split`, `Join` et `Duplicate`, permettant de séparer et rassembler des données dans l'application.

Subgraph Le langage ΣC propose également un composant `subgraph`. Semblable aux agents, ce composant possède des sections `interface` et `map`, permettant d'instancier et de définir un sous-graphe d'agents réutilisable. Contrairement aux agents, un `subgraph` n'a pas de comportement propre, ni d'état interne.

Pour représenter un programme SJD en ΣC, les nœuds `Filter` seront remplacés par des agents équivalents ; les nœuds de routage seront remplacés par les agents systèmes correspondants ; les interconnexions du graphe seront réalisées avec la primitive `connect` dans une section `map`. Dans la section suivante nous donnons un graphe SJD permettant de multiplier deux matrices et sa représentation en ΣC (cf. extrait de code 2.1). Cet exemple permettra de mieux comprendre la syntaxe des primitives.

Nous n'avons décrit que les fonctionalités de ΣC nécessaires pour représenter les graphes SJD ; cependant le langage possède d'autres fonctionnalités plus avancées. Pour n'en citer que deux :

- Les agents ΣC peuvent, si nécessaire, être décrits dans un modèle plus général que le modèle cyclo-statique dans lequel le comportement des agents sera défini par une machine à états autorisant un certain degré de non-déterminisme.
- Certains agents systèmes peuvent être contrôlés par des prédicats pour décrire des comportements dynamiques.

Le lecteur intéressé pourra se référer à la définition du langage [GBL+08].

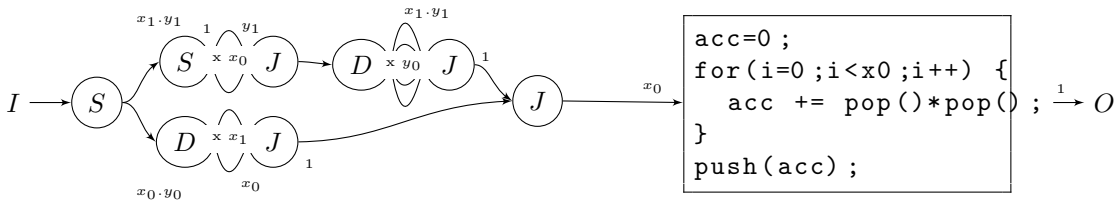


FIGURE 2.9 – Multiplication de matrices en SJD.

2.3.6 Quelques exemples de programmes SJD

Voyons maintenant comment coder les exemples présentés en début de chapitre avec le langage SJD.

Multiplication de matrices

On a reformulé en SJD en figure 2.9 la multiplication matricielle tirée des exemples du compilateur StreamIt. L'entrée I va produire les éléments de la matrice $\mathbf{A}$ suivis des éléments de la matrice $\mathbf{B}$ [1]. Les éléments des matrices sont produits dans l'ordre des lignes. Ils vont être séparés en deux flux par le premier `Split` qui envoie les éléments de $\mathbf{A}$ sur la branche inférieure et les éléments de $\mathbf{B}$ sur la branche supérieure.

Sur la branche inférieure, on va réaliser autant de copies des lignes de $\mathbf{A}$ que de colonnes de $\mathbf{B}$. On utilise donc un nœud $D(x_1)$ qui va produire x_1 copies de la matrice $\mathbf{A}$. Ces copies alimentent un nœud $J(\underbrace{x_0 \dots x_0}_{\times x_1})$ qui les regroupe par blocs de taille x_0 (la longueur d'une ligne de $\mathbf{A}$).

Sur la branche supérieure, on commence par transposer la matrice $\mathbf{B}$ de manière à pouvoir lire les éléments dans l'ordre des colonnes. Un premier nœud $S(\underbrace{1 \dots 1}_{\times x_0})$ consomme les éléments d'une ligne et les éclate selon leur colonne sur ses x_0 sorties. Puis un nœud $J(\underbrace{y_1 \dots y_1}_{\times x_O})$ les regroupe par colonne. Une fois la matrice transposée, on fait y_0 copies de la matrice $\mathbf{B}$ avec la combinaison $D(y_0) \rightarrow J(\underbrace{x_1.y_1 \dots x_1.y_1}_{y_0})$.

Finalement, le nœud $J(1,1)$ en sortie de graphe entrelace chaque ligne de $\mathbf{A}$ avec chaque colonne de $\mathbf{B}$. Puis le nœud `Filter` effectue le produit scalaire des lignes de $\mathbf{A}$ et des colonnes de $\mathbf{B}$ et produit les éléments de la matrice $\mathbf{C}$.

Pour illustrer la représentation textuelle de SJD on donne aussi le même programme écrit en ΣC dans l'extrait de code 2.1.

Transformée rapide de Fourier

Tout comme pour la multiplication matricielle, pour implémenter la FFT nous avons repris une version distribuée avec StreamIt (`FFT5.str`) que nous avons retranscrit en SJD sur la figure 2.10.Les deux étapes de l'algorithme apparaissent clairement dans le graphe.

Un premier sous-graphe composé de nœuds `Split` et `Join` réorganise les données en séparant les données paires et impaires. La série de `Split` éclate les éléments du vecteur sur différents canaux, puis la cascade de `Join` les rassemble par groupes de 2, 4 et 8 éléments qui correspondent

1. On aurait pu également utiliser des `Input` distincts pour chaque matrice.

```
agent DotProduct (int y0) {
  /* Calcule le produit scalaire de deux vecteurs de taille y0 */
  interface {
    in<float> I;
    out<foat> O;
  }
  void start() exchange(I[] A[2*yo], O[] acc) {
    acc = 0;
    for (int i = 0; i < yo; i++)
      acc += A[i]*A[i+1];
  }
}

subgraph Transpose(int x, int y) {
  /* Transpose une matrice de dimensions (x,y) */
  interface {
    in<float> I; out<float> O;
  }
  map {
    agent S = new Split<float>(x, 1);
    agent J = new Join<float>(x, y);
    for (int i = 0; i < x; i ++)
      connect(S.output[i], J.input[i]);
  }
}

subgraph DuplicateVectors(int n, int y) {
  /* Lit des vecteurs de taille y; chaque vecteur lu est emis n fois. */
  interface {
    in<float> I; out<float> O;
  }
  map {
    agent D = new Dup<float>(n, y);
    agent J = new Join<float>(n, y);
    for (int i = 0; i < x; i ++)
      connect(D.output[i], J.input[i]);
  }
}

subgraph MatrixMultiply (int x0, int y0, int x1, int y1) {
  /* Realise la multiplication de deux matrices de dimensions (x0,y0) et (x1,y1) */
  interface {
    in<float> A, B;
    out<float> C;
  }
  map {
    /* On duplique les lignes de A */
    agent DA = new DuplicateVectors(x1,x0);
    connect(A, DA.I);

    /* On transpose B pour dupliquer ses colonnes */
    agent T = new Transpose(x1,y1);
    agent DB = new DuplicateVectors(y0,x1*y1);
    connect(B, T.I);
    connect(T.O, DB.I);

    /* On multiplie chaque ligne avec chaque colonne */
    agent J = new Join(2, 1);
    agent M = new DotProduct(x0);
    connect(DA.O, J.input[0]);
    connect(DB.O, J.input[1]);
    connect(J.output[0], M.I);
    connnect(M.O, C);
  }
}
```

Extrait de code 2.1 – Le graphe SJD pour la multiplication matricielle exprimée en ΣC

aux séparations pair/impair à chaque niveau de la récursion. Pour des vecteurs de taille 16, on se retrouve à la sortie de cet étage avec les données dans l'ordre

$$[I_0, I_8, I_4, I_{12}, I_2, I_{10}, I_6, I_{14}, I_1, I_9, I_5, I_{13}, I_3, I_{11}, I_7, I_{15}].$$

Un deuxième sous-graphe en aval, est composé de nœuds de calcul qui réalisent les FFT-2 à chaque niveau de la récursion. Bien sûr, pour chaque filtre on a précalculé les coefficients $e^{-\frac{2\pi i}{N}k}$. Un filtre ne fait donc qu'une multiplication et une addition complexes.

Filtre gaussien

Nous allons écrire en SJD le filtre gaussien présenté en § 2.1.3. Pour l'exemple, nous supposerons que l'on travaille sur une image de taille $(w = 10) \times (h = 10)$. Le filtre gaussien convolue l'image avec un noyau 3×3. Nous décomposons le filtre dans les deux étapes représentées sur la figure 2.11(a) : à droite, le nœud `Filter` se charge de calculer la convolution, à gauche, un sous-graphe SJD se charge d'extraire les blocs qui doivent être convolués.

Le nœud `Filter` s'occupe de calculer le produit point par point des blocs de 3×3 pixels avec le noyau de la convolution. Ce filtre consomme donc 9 pixels à chaque exécution et produit 1 pixel en sortie. La fonction de calcul associée au filtre est transcrite en figure 2.11(c).

Pour chaque pixel de l'image d'origine il faut extraire le bloc contenant les 8 voisins directs. Ceci est clairement une opération de réorganisation de données que nous pouvons exprimer en utilisant les nœuds de routage. Le graphe de la figure 2.11(b) est une implémentation possible de l'extraction des blocs. Pour comprendre son fonctionnement nous allons analyser chacun des étages qui le composent : extraction des lignes, transposition et composition des blocs.

Extraction des lignes Le premier étage, va extraire les lignes des blocs 9×9. Pour cela l'image est dupliquée en trois flux à l'aide du nœud D en entrée. Le premier flux est préfixé d'une valeur par défaut $(C - J(1, 10))$, et sa dernière valeur est supprimée $(S(10, 1) - T)$. On obtient sur la première branche l'arrangement $[0, I_0, I_1, I_2, I_3, I_4, I_5, I_6, I_7, I_8]$ où I_k représente le k^{e} point de l'image. La deuxième copie du flux n'est pas altérée ; on obtient donc sur la deuxième branche l'arrangement identité $[I_0, I_1, I_2, I_3, I_4, I_5, I_6, I_7, I_8, I_9]$. La première valeur du troisième flux est supprimée $(S(1, 9) - T)$, et il est postfixé d'une valeur par défaut $(C - J(9, 1))$; on obtient l'arrangement $[I_1, I_2, I_3, I_4, I_5, I_6, I_7, I_8, I_9, 0]$.

Enfin, les trois flux sont recomposés en utilisant le nœud $J(1, 1, 1)$, ce qui pour chaque ligne de l'image se traduit dans l'arrangement $[0, I_0, I_1, \quad I_0, I_1, I_2, \quad I_2, I_3, I_4, \quad \ldots \quad I_8, I_9, 0]$. En sortie de ce premier étage on obtient donc pour chaque pixel k d'une ligne, les triplets composés des pixels $[I_{k-1}, I_k, I_{k+1}]$.

Transposition Le deuxième étage, va réordonner les blocs par colonnes. Pour cela nous utilisons l'association $S(\underbrace{3, \ldots, 3}_{\times 10}) - J(\underbrace{30, \ldots, 30}_{\times 10})$ qui range les triplets par colonnes :

$$[0, I_0, I_1, \quad 0, I_{10}, I_{11}, \quad \ldots \quad , 0, I_{90}, I_{91}, \quad I_0, I_1, I_2, \quad I_{10}, I_{11}, I_{12}, \ldots]$$

Composition des blocs Le dernier étage va dupliquer et regrouper les lignes produites par le premier étage pour former les blocs attendus. En fait, son fonctionnement est le même que pour le premier étage, sauf qu'il travaille sur des blocs de taille 3. En sortie du troisième et dernier étage nous allons obtenir le réarrangement :

$$[0, 0, 0, 0, I_{10}, I_{11}, 0, I_{20}, I_{21}, \quad 0, I_{10}, I_{11}, 0, I_{20}, I_{21}, 0, I_{21}, I_{22} \quad \ldots]$$

Ce sont exactement les blocs de taille 3×3 que l'on obtiendrait en prenant les voisins de tous les pixels en parcourant les points de l'image selon l'ordre des colonnes.

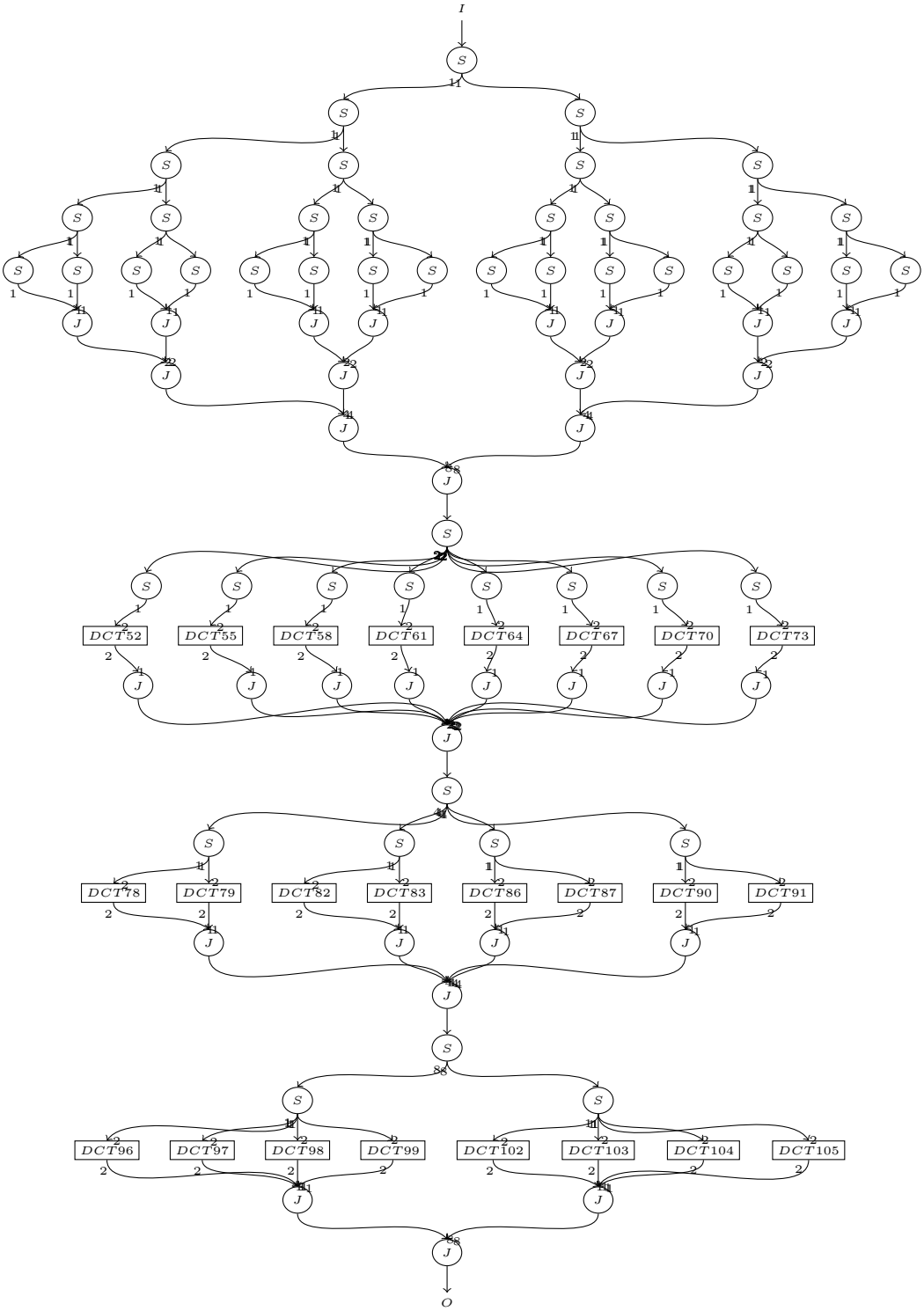
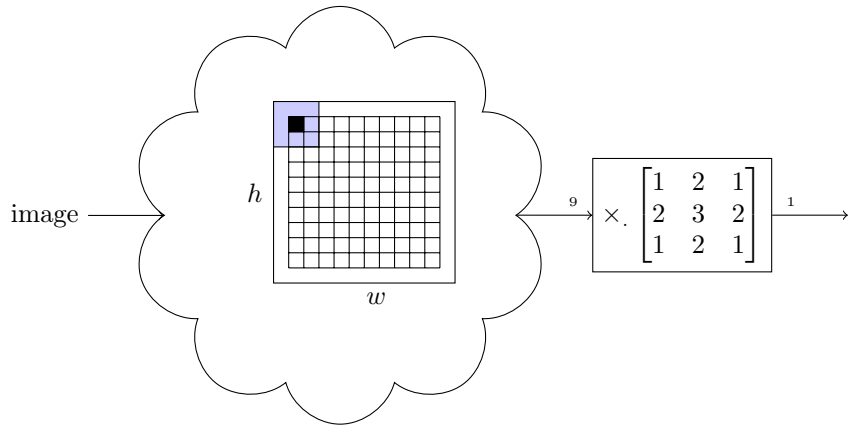
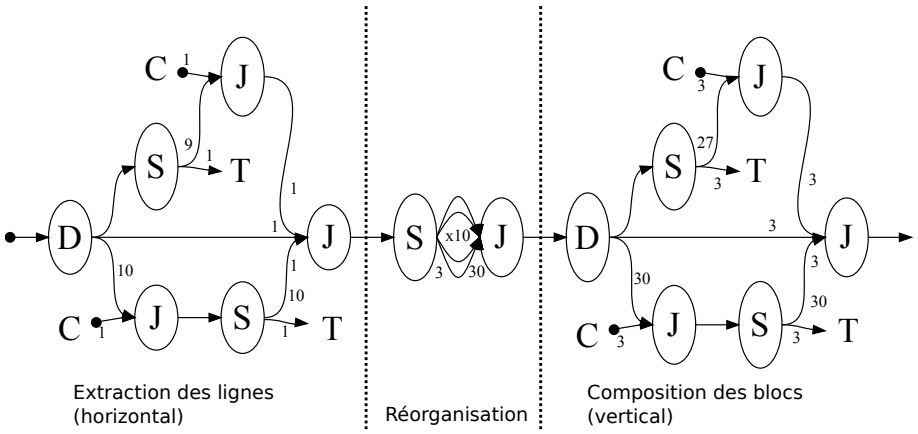


FIGURE 2.10 – L'implémentation de la FFT en SJD sur des vecteurs de taille 16.



(a) Les deux étapes du filtre de Gauss : extraction des blocs de données et calcul de la convolution.



(b) Extraction des blocs de données en SJD.

```
i1 = pop(); i2 = pop(); ... ; i9 = pop();
result = 1 * i1 + 2 * i2 + 1 * i3
       + 2 * i4 + 3 * i5 + 2 * i6
       + 1 * i7 + 2 * i8 + 1 * i9 ;
push(result);
```

(c) Code de la fonction f associée au nœud `Filter`.

FIGURE 2.11 – Une implémentation du filtre gaussien dans le langage SJD.

En passant ces blocs là au filtre F, nous obtenons la transformée de Gauss sur une image de taille 10×10. Puisque les blocs ont été obtenus en itérant en premier sur les colonnes l'image sera transposée ; si on le désirait on pourrait la redresser en rajoutant un étage de transposition en bout de chaîne $(S(\underbrace{1, \ldots, 1}_{\times 10}) - J(\underbrace{10, \ldots, 10}_{\times 10}))$.

Filtre de Sobel

Le filtre de Sobel est également basé sur deux convolutions ; on peut donc réutiliser l'implémentation réalisée pour le filtre gaussien, en adaptant les valeurs du noyau de convolution. Une alternative consiste à utiliser la séparabilité du filtre, en le décomposant en deux convolutions de dimension 1. Il suffit alors d'extraire les blocs de taille 3×1 et 1×3 de l'image à l'aide du premier étage du graphe SJD utilisé pour le filtre Gaussien.

Une fois les convolutions calculées il faut seuiller les points d'intérêt, pour cela on peut rajouter un filtre en fin de chaîne qui applique la fonction,

```
if (pop() > seuil)
  coord = [x,y];
else
  coord = [-1,-1];
push(coord);
```

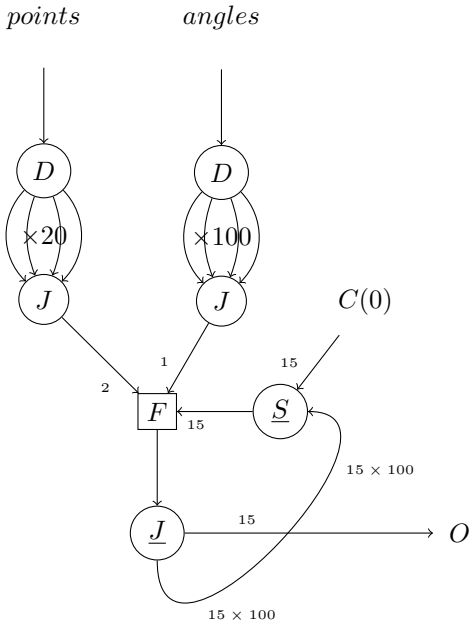
On remarque que cette fonction produit les coordonnées des points d'intérêt, et $(-1, -1)$ pour les autres points. On est obligé de produire tout de même des données pour les points qui ne nous intéressent pas. En effet, le langage SJD est basé sur les CSDF : un filtre doit toujours satisfaire son cycle de production. Il est interdit de produire des valeurs parfois et parfois pas. Pour exprimer ce genre de comportements il faudrait se placer dans un formalisme moins contraint comme les DDF.

Filtre de Hough Le filtre de Hough s'écrit en pseudo-code :

```
for x,y in points:
  for angle in angles:
    if ((x,y) != (-1,-1)):
      r = x * cos(angle) + y * sin(angle)
      H[r,angle] += 1
```

Nous observons que le corps du nid de boucles n'est pas une fonction pure. En effet, il existe une dépendance sur H entre les itérations successives. Nous pourrions utiliser un filtre impur, le graphe SJD serait alors simplement un nœud `Filter` (dont la fonction associé est le pseudo-code ci-dessus) qu'alimenteraient les entrées : les points de l'image et les angles. Ici nous avons choisi de montrer qu'il est parfois possible de transformer un filtre impur en filtre pur en exposant les dépendances de données dans le graphe. L'intérêt d'une telle opération apparaîtra dans le chapitre 3 où l'on verra que les transformations de graphes SJD ne s'appliquent qu'aux nœuds purs.

Supposons que la taille de l'image soit 10×10 et qu'on échantillonne 20 valeurs sur $[-\pi; \pi]$ et 15 valeurs sur $[0; rmax]$. Une version en SJD de l'algorithme est donné en figure 2.12. L'idée derrière cette implémentation c'est que les colonnes de H pour des angles différents sont indépendantes. On peut donc rendre le filtre pur en passant explicitement l'état de la colonne en cours $Hcol$ à travers le cycle du graphe. Pour chaque valeur de l'angle on traite les 100 points de l'image en modifiant pour chaque point la colonne $Hcol$ qui est propagée à travers le cycle. Une fois l'image traitée, on produit la colonne courante sur la sortie à l'aide du nœud sans-délai J et on réinitialise le vecteur $Hcol$ à zéro à l'aide du nœud sans-délai S. On peut alors traiter l'angle suivant.



(a) Graphe SJD qui réalise un filtre de Hough.

```
x,y = p.pop(2);
angle = angle.pop(2);
Hcol = Hcol.pop(15);

if ((x,y) != (-1,-1)) {
  r = x*cos(angle) + y*sin(angle);
  Hcol[r] += 1;
}
push(Hcol);
```

(b) Code de la fonction f associée au nœud `Filter`. Sur les filtres à plusieurs entrées, la primitive pop doit être appelée avec le nom du canal sur lequel on veut consommer.

FIGURE 2.12 – Implémentation du filtre de Hough en SJD.

2.4 Propriétés de SJD

Maintenant que nous nous sommes familiarisés avec le langage SJD à travers ces quelques exemples, nous allons définir sa sémantique formellement et étudier certaines de ses propriétés.

Proposition 2.4.1. *Les graphes SJD sont des graphes dans le modèle CSDF.*

Démonstration. Cette propriété découle du fait que, par construction, tous les types de nœuds du langage SJD ont un comportement cyclo-statique :

- les nœuds $F(c_1, p_1)$ ont pour cycle de consommation $\{c_1\}$ et pour cycle de productions $\{p_1\}$;
- les nœuds I et $C(v)$ ont pour cycle de production $\{1\}$;
- les nœuds O et T ont pour cycle de consommation $\{1\}$;
- les nœuds $D(m)$ ont pour cycle de consommation $\{1\}$ et pour cycle de production sur la i^{e} sortie $\{1\}$;
- les nœuds $S(c_1 \dots c_m)$ ont pour cycle de consommation $\{c_1, \dots, c_m\}$ et pour cycle de production sur la i^{e} sortie $\{\underbrace{0, \dots, 0,}_{\times i-1} c_i, 0, \dots, 0\}$;
- les nœuds $J(p_1 \dots p_n)$ ont pour cycle de production $\{p_1, \dots, p_n\}$ et pour cycle de consommation sur la i^{e} entrée $\{\underbrace{0, \dots, 0,}_{\times i-1} p_i, 0, \dots, 0\}$.

□

2.4.1 Propriétés des CSDF

Les programmes écrits dans le langage SJD héritent donc de la sémantique et de toutes les propriétés des CSDF (et par transitivité des KPN). Dans cette section nous allons rappeler quelques résultats de la théorie des CSDF dont nous aurons besoin pour étudier l'expressivité du langage SJD et pour définir la correction des transformations de graphes CSDF dans le chapitre 3. Pour plus d'informations sur les propriétés ci-dessous le lecteur pourra se référer à [LP95] qui fait une excellente revue de la théorie des flots de données. Certaines des propriétés ci-dessous on été prouvées sur les SDF, mais s'étendent naturellement aux CSDF.

Trace d'un canal

Les données transitant dans les canaux sont représentées dans un alphabet $\mathcal{A}$. On notera $\mathcal{A}^\star$ l'ensemble des *traces* finies ou infinies formées à partir d'éléments dans $\mathcal{A}$.

Définition 2.4.1 (ordre préfixe)**.** *On peut munir $\mathcal{A}^\star$ d'un ordre partiel préfixe : $X \sqsubseteq Y$* ssi *X est un préfixe de Y.*

Définition 2.4.2 (majorant)**.** *Un ensemble de traces $\mathcal{X}$ possède un majorant Y* ssi *$\exists Y \in \mathcal{A}^\star$ tel que $\forall X \in \mathcal{X}, X \sqsubseteq Y$.*

Définition 2.4.3 (borne supérieure)**.** *Un ensemble de traces $\mathcal{X}$ possède une borne supérieure $\sqcup\mathcal{X}$* ssi

- *$\sqcup\mathcal{X}$ est un majorant de $\mathcal{X}$.*
- *$\sqcup\mathcal{X}$ est le plus petit (selon $\sqsubseteq$) des majorants de $\mathcal{X}$.*

Proposition 2.4.2 (suite de traces croissante)**.** *Une suite de traces (finie ou infinie)* $\mathcal{X} = \{X_1, X_2, \ldots\}$ *qui vérifie* $X_1 \sqsubseteq X_2 \sqsubseteq \ldots$ *admet une borne supérieure.*

On étend naturellement les propriétés ci dessus aux ensembles de $n \in \mathbb{N}^\star$ traces $(\mathcal{A}^\star)^n$, pour plus de détails consulter [Kah74][LP95].

Chaque canal d'un CSDF va transporter une suite possiblement infinie de données, que nous appellerons trace du canal. La trace d'un canal e sera alors notée $t(e) = \{x_1, x_2, \ldots\} \in \mathcal{A}^\star$. Dans le cas d'un graphe SJD, puisque les nœuds `Input` et `Output` ont un seul arc, nous noterons sans ambiguïté que la trace du nœud est la trace de son unique arc. Munis de cette notation les traces des entrées et les traces des sorties d'un graphe peuvent être définies.

Définition 2.4.4 (Traces entrantes et sortantes d'un graphe SJD)**.** *Les traces entrantes sont*

$$I(G) = [t(i) : i \in Inputs \cup Constants]$$

respectivement les traces sortantes sont

$$O(G) = [t(o) : o \in Outputs \cup Sinks]$$

Définition 2.4.5 (Traces entrantes et sortantes d'un sous-graphe SJD)**.** *Soit un sous-graphe* $H \subseteq G$*. Les* arcs frontières *de* H *sont les arcs entre* H *et* $G \backslash H$*). Les traces entrantes (resp. sortantes) d'un sous graphe* $H \subseteq G$ *sont les traces des* arcs frontières *entrants (resp. sortants).*

Définition 2.4.6 (Taille des traces)**.** *Étant donné un vecteur de traces* $O[G] = [t_1, \ldots, t_n]$*, nous appellerons taille des traces le vecteur d'entiers,* $\overline{O[G]} = [\overline{t_1}, \ldots, \overline{t_n}]$ *où* $\overline{t_i}$ *est le nombre d'éléments dans* t_i*.*

Continuité et Monotonicité Soit un nœud d'un CSDF avec n arcs en entrée et m arcs en sortie, on peut lui associer un processus $F : (\mathcal{A}^\star)^n \mapsto (\mathcal{A}^\star)^m$, qui met en relation les traces en entrée avec les traces en sortie.

Définition 2.4.7. *(Continuité) Soit une suite croissante de traces* $\mathcal{X} = \{X_1 \sqsubseteq X_2 \sqsubseteq \ldots\}$*. Un processus est dit* continu ssi *la suite* $F(\mathcal{X}) = F(X_1), F(X_2), \ldots$ *admet une borne supérieure et que* $F(\sqcup\mathcal{X}) = \sqcup F(\mathcal{X})$*.*

Définition 2.4.8 (Monotonicité)**.** *Un processus* F *est monotone* ssi $X \sqsubseteq X' \implies F(X) \sqsubseteq F(X')$*. Tout processus continu est monotone dans les topologies de Scott[Sco72]. En particulier, dans le cadre des flots de données on pourra se référer à la preuve dans [LP95] (la réciproque est fausse).*

Théorème 2.4.3 (Continuité d'un CSDF)**.**

- *Chaque nœud d'un CSDF peut-être représenté par un processus continu [Kah74].*
- *(composition) Un CSDF lui même peut-être représenté par un processus continu. Ce résultat est prouvé par Panangaden dans [PS92].*

La monotonicité d'un CSDF garantit qu'il est exécutable de manière itérative, en effet un nœud peut commencer à calculer avant d'avoir reçu toutes les données en entrée puisque les entrées futures n'affecteront que les sorties futures. La continuité d'un CSDF interdit à un nœud de consommer une séquence infinie en entrée avant de produire des données.

Ordonnancement

Les résultats ci-dessous ont été établis pour les SDF puis étendus dans [BELP95] pour les CSDF. Quand on parle de production ou de consommation d'un canal dans cette section on considère que c'est le nombre d'éléments produits ou consommés durant un cycle complet du nœud CSDF.

Définition 2.4.9 (Ordonnancement)**.** *Un ordonnancement est un ordre d'exécution des acteurs du graphe. Chaque nœud peut apparaître plusieurs fois dans l'ordonnancement.*

Un ordonnancement fini est composé d'un nombre fini d'apparitions. Par exemple l'ordonnancement fini AAABB, correspond à trois exécutions du nœud A, suivies de deux exécutions du nœud B.

Définition 2.4.10 (Ordonnancement stationnaire)**.** *On dira qu'un graphe possède un ordonnancement stationnaire,* ssi *il admet un ordonnancement fini pour lequel le nombre d'éléments en attente dans les canaux du CSDF est le même avant et après son exécution.*

Un ordonnancement stationnaire peut être exécuté un nombre infini de fois en mémoire bornée.

Le nombre d'éléments échangés sur tout canal pour une exécution de l'ordonnancement est fini puisque c'est un ordonnancement fini. Donc une exécution de l'ordonnancement est possible en mémoire bornée. Puisque le nombre d'éléments sur les canaux est invariant par exécution de l'ordonnancement, le fait de répéter plusieurs fois l'ordonnancement ne nécessite pas plus de mémoire que s'il était exécuté une seule fois.

Définition 2.4.11 (Consistance)**.** *Un CSDF est consistant lorsqu'il admet un ordonnancement stationnaire non vide.*

Lee et Messerschmitt donnent une condition nécessaire et suffisante pour déterminer la consistance d'un CSDF. Nous rappelons succinctement leur résultat ci-dessous.

Soit un CSDF composé des nœuds $\{N_1, \ldots, N_L\}$ et des arcs $\{C_1, \ldots, C_M\}$. Supposons l'existence d'un ordonnancement stationnaire. Appelons r_{N_j} le nombre d'apparitions de N_j dans cet ordonnancement.

Soit un arc C_k reliant les nœuds N_u et N_v, au cours d'une exécution de l'ordonnancement, N_u produit $r_{N_u}.prod(N_u)$ éléments et N_v consomme $r_{N_v}.cons(N_v)$ éléments. Si l'ordonnancement est stationnaire alors

$$r_{N_u}.prod(N_u) = r_{N_v}.cons(N_v)$$

Si on écrit la précédente équation pour tous les canaux du graphe CSDF, on obtient un système d'équations $\mathbf{\Gamma}.\vec{r} = 0$ où l'inconnue est $\vec{r} = [r_{N_1}, \ldots, r_{N_L}]$. On remarque que le vecteur nul est toujours solution de ce système, en effet l'ordonnancement vide est stationnaire.

Définition 2.4.12 (Vecteur de répétition)**.** *On appellera les solutions non nulles du système* $\mathbf{\Gamma}.\vec{r} = 0$ *des vecteurs de répétition.*

Théorème 2.4.4 (Lee, Messerschmitt et Bilsen)**.** *Un CSDF est consistant si* $\mathbf{\Gamma}$ *est de rang* $L - 1$.

Le cas échéant étant donné un vecteur de répétition $\vec{r}$*, le plus petit vecteur de répétition non nul est :*

$$\vec{q} = \{\frac{r_1}{s}, \ldots, \frac{r_{N_M}}{s}\}$$

avec $s = pgcd(r_1, \ldots, r_{N_L})$*. On dira que* $\vec{q}$ *est le vecteur de répétition minimal.*

Définition 2.4.13 (Éléments échangés sur un arc)**.** *Le nombre d'éléments échangés sur le canal* C_k *sur l'ordonnancement stationnaire minimal est noté,* $\beta(C_k) = q_{N_u}.prod(N_u) = q_{N_v}.cons(N_v)$.

Le théorème ci-dessus donne une condition nécessaire et suffisante à l'existence d'un ordonnancement stationnaire. Rien ne garantit cependant, en cas de cycles, que l'ordonnancement ne produise pas un interblocage comme défini ci-dessous.

Définition 2.4.14 (Interblocage)**.** *Un ordonnancement produit un interblocage* ssi *lors de l'exécution d'un nœud il n'y a pas assez d'éléments dans les canaux entrants pour satisfaire ses taux de consommation.*

Définition 2.4.15 (Vivacité)**.** *Un CSDF est vivace* ssi *il admet un ordonnancement exécutable un nombre infini de fois sans interblocage.*

Définition 2.4.16 (Programme correct)**.** *Un CSDF est correct s'il est consistant et s'il est vivace.*

Pour montrer la correction d'un CSDF il suffit de montrer sa consistance, et de montrer qu'il existe un ordonnancement stationnaire qui n'introduit pas d'interblocages. Comme les ordonnancement stationnaires conservent le nombre d'éléments dans les files des canaux, il suffit de vérifier l'absence d'interblocages sur *une et une seule* exécution de l'ordonnancement.

2.4.2 Expressivité du langage SJD

Maintenant que nous avons défini concrètement la sémantique des CSDF et donc des SJD, nous pouvons nous demander quels types de réorganisation de données le langage SJD peut exprimer. Comme le remarque Thies dans [Thi09], les programmes qu'exécutent les nœuds `Filter` sont Turing-complets ; il en découle que SJD et StreamIt peuvent exprimer, à l'aide d'un filtre, toute réorganisation de données qui est programmable dans une machine de Turing. Fort de ce constat, Thies s'intéresse à *l'expressivité du graphe*, c'est-à-dire l'expressivité obtenue en l'absence de filtres. Dans cette section on parlera donc de l'expressivité obtenue par la seule utilisation des nœuds de routage (`Split`, `Join`, `Sink`, `Constant`).

Arrangement avec répétition Un arrangement avec répétition est un tirage de m éléments parmi n éléments avec remise (le même élément peut-être choisi plusieurs fois). L'ordre des tirages est important. Plus formellement, les arrangements avec répétition de l'ensemble $[1, 2, \ldots, n]$ sont les ensembles $[\phi(1), \phi(2), \ldots, \phi(m)]$ où ϕ est une application de $[1, 2, \ldots, m]$ dans $[1, 2, \ldots, n]$.

Théorème 2.4.5. *Expressivité de SJD Les graphes SJD composés uniquement de nœuds de routage réalisent exactement tous les arrangements avec répétition des données entrantes.*

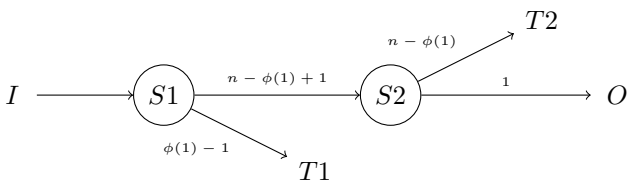
Pour montrer ce théorème , nous allons montrer que tout arrangement avec répétition peut être réalise par un graphe SJD (lemme 2.4.1) et que tout graphe SJD réalise un arrangement avec répétition (lemme 2.4.3).

Lemme 2.4.1. *Soit un flux de taille finie n, $I = [a_1, a_2, \ldots, a_n]$. Soit O un quelconque arrangement avec répétition de I, $O = [a_{\phi(1)}, a_{\phi(2)}, \ldots, a_{\phi(m)}]$.*

Il est possible de construire un graphe composé exclusivement de nœuds de routage consistant et vivace qui produit O à partir du flot I sur un ordonnancement stationnaire.

Démonstration. Par induction sur m :

- $m = 1$ et $O = [a_{\phi(1)}]$ et $\phi(1) \in [1, \ldots, n]$. Le graphe suivant,



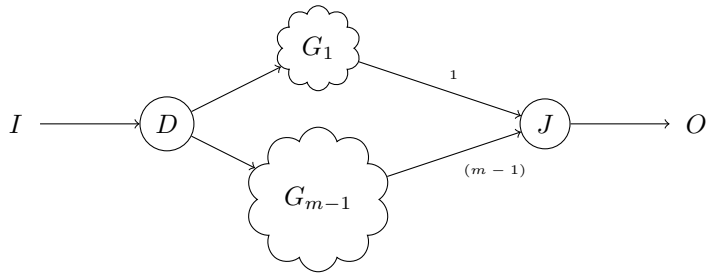
  est consistant et sans interblocages, il possède un ordonnancement stationnaire minimal $n \to 1$ (obtenu pour le vecteur de répétition $[1, 1, 1, 1, 1, 1]$).

  L'unique élément produit lors de l'exécution de l'ordonnancement stationnaire minimal est l'élément $a_{\phi(1)}$. En effet le premier Split $S1$ élimine les premiers $\phi(1) - 1$ éléments, et le deuxième Split $S2$ produit l'élément $a_{\phi(1)}$ et élimine les éléments restants.

  La condition initiale est donc vérifie.

- Pour $m > 1$, O s'écrit sous la forme $[a_{\phi(1)}, \ldots, a_{\phi(m-1)}, a_{\phi(m)}]$. Par hypothèse de récurrence il existe un graphe G_{m-1} consistant, sans interblocages et composé uniquement des nœuds de routage qui produit $[a_{\phi(1)}, \ldots, a_{\phi(m-1)}]$. De la même manière il existe un graphe G_1 qui à partir de I produit $[a_{\phi(m)}]$.

  Construisons le graphe G suivant,



  Ce graphe est sans interblocages car G_1 et G_{m-1} n'en avaient pas et on n'a pas introduit de cycles. G est également consistant (chaque nœud est exécuté exactement une fois dans l'ordonnancement stationnaire minimal) et il produit bien la suite de valeurs attendues.

□

Lemme 2.4.2. *(Généralisation) Soit $I_1, I_2, \ldots, I_p$ flux de taille finie. Soit $O_1, O_2, \ldots, O_q$ arrangements avec répétitions des éléments de $I = \bigcup_{i=0}^{p} I_i$.*

Il existe un graphe composé uniquement de nœuds de routage, consistant et vivace, qui produit $O_1, O_2, \ldots, O_q$ à partir des flots $I_1, I_2, \ldots, I_p$ sur un ordonnancement stationnaire.

Démonstration. Formons une suite O en concaténant (dans l'ordre) les suites $O_1, O_2, \ldots, O_q$. Puisque chacune des suites O_i est un arrangement avec répétitions des éléments de I, alors O est lui même un arrangement avec répétitions de I. En appliquant le lemme 2.4.1, nous savons qu'il existe un graphe G consistant, sans interblocages, qui à partir de I produit O sur un ordonnancement stationnaire.

Maintenant on crée un nœud `Join` J qui rassemble toutes les entrées de I_1 à I_p. On crée également un nœud `Split` S qui alimente toutes les sorties de O_1 à O_q. Nous observons que la sortie après une exécution de J forme la suite des éléments de I. Il suffit de connecter donc J avec G, et de distribuer avec S la sortie de G sur les différents O_i. □

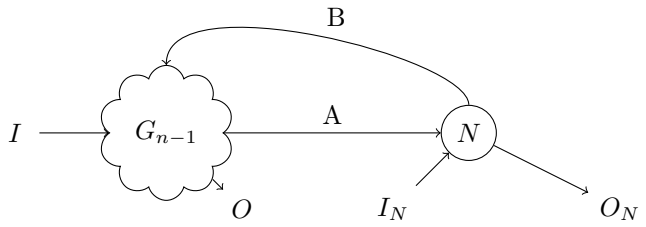
Lemme 2.4.3 (Réciproque)**.** *Soit un graphe G consistant et vivace formé uniquement de nœuds de routage. De plus, supposons que lors d'une exécution d'un ordonnancement stationnaire, G consomme sur chaque `Input` i la suite I_i d'éléments et produit sur chaque `Output` j la suite O_j d'éléments.*

Alors pour tout j, O_j est un arrangement avec répétition de $I = \bigcup_{j=0}^{p} I_j$.

Démonstration. Par induction sur le nombre de nœuds $|G|$ du graphe G.

- $|G_1| = 1$, on vérifie facilement pour les nœuds `Split`, `Join`, `Duplicate` que les sorties sont un arrangement avec répétition des entrées. Lorsqu'il n'y a pas de sorties (nœud `Sink`) on admet que la trace vide $\{\}$ est un arrangement avec répétition de tout ensemble, et en particulier de I.

- $|G_n| = n$, soit un nœud N quelconque de G. Nous pouvons réécrire G_n sous la forme suivante :



Sans perte de généralité nous supposons que G_{n-1} possède une entrée I et une sortie O. Pour couvrir tous les types de nœud, nous supposons que N possède une entrée interne A, une entrée externe I_N, une sortie interne B et une sortie externe O_N.

Par hypothèse, G_n est consistant et sans interblocages ; il admet un ordonnancement minimal stationnaire. Soient A^1 (resp. B^1) les données consommées sur A (resp. produites sur B) lors de la première exécution de N dans cet ordonnancement. Par monotonicité des KPN, $A^1 \sqsubseteq G_{n-1}(I)$. Or par hypothèse de récurrence $G_{n-1}(I)$ est un arrangement avec répétition de I, donc son préfixe A^1 est aussi un arrangement avec répétition de I.

En appliquant l'hypothèse de récurrence au sous-graphe de taille 1 composé de l'unique nœud N, on montre que $B^1 \sqsubseteq N(I_N, A^1)$ est un arrangement avec répétition de $I \cup I_N$.

De la même manière, à la k^{e} exécution de N, celui ci consomme,

$$A^k \sqsubseteq G_{n-1}(I, B^1, \ldots, B^{k-1})$$

Par induction sur k on montre que tous les A^k et B^k sont des arrangements avec répétitions de $I \cup I_N$. Il s'ensuit que O et O_N sont également des arrangements avec répétitions de $I \cup I_N$.

□

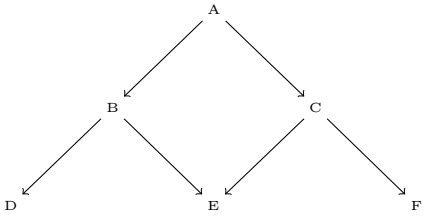
Par sa structure série-paralèlle et le manque de nœuds `Sink`, le langage StreamIt ne permet pas en utilisant uniquement les nœuds de routage d'exprimer tous les arrangements avec répétition. Il est par exemple impossible d'inverser les paires d'éléments consécutives dans un flux. Il est cependant possible, bien entendu, d'exprimer ces réorganisations dans StreamIt en utilisant des nœuds filtres (dont les programmes sont Turing-complets) ; mais dans ce cas l'information sur le transfert des données devient opaque pour le compilateur.

Il est également possible de transformer un graphe arbitraire en un graphe série-parallèle en ajoutant dans StreamIt des nœuds `Sink`. C'est ce qui est fait dans le programme `DES` de la suite de benchmarks StreamIt[Gro]. Pour cela un nœud `Sink` est fabriqué à l'aide d'un filtre à une entrée et pas de sorties ; ce qui permet de décimer les données.

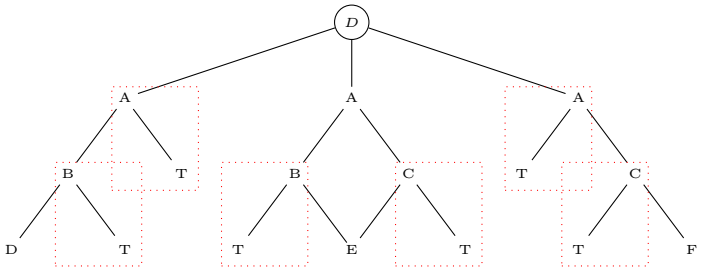
Cette transformation est néanmoins problématique pour certains graphes arbitraires qui subissent une explosion combinatoire du nombre des nœuds lors de la transformation en graphes série-parallèle. Par exemple pour rendre série-parallèle le graphe en figure 2.13(a) en introduisant des faux nœuds `Sink`, il faut dupliquer le nœud A trois fois. Puis sur chacune des instances de A, on ne conservera que les sorties dont on a besoin pour le calcul de D, E et F.

2.5 Langage SLICES

Le langage SJD est très expressif et les nœuds de routage nous permettent d'écrire toutes les réorganisation de données statiques finies. Cependant le langage est très verbeux et peu naturel. Comme nous avons pu constater, les graphes de réorganisation de données pour le filtre Gaussien ou la multiplication matricielle sont difficiles à écrire. De plus le programmeur doit calculer à la main les consommations et productions souvent complexes ce qui est source d'erreurs.



(a) Graphe arbitraire



(b) Transformation en graphe série-parallèle

FIGURE 2.13 – Transformation d'un graphe arbitraire en graphe série-parallèle, les faux nœuds `Sink` sont en pointillées rouges. Dans cet exemple le nombre de nœuds double lors de la transformation.

Nous proposons donc un langage de haut-niveau, SLICES, qui facilite l'expression de réorganisations de données régulières, à travers la manipulation de formes et de motifs glissants. Le langage est construit autour de cinq primitives :

Shape Le constructeur `shape` déclare une vue multidimensionnelle d'un flux brut.

Grid Le constructeur `grid` déclare une grille, c'est-à-dire, un maillage de points d'ancrage sur les données d'une vue.

Block Le constructeur `block` permet de sélectionner des voisinages autour de chaque point d'ancrage d'une grille.

Zip Le constructeur `zip` permet d'entrelacer les données de deux flux.

Iterateur Les itérateurs permettent d'entrelacer les données provenant de plusieurs flux.

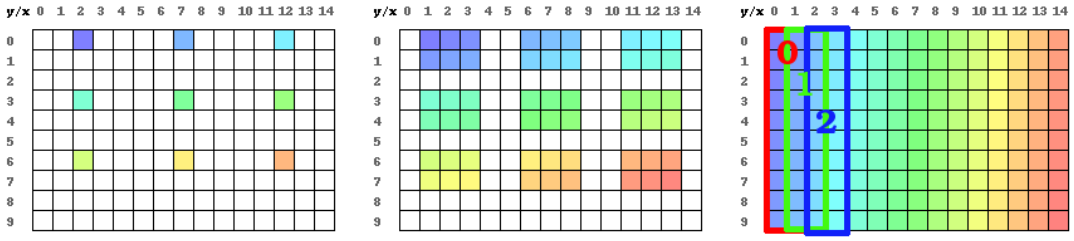
Ces constructeurs peuvent être assemblés pour décrire des découpage de données complexes. Les combinaisons possibles sont celles autorisées par le système de types de SLICES que l'on introduira en § 2.5.6. Le système de types garantit que toutes les programmes SLICES correctement typés correspondent à une réorganisation de données légale.

Cette section décrit plus en profondeur les résultats publiés dans [dOCLB10a].

2.5.1 Shape

SLICES travaille sur des données multidimensionnelles. Le langage restructure les flux de données en entrée dans des tableaux en utilisant des types *shape* paramétrés. La conversion d'un flux vers un type *shape* crée une vue multidimensionnelle des données du flux ; par exemple

```
shape[10] A = brut.input
shape[15,10] B = brut.input
shape[3,3,3] C = brut.input
```



(a) `B[2 :15 :5,0 :8 :3]` (b) `B[2 : :5, :8 :3] x (-1 :1, 0 :1)` (c) `B[ : :,0 :1 :] x (0 :1,0 :9)`

FIGURE 2.14 – Ensemble de points décrits par, (a) une grille de type *grid*[2], (b) des blocs sur une grille de type *blocks*[2], (c) des blocs *blocks*[2] dont les blocs successifs se superposent. Le gradient de couleurs indique l'ordre d'itération sur les éléments (les couleurs froides sont les premières).

`brut.input` est le flots de données brut correspondant à l'arc SJD nommé `brut`, nous montrerons par la suite (§ 2.5.7) comment nommer les flux brut SJD. On crée trois vues du flot avec des types *shape* différents. Le flot input est vu en A comme une suite de vecteurs de longueur 10, en B comme une suite de matrices de taille 15×10, finalement en C comme une suite de cubes de taille $3 \times 3 \times 3$.

En généralisant, un type `shape`$[s_1, \ldots, s_d]$ voit un flux comme une série de parallélotopes droits à d dimensions et dont les longueurs des côtés sont les s_i. Dans la vue `shape`$[s_1, \ldots, s_d]$, le point de coordonnées $(x_1, \ldots, x_d)$ du premier parallélotope correspond à la position dans le flux d'origine calculée par la formule :

$$\sum_{i=1}^{d} x_i * \prod_{j=1}^{i-1} s_j$$

Pour le n-ième parallélotope, il suffit de rajouter la taille des $(n-1)$ premiers parallélotopes soit $(n-1).\prod_{i=1}^{d} s_i$.

2.5.2 Grids

Sur des types *shape* nous pouvons appliquer l'opérateur de grille, qui est défini pour chaque dimension i du type shape par trois paramètres (l_i, h_i, δ_i) :

- l_i représente la première coordonnée de la grille sur la dimension i ;
- δ_i représente le pas de la grille sur la dimension i ;
- h_i représente la borne supérieure de la grille sur la dimension i (selon le pas, elle peut ne pas être atteinte).

La grille sélectionne donc sur chaque dimension l'ensemble des points compris entre l_i et h_i et espacés de δ_i, c'est-à-dire l'ensemble :

$$G_i = \{\delta_i.k.\vec{e_i} : \forall k \in [|\frac{l_i}{\delta_i}; \frac{h_i}{\delta_i}|]\}$$

Les éléments de la grille sont construits en effectuant le produit cartésien [2] des G_i :

$$G = G_1 \otimes \cdots \otimes G_d$$

2. Nous notons $\otimes$ le produit cartésien, pour ne pas le confondre avec l'opérateur $\times$ de SLICES (§ 2.5.3).

puis en classant les points d'ancrage par ordre lexicographique (p. ex. en deux dimensions les points sont pris de gauche à droite puis de haut en bas). Cet ordre détermine un itérateur de type grille $G(n)$, où $G(0)$ est le premier élément, $G(1)$ le second, et ainsi de suite.

Pour déclarer les paramètres d'une grille nous utilisons une notations par tranches, qui s'inspire de celle utilisée dans Python et Matlab. Les paramètres l_i, h_i, δ_i sont séparés par le signe deux-points et les dimensions sont séparées par des virgules, $[l_1 : h_1 : \delta_1, \ldots, l_d : h_d : \delta_d]$.

Pour appliquer une grille sur un type *shape*, il suffit de postfixer celui-ci par l'opérateur de grille. Par exemple l'application de la grille $[2 : 15 : 5, 0 : 8 : 3]$ sur le *shape* B, se note `B[2 :15 :5,0 :8 :3]` et décrit l'ensemble des points de la figure 2.14(a). Si les dimensions de la grille ne correspondent pas aux dimensions du type *shape* sur lequel elle est appliquée, une erreur de type est levée à la compilation.

Pour faciliter l'écriture, il est possible d'omettre un ou plusieurs des paramètres d'une grille. Les valeurs omises sont remplacées par des valeurs par défaut en suivant ces règles :

- si l_i est absent, il est remplacé par 0 ;
- si h_i est absent, il est remplacé par s_i (la taille de la i^{e} dimension du type *shape* sous-jacent) ;
- si δ_i est absent, il est remplacé par 1.

2.5.3 Blocs

L'opérateur de bloc s'applique uniquement sur un type grille. Un bloc est une boîte (parallélotope droit) de dimension d. Sur chaque dimension, le bloc est paramétré par une extension à gauche a_i et une extension à droite b_i : $(a_1 : b_1, \ldots, a_d : b_d)$ avec $a_i, b_i \in \mathbb{Z}$

Par exemple $(-1 : 1, 0 : 1)$ définit un bloc de dimension 2 et de taille 3×2. Ce bloc décrit l'ensemble ordonné de vecteurs $\{(-1, 0), (0, 0), (1, 0), (-1, 1), (0, 1), (1, 1)\}_{lex}$. Le suffixe *lex* indique que les vecteurs de l'ensemble sont ordonnées selon un ordre lexicographique. On ne peut utiliser un bloc à lui tout seul, il faut l'appliquer à une grille de dimension compatible en utilisant l'opérateur produit ($\times$). Par exemple,

```
B[2 : :5 , :8 :3] x (-1 :1,0 :1)
```

décrit les points sur la figure 2.14(b). Si un bloc n'a pas la même dimension que la grille à laquelle on l'applique, une erreur de type est levée à la compilation. Un bloc définit l'ensemble des points obtenu en centrant le bloc sur chaque point de la grille et en prenant les points résultant de la somme des vecteurs du bloc au point d'ancrage de la grille. Le bloc est ainsi estampillé autour de chacun des points d'ancrage. Formellement, les points sélectionnés, sont donnés par l'itérateur sur des ensembles ordonnés suivant :

$$GB(n) = \{g + b : \forall b \in B\}_{lex}$$

On peut avoir des recouvrements entre les blocs successifs, par exemple,

```
B[ : :,0 :1 :] x (0 :1,0 :9)
```

extrait toutes les paires adjacentes de colonnes de A (cf. figure 2.14(c)).

Quand une partie (voir la totalité) d'un bloc est en dehors des frontières définies par le type *shape* sous-jacent, une valeur de remplissage sera produite. Si elle n'est pas configurée la valeur de remplissage par défaut est 0, mais elle peut-être changée au niveau du type shape avec la primitive `padding(B) = -1`, qui ici change la valeur de remplissage pour tous les blocs qui seraient extraits sur des grilles de `B` à -1.

Si une des valeurs a_i ou b_i est omise dans un des 2-uplets d'un bloc, on prendra par défaut $a_i = 0$ et $b_i = s_i$, comme pour les grilles.

2.5.4 Itérateurs

Nous verrons dans la § 2.5.6 que *shape*, *grid* et *blocks* sont des sous-types du type *iterator* qui représente les suites d'éléments avec un ordre. Une instance *grid* peut être vue comme l'itérateur qui retourne les points sélectionnés par la grille dans l'ordre lexicographique. Une instance *blocks* peut être vue comme l'itérateur qui retourne des ensembles ordonnés de points correspondant à chaque estampillage du bloc sur les points d'ancrage la grille.

Différentes instances de type *iterator* peuvent être combinées de manière à changer l'ordre des éléments dans un flux, en utilisant les primitives `for`, `in` et `push`. La construction `for in` permet d'itérer sur les éléments d'un itérateur donnée. Le mot-clé `push` produit l'élément ou l'ensemble ordonné d'éléments courants sur un des flux sortants.

```
shape[3] D = DI.input
shape[2] E = EI.input
for d in D:
  for e in E:
    push e
    push d
```

produit les éléments

$$\{E_0, D_0, E_1, D_0, E_0, D_1, E_1, D_1, \ldots\}$$

2.5.5 L'opérateur zip

Un nid de boucles `for` permet de réaliser le produit cartésien des éléments de deux itérateurs. Lorsque l'on veut entrelacer les éléments de plusieurs itérateurs deux à deux on utilisera l'opérateur `zip`. L'opérateur `zip` est polymorphe : il s'applique aussi bien aux itérateur de type *iterator* qu'aux ensembles ordonnés de type *orderedset*.

Par exemple, `zip(A,B)` entrelace les éléments des opérandes, en créant un nouvel itérateur ou ensemble ordonné qui alternativement retourne un élément de `A` et un élément de `B` :

$$Z(n) = \begin{cases} A(\frac{n}{2}) & \text{if } n \equiv 0(mod.2) \\ B(\frac{n-1}{2}) & \text{if } n \equiv 1(mod.2) \end{cases}$$

2.5.6 Système de types

Les constructions du langage SLICES précédentes admettent un système de types strict qui assure que seuls les *programmes corrects* sont acceptés. Les programmes corrects doivent être construits à partir d'itérateurs ou d'ensembles ordonnés. Chaque itérateur ou ensemble ordonné, est obtenu à partir d'un assemblage de `shape`, `grid`, et/ou `block` de même dimension. Tout programme qui respecte le système de types ci-dessous, peut-être compilé vers un programme SJD ordonnançable en mémoire bornée et sans interblocages.

Les constructeurs `shape`, `grid` et `block` produisent des instances des types *shape*, *grid* et *blocks* respectivement. Ces trois types sous-classent le type abstrait *iterator*.

$$grid, blocks, shape \in iterator$$

Le constructeur `shape` crée une instance de type éponyme.

$$shape[s_1, \ldots, s_d] : shape[d]$$

Le constructeur `grid`, dont la syntaxe est donnée ci-dessous, s'applique à une instance de type *shape* et produit une instance de type *grid*.

$$[l_1 : h_1 : \delta_1, \ldots, l_d : h_d : \delta_d] : shape[d] \rightarrow grid[d]$$

Le constructeur `block` s'applique uniquement à une instance de type *grid* et produit une instance de type *blocks*.

$$(a_1 : b_1, \ldots, a_d : b_d) : grid[d] \rightarrow blocks[d]$$

Finalement, les boucles `for` prennent une instance de type *iterator* (c'est-à-dire un objet *shape*, *grid* ou *blocks*) et produisent un type *orderedset*.

$$for.in : iterator \rightarrow orderedset$$

Les opérateurs `push` et `zip` sont polymorphes :

$$\begin{aligned} push &: iterator \rightarrow IO \\ push &: orderedset \rightarrow IO \\ zip &: iterator, iterator \rightarrow iterator \\ zip &: orderedset, orderedset \rightarrow orderedset \\ zip &: iterator, orderedset \rightarrow orderedset \\ zip &: orderedset, iterator \rightarrow orderedset \end{aligned}$$

Le type IO retourné est un type factice qui représente l'effet de bord induit par `push` lorsqu'il écrit son opérande sur un canal sortant.

2.5.7 Intégration de SLICES et SJD

SLICES est spécialisé dans la description haut-niveau de découpages de données. Pour pouvoir écrire des programmes complets il faut donc l'intégrer à un langage de stream généraliste. Dans cette section nous allons montrer comment intégrer SLICES à SJD en utilisant les notations de ΣC.

Nous avons vu que ΣC définit un agent particulier de type `subgraph` (§ 2.3.5) qui permet de décrire des sous-graphes d'agents. Nous proposons un nouveau constructeur `slices` qui étend le composant `subgraph`. Le nouveau constructeur conserve les sections `interface` et `map` qui définissent les entrées/sorties, leurs interconnexions et les types de données utilisés ; il possède également un corps qui est écrit en SLICES :

```
slices MatrixMultSubGraph{
  interface{
    in<double> IA;
    in<double> IB;
    out<double> OC;
    out<double> O;
  }
  map {
    agent F = new DotProduct(y0);
    connect(OC, F.input);
    connect(F.output, O);
  }

  shape[x0,y0] A = IA.input
  shape[x1,y1] B = IB.input

  for l in A[0:1:,::] x (0:x0,0:0):
    for c in B[::,0:1:] x (0:0,0:y1):
      OC.push zip(l,c)
}
```

On a ici déclaré un sous-graphe `MatrixMultSubGraph`. La section `interface` déclare deux entrées brutes `IA` et `IB` avec des éléments de type flottant et une sortie brute `OC` de même type. Le corps de l'agent est lui remplacé par du code SLICES qui peut faire référence aux canaux entrants et sortants. Nous commenterons le corps cet agent (utilisé pour la multiplication matricielle) dans la section suivante. Finalement, la section `map` connecte la sortie du sous-graphe SLICES à un agent qui calcule produit scalaire entre les vecteurs de A et de B.

On montrera en § 2.6 qu'il est possible de compiler SLICES vers SJD. Cela permet au frontend de notre chaîne de remplacer chaque agent `slices` avec un agent équivalent `subgraph` qui possède la même interface mais dont le comportement est décrit en ΣC.

2.5.8 Exemples

Voyons maintenant comment s'écrivent les exemples du § 2.1. SLICES sera utilisé pour décrire les réorganisations de données, et la partie du programme chargée du calcul sera décrite en ΣC.

Filtre gaussien Le filtre gaussien peut-être séparé en deux étapes : la première extrait des blocs de taille 3×3, la deuxième, constituée d'un nœud `Filter`, calcule la convolution. Ci-dessous nous montrons le programme SLICES qui décrit la première étape.

```
shape[w,h] I = image.input
for block in I[::,::] x (-1:1,-1:1):
  push block
```

La première ligne du programme, déclare un type *shape* de la taille de l'image à partir du flux brut `image`. Pour chaque pixel de l'image nous souhaitons extraire les blocs formés par les 8 pixels voisins et le pixel courant. Nous prenons une grille avec tous les points de l'image comme points d'ancrage avec `I[ : :, : :]`. Sur chaque point d'ancrage nous estampillons un bloc 3×3 centré avec `x (-1 :1,-1 :1)`. Puis à l'aide d'une boucle `for` nous itérons sur l'ensemble des blocs et nous les produisons sur la sortie. Le programme entier avec filtre de calcul, est donné dans l'extrait de code 2.2.

Filtre de Sobel Le filtre de Sobel est également construit comme une convolution, on peut donc reprendre le code utilisé pour le filtre gaussien. Cependant le filtre de Sobel est séparable, il est possible de réaliser la convolution ligne par ligne, puis colonne par colonne.

L'extraction des blocs de taille (3×1) sur chaque ligne peut être réalise avec le code suivant,

```
shape[w,h] I = image.input
for triplet in I[::,::] x (-1:1,0:0)
  push triplet
```

L'extraction des blocs de taille (1×3) sur chaque colonne se fait en deux étapes, tout d'abord on extrait les colonnes avec le code suivant,

```
shape[w,h] I = image.input
for column in I[::,0:1:] x (0:0,:):
  columns.push column
```

puis on extrait les blocs au sein de chaque colonne,

```
shape[h] I = columns.input
for triplet in I[::] x (-1:1):
  push triplet
```

Multiplication de Matrices La multiplication de matrices est aisée à décrire en SLICES. On souhaite extraire les lignes de A, les colonnes de B, puis itérer sur toutes les combinaisons de lignes et de colonnes.

```
slices ExtractBlocks(int w, int h) {
  interface{
    input<int> image;
    output<int> blocs;
  }
  shape[w,h] I = image.input
  for block in I[::,::] x (-1:1,-1:1):
    blocs.push block
}

agent Convolve(int w, int h) {
  interface {
    input<int> Blocs;
    output<int> Result;
  }
  map {
    slices E = new ExtractBlocks(w, h);
    connect(E.blocs, self.blocs)
  }
  const kernel[] = {1,2,1,2,3,2,1,2,1};
  void start() exchange(Blocs[] blocs[9], Result[] result) {
    result = 0;
    for (int i=0;i<9;i++) {
      result += blocs[i]*kernel[i];
    }
  }
}
```

Extrait de code 2.2 – Programme SLICES/ΣC qui implémente le filtre gaussien.

```
shape[x0,y0] A = IA.input
shape[x1,y1] B = IB.input

for l in A[0:1:,::] x (0:x0-1,0:0):
  for c in B[::,0:1:] x (0:0,0:y1-1):
    push zip(l,c)
```

On commence par déclarer les dimensions des deux entrées avec le constructeur `shape`. L'entrée `A` est vue comme un flot de matrices de $\mathcal{M}_{x_0,y_0}$ et l'entrée `B` est vue comme un flot de matrices de $\mathcal{M}_{x_1,y_1}$.

Puis on étend chaque point de la première colonne de A avec un bloc d'extension x_0, ce qui produit l'ensemble des lignes de A. De la même façon, on étend chaque point de la première ligne de B avec un bloc d'extension y_1, ce qui produit l'ensemble des colonnes de B. On itère enfin sur le produit cartésien des lignes et des colonnes avec le nid de boucles.

Le `zip` est utilisé car le filtre en aval attend que les lignes et colonnes soient entrelacées ; si le filtre consommait d'abord une ligne puis une colonne, on l'aurait remplacé par

```
push l
push c
```

Transformation de Fourier Rapide Nous allons utiliser SLICES pour décrire l'étage de séparation des données paires et impaires. Cette opération est très facile à décrire,

```
shape[N] I = I.input
for pair in I[::2]:
  paires.push pair
for impair in I[1::2]:
  impaires.push impair
```

On selectionne d'abord les données paires, en superposant une grille de pas 2. Puis, pour les données impaires, on décale la grille de 1.

Bien sûr la FFT nécessite de répéter cet étage $log(N)$ fois, cette répétition peut-être décrite dans ΣC qui va instantier autant d'étages SLICES que nécessaire.

Filtre de Hough L'extraction des couples formés par les angles et les points nécessaires pour implémenter les itérations du filtre de Hough peut être écrites en SLICES avec le code suivant,

```
shape[2,w*h] points = points.input
shape[alpha] angles = angles.input
for a in angles:
  for p in points[0:1:,::]x(:,0:0):
    push a
    push p
```

On déclare l'entrée brute `points`, comme un ensemble de $w \times h$ couples de coordonnées. L'entrée `angles` est vue tout simplement comme la suite des valeurs échantillonnées. On selectionne avec un bloc les coordonnées de p et on effectue le produit cartésien des angles et des coordonnées.

Les valeurs produites par cette réorganisation de données pourront alimenter un filtre impur ou pur (dans ce cas il faudra comme dans l'exemple SJD que le filtre possède un arc réentrant pour passer explicitement l'état d'une exécution à l'autre) qui calculera la transformée de Hough.

2.5.9 Expressivité du langage SLICES

Bien que plus haut-niveau, SLICES n'est pas aussi expressif que SJD. En effet il n'est pas possible avec SLICES d'exprimer tous les arrangements avec répétition. Parmi les arrangements qui ne peuvent pas être écrits en SLICES on trouve :

- des parcours de données complexes, comme des parcours en zigzag ou avec des sauts non réguliers entre blocs consécutifs ;
- l'extraction de motifs non rectangulaires (il est cependant possible d'utiliser SLICES pour extraire le rectangle englobant du motif et ne conserver que les points désirés).

Les parcours complexes de données devront donc être exprimés dans le langage SJD.

2.6 Compilation de SLICES vers SJD

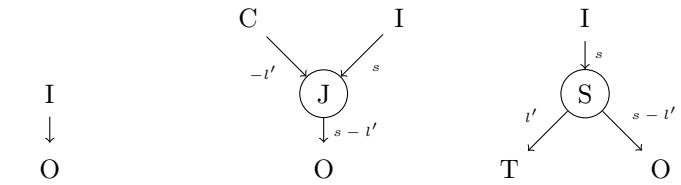
Cette section présente le processus de compilation du langage SLICES vers le langage SJD. On montrera d'abord comment extraire l'ensemble de points décrit par les types $blocks[1]$ de dimension 1 en utilisant un graphe SJD. Puis nous montrerons qu'il est possible de construire le graphe SJD qui produit un $blocks[n]$ à partir de n graphes $blocks[1]$. Enfin, nous montrerons comment compiler des nids de `for in push` et la primitive `zip`.

2.6.1 Compilation des blocks de dimension 1

Nous observons que $[l : h : \delta]$ est équivalent à $[l : h : \delta] \times (0 : 0)$; autrement dit les points décrits par une grille de dimension 1 sont les points décrits par un bloc d'extension nulle appliqué à cette grille. On considérera donc les types *grid* comme des cas particuliers de *blocks*.

On sépare l'extraction de $blocks[1] \equiv [l : h : \delta] \times (a : b)$ en deux étapes :

- D'abord on extrait du flot brut l'intervalle qui contient l'ensemble des blocs que l'on veut extraire. Cet intervalle, appelé par la suite région, correspond aux points définis par $[l' : h' : 1]$ avec $l' = l - a$ la coordonnée du premier élément du premier bloc et $h' = l' + \delta.((h-l)\text{div. } \delta) + b$ la coordonnée du dernier élément du dernier bloc.



(a) $l' = 0$ (b) $l' < 0$ (c) $l' > 0$

FIGURE 2.15 – Les différents graphes SJD utilisés pour ajuster la borne inférieure d'une région en fonction de la valeur de l'.

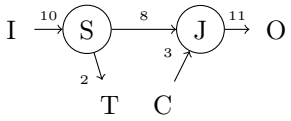


FIGURE 2.16 – Graphe généré pour extraire une région lorsque $l' = -3$, $h' = 8$ et $s = 10$.

- Une fois obtenue la région contenant les blocs désirés, nous extrayons les blocs $[:: \delta] \times (0 : w)$, où $w = b - a + 1$ est la largeur de ces derniers.

Parce qu'un flot de données peut contenir plusieurs (voir une infinité) de motifs, il est important que les graphes SJD compilés puissent être appelés plusieurs fois à la suite. Pour ce faire nous avons généré des graphes dont l'ordonnancement stationnaire minimal travaille exactement sur la taille des motifs entrants. Autrement dit à la fin du traitement d'un motif, il n'y a pas d'éléments en attente de traitement sur les arcs du graphe ; il peut donc être réutilisé à nouveau sans effets de bord.

Extraction d'une région

Nous souhaitons extraire la région $[l' : h' :]$ à l'intérieur d'un type $shape[1]$ de taille s. Si la région est $[0 : s :]$, nous n'avons rien à faire. Dans les autres cas nous devrons soit couper certaines données (quand la région est plus petite que le `shape`), soit injecter des données de remplissage (quand la région déborde en dehors du `shape`). Il faut considérer ces deux possibilités pour la borne supérieure et inférieure. Nous détaillons ici le processus pour la borne inférieure,

- lorsque $l' = 0$, on ne fait rien (cf. figure 2.15(a)).
- lorsque $l' < 0$, on injecte $-l'$ éléments de remplissage en utilisant un nœud J et une source constante C (cf. figure 2.15(b)).
- lorsque $l' > 0$ on élimine les premiers l' éléments en utilisant un nœud S et un `Sink` T (cf. figure 2.15(c)).

Par exemple, pour $l' = -3$, $h' = 8$ and $s = 10$ nous obtiendrions le graphe de la figure 2.16.

Lorsque la région est sélectionnée, nous pouvons extraire la séquence de blocs désirés à l'intérieur de celle-ci. Les blocs de dimension 1 sont définis par w leur largeur et δ la translation entre le début de deux blocs. Nous pouvons distinguer parmi trois cas en fonction du rapport $\frac{w}{\delta}$:

- $\frac{w}{\delta} \leq 1$, les blocs ne se superposent pas ;
- $1 < \frac{w}{\delta} < 2$, les blocs se superposent partiellement ;

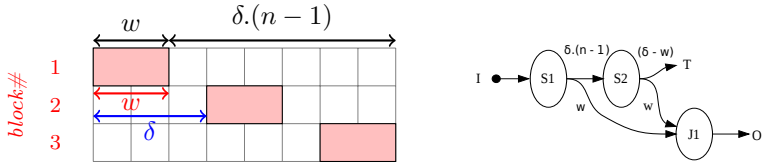


FIGURE 2.17 – Extraction des blocs sans recouvrement ($\frac{w}{\delta} \leq 1$)

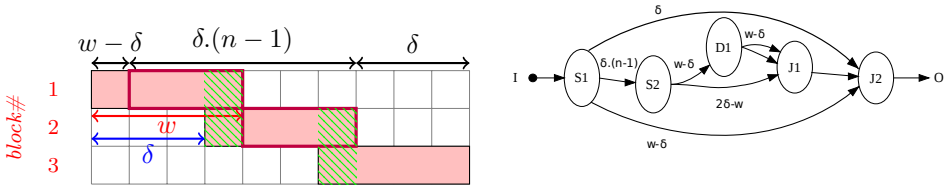


FIGURE 2.18 – Extraction des blocs avec recouvrement partiel ($1 < \frac{w}{\delta} < 2$)

- $2 \geq \frac{w}{\delta}$, les blocs se superposent totalement.

Nous traitons dans la suite chacun des cas séparément.

Extraction des blocs sans recouvrement (figure 2.17)

Ce premier cas est le cas le plus simple, on doit extraire une séquence de n blocs de taille w qui sont séparés par des trous de taille $\delta - w$ (positive car les blocs ne se recouvrent pas).

Il faut imaginer les points décrits comme une suite de blocs et de trous. Les données situées à l'emplacement des trous doivent être dirigées vers un nœud `Sink`. Les données situées à l'emplacement des blocs doivent être produites sur la sortie. Pour ce faire, nous extrayons le premier bloc avec $S1$. Après cela nous nous retrouvons avec $(n-1)$ éléments de la forme {bloc, trou}. Nous séparons les blocs des trous en utilisant la combinaison $(S2 + T)$.

Extraction des blocs avec recouvrement partiel (figure 2.18)

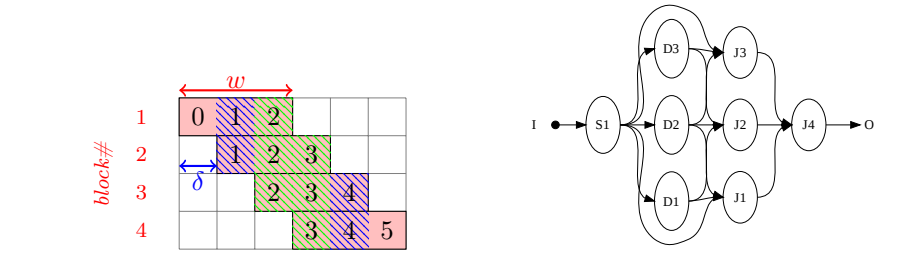
Dans ce deuxième cas, il faut extraire n blocs de taille w. On peut observer que chaque paire de blocs consécutifs se recouvre sur une longueur de $w - \delta$ éléments ; il faut donc dupliquer ces éléments là de manière à pouvoir donner une copie à chacun des blocs.

Comme dans le cas précèdent nous produisons tout d'abord $(w - \delta)$ éléments sur la sortie pour nous retrouver avec une suite de $(n-1)$ éléments identiques (ce sont les éléments entourés de rouge sur la figure 2.18).

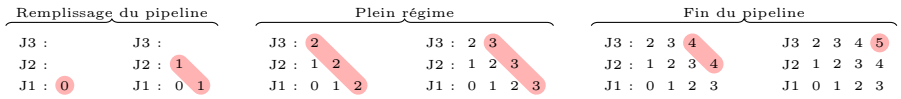
Avec $S2$ nous séparons la partie sans recouvrement et la partie avec recouvrement. Nous dupliquons les éléments de cette dernière en utilisant $D1$ et nous rassemblons le tout en utilisant $J1$. A la fin de ce processus il nous manque les δ éléments du dernier bloc que nous produisons sur le troisième arc entrant de $J2$ ce qui achève l'extraction dans le cas du recouvrement partiel.

Extraction des blocs avec recouvrement total (figure 2.19)

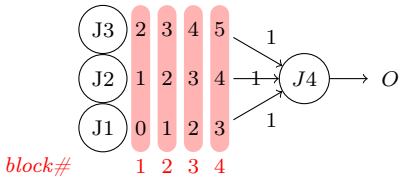
Dans ce troisième et dernier cas, il s'agit d'extraire n blocs qui sont totalement recouverts. Chaque élément peut appartenir à plusieurs blocs en même temps. Le nombre maximal de blocs qui se partagent un même élément est noté $m_{overlap} = min(\lceil \frac{w}{\delta} \rceil, n)$.



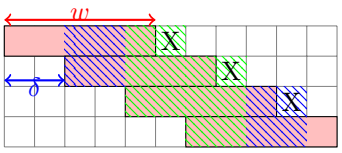
(a) Schéma d'extraction



(b) Pipeline pour l'extraction des blocs avec recouvrement total



(c) Production des blocs



(d) Éléments en trop lorsque $(w \bmod \delta) \neq 0$

FIGURE 2.19 – Extraction des blocs avec recouvrement total $(2 \leq \frac{w}{\delta})$

L'extraction des blocs avec recouvrement total, peut s'apparenter au remplissage de $m_{overlap}$ pipelines. Supposons, comme sur la figure 2.19, que $m_{overlap} = 3$. Puisque les blocs se déplacent sur des pas de δ, on peut décomposer les éléments à retourner en motifs de taille $\delta = 1$.

En un premier temps concentrons nous sur le nombre de copies nécessaires. On produit une fois le premier motif (l'élément 0), puis deux fois le deuxième motif (l'élément 1). Pour les $n - 2 \times (m_{overlap} - 1) = 2$ motifs suivants on va faire trois copies. Puis pour les deux derniers blocs nous ne produisons que deux copies et une copie respectivement.

Pour réaliser ces copies nous mettons en place le réseau de nœuds D et S de la figure 2.19. La couche de nœuds D se charge de dupliquer les éléments : $D1$ et $D3$ font les deux copies nécessaires pour le deuxième et avant-dernier motif, $D2$ fait les trois copies nécessaires pour tous les éléments centraux. Les copies sont alors redirigées sur trois nœuds J qui assemblent les éléments des trois étages du pipeline ci-dessus, $J1$, $J2$ et $J3$. L'exécution de ce réseau de nœuds peut-être vu comme le remplissage des pipelines de la figure 2.19(b). Sur cette figure on a représenté représenté les files d'entrée des nœuds `Join` au fur et à mesure de l'exécution des nœuds `Duplicate`.

Dans quel ordre faut-il produire les motifs ? Observons la figure 2.19(c) qui reprend le dernier état du pipeline ; mais considérons maintenant les éléments par colonnes. Les éléments des colonnes correspondent exactement aux blocs qu'il nous faut retourner. Il suffit donc de placer un nœud `Join` à $m_{overlap}$ entrées de consommation δ éléments et d'y connecter les étages $J1$,$J2$ et $J3$. Cette approche se généralise à toute valeur de $m_{overlap}$ et de δ.

Lorsque $(w \bmod \delta) \neq 0$, on peut produire des éléments en trop à la fin de chaque bloc. Par exemple sur la figure 2.19(d) les éléments marqués d'un X sont retournés alors qu'il ne devraient pas être sélectionnés. On pourrait éviter de sélectionner ces éléments mais cela nous obligerait à traiter spécialement la fin des blocs ce qui compliquerait le graphe résultant. On a préféré rediriger ces éléments en trop vers un nœud `Sink` par utilisation d'un combinaison de $S - T$ (de la même

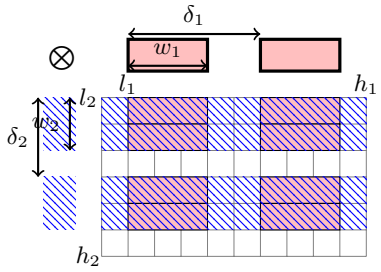


FIGURE 2.20 – L'extraction de régions multidimensionnelles peut se décomposer en deux extractions de régions unidimensionnelles.

manière que les éléments en trop étaient éliminés lors de l'extraction d'une région).

2.6.2 Compilation des blocks multidimensionnels

Maintenant que nous avons montré comment extraire des *blocks*[1], nous généralisons notre approche aux dimensions supérieures. Les grilles et blocs multidimensionnels sont par construction définis comme des produits cartésiens de grilles et blocs unidimensionnels (avec un ordre lexicographique). Par exemple le *blocks*[2],

$$[l_1 : h_1 : \delta_1, l_2 : h_2 : \delta_2] \times (a_1 : b_1, a_2 : b_2)$$

se décompose en,

$$\{([l_1 : h_1 : \delta_1] \times (a_1 : b_1)) \otimes ([l_2 : h_2 : \delta_2] \times (a_2 : b_2))\}_{lex}$$

comme sur la figure 2.20.

Nous allons exploiter cette propriété de composition pour compiler des graphes pour les *blocks*[d] en utilisant un ensemble de graphes *blocks*[1] :

1. On décompose *blocks*[d] dans ces composantes 1D, $([l_i : h_i : \delta_i] \times (a_i : b_i))$, avec $1 \leq i \leq d$.
2. On défini la taille repliée f pour la dimension dim comme

$$f(dim) = \prod_{i=1}^{dim-1} s_i \quad \text{avec } f(1) = 1$$

   C'est en fait le nombre d'éléments contenus dans les hyper-plans obtenus en coupant selon la dimension dim.

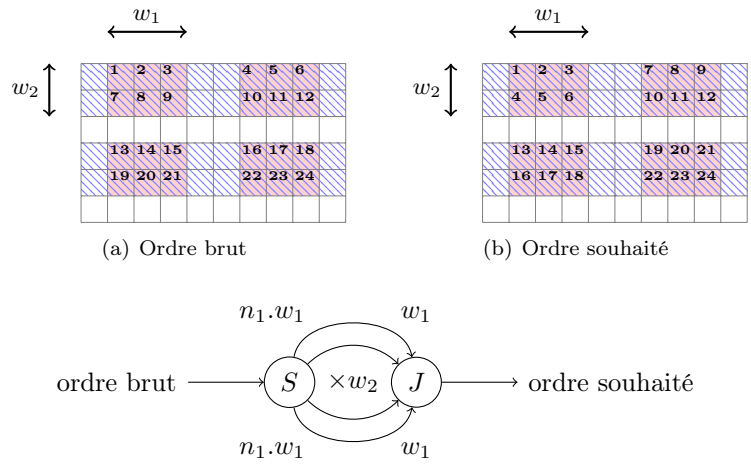
3. On compile pour chaque i, le graphe G_i qui produit les éléments dans,

$$[l_i.f(i) : h_i.f(i) : \delta_i.f(i)] \times (a_i.f(i) : b_i.f(i))$$

4. Nous connectons, bout à bout, les graphes G_i de manière à produire le graphe final G,

$$G \equiv G_d \rightarrow G_{d-1} \cdots \rightarrow G_0$$

Le graphe G obtenu extrait les blocs de dimension d souhaités. L'idée est que chaque G_i extrait la composante unidimensionnelle correspondant à la dimension i, mais les consommations sont modifiées de manière à travailler avec des motifs de taille $f(i)$. Par exemple sur la figure 2.20 G_2 extrait $([l_2 : h_2 : \delta_2] \times (a_2 : b_2))$ (dans la marge gauche), mais en considérant des motifs de taille



(a) Ordre brut

(b) Ordre souhaité

(c) Graphe SJD réalisant l'ordre souhaité

FIGURE 2.21 – Étage de tri pour ranger les blocs dans l'ordre de la grille.

s_1 (la longueur d'une ligne). Ce premier processus produit la région hachurée. Puis le graphe G_1 extrait, à l'intérieur de la région hachurée, les blocs $([l_1 : h_1 : \delta_1] \times (a_1 : b_1))$ (dans la marge supérieure) avec des éléments de taille 1, en générant ainsi les blocs bidimensionnels souhaités.

Il reste un problème à résoudre : le procédé ci-dessus n'extrait pas les éléments dans l'ordre lexicographique dicté par la grille. Pour remettre les blocs dans l'ordre nous utilisons un étage de tri, comme sur la figure 2.21.

2.6.3 Compilation du zip

Rappelons que l'opérateur zip est polymorphe. Lorsque les opérandes sont des ensembles ordonnés, sa sémantique est facile à implémenter. On utilise un nœud `Split` aux consommations $1, \ldots, 1$ qui va entrelacer les ensembles passés en argument.

Lorsque que les opérandes sont des itérateurs, il faut prendre en compte la taille des éléments itérés pour l'entrelacement. Par exemple un itérateur A qui retourne des ensembles ordonnés de taille 10 et un itérateur B qui retourne des ensembles ordonnés de taille 5 seront entrelacés par l'expression $zip(A, B)$ en utilisant un nœud `Split` aux consommations $10, 5$.

Finalement, pour des opérandes mixtes, on prendra 1 comme consommation pour les entrées de type *orderedset* et pour les entrées de type *iterator* on prendre la taille des éléments itérés.

2.6.4 Compilation des nids de boucles

Plusieurs flots peuvent être combinés en emboîtant des boucles `for in`. Notre compilateur analyse les nids de boucles et réordonne et duplique les éléments en adéquation avec la profondeur de chaque `push`.

Considérons l'exemple suivant :

```
for a in A:
  for b in B:
    for c in C:
      for d in D:
        push c
      push b
```

```
shape[3] B = IB.input
shape[2] A = IA.input
for a in A:
  for b in B:
    push a
    push b
  push a
```

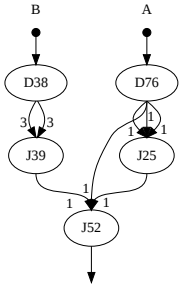


FIGURE 2.22 – Compilation des nids de boucles : à gauche, un exemple de nids de boucles, à droite, le sous-graphe SJD correspondant.

Soit $\overline{X}$ le nombre d'éléments dans l'itérateur X. Pour chaque *push* x, on définit les objets suivants :

- L'itérateur qui génère x est appelé l'itérateur de base et on le notera $base(x)$. Par exemple sur l'exemple ci-dessus l'itérateur de base de "push b" est B.
- $Outer(x)$ est l'ensemble des itérateurs qui englobent $base(x)$. Ici $Outer(b) = \{A\}$ et $Outer(c) = \{A, B\}$.
- $Inner(x)$ est l'ensemble d'itérateurs englobés par $base(x)$ *et* englobant "push x". Ici $Inner(b) = \{C\}$ et $Inner(c) = \{D\}$.

La taille de $Outer(x)$ est calculée de la façon suivante,

$$\overline{Outer(x)} = \prod_{O \in Outer(x)} \overline{O}$$

de la même manière,

$$\overline{Inner(x)} = \prod_{I \in Inner(x)} \overline{I}$$

Pour satisfaire la sémantique des boucles, pour chaque primitive "push x" il nous faut rejouer l'itérateur de base autant de fois que le nombre d'itérations des boucles extérieures $\overline{Outer(x)}$; et maintenir la valeur courante autant de fois que le nombre d'itérations des boucles intérieures $\overline{Inner(x)}$. On peut exprimer ces opérations en associant un nœud D et un nœud J. Par exemple, sur la figure 2.22, on a représenté un nid de boucles exemple et le sous-graphe SJD compilé.

2.6.5 Compilation d'un programme SLICES

Dans chacune des sections précédentes on a donné pour chaque primitive de SLICES un patron permettant de construire un sous-graphe SJD équivalent. Nous allons maintenant ébaucher l'architecture de notre compilateur ; et montrer comment il permet d'assembler ces différentes pièces pour compiler un programme SLICES vers un graphe SJD. La première phase de notre compilateur est un parseur classique, construit autour de `flex` et `bison` qui permet de construire un arbre syntaxique à partir d'un programme SLICES. La grammaire formelle de SLICES est donnée sur la figure 2.23 dans la représentation BNF. Un programme est constitué d'une liste de déclarations (`decls`) et d'une liste d'instructions (`stmts`).

En un premier temps, la liste de déclarations est parcourue par le compilateur qui renseigne au sein d'un dictionnaire contexte les différents types `shape` déclarés (et éventuellement la valeur

```
root ::= decls stmts
decls ::= decl NEWLINE
        | decls decl NEWLINE
decl ::= shape_decl
       | padding_decl
stmts ::= statement
        | stmts statement
statement ::= push_stm NEWLINE
            | for_loop DEDENT
collection_id ::= IDENTIFIER
padding_decl ::= 'padding' '(' collection_id ')' '=' INTEGER
shape_decl ::= 'shape' shape_dim collection_id
             '=' CHANNEL . 'input'
shape_dim ::= '[' dim_list ']'
dim_list ::= INTEGER
           | dim_list ',' INTEGER
for_loop ::= 'for' collection_id 'in' iterator
             ':' NEWLINE INDENT stmts
push_stm ::= CHANNEL '.' 'push' iterator
           | 'push' iterator
zip_stm ::= 'zip' '(' iterator ',' iterator ')'
iterator ::= collection_id
           | slices
           | zip_stm
slices ::= grid
         | blocks
grid ::= collection_id '[' triplet_list ']'
blocks ::= grid 'x' '(' couple_list ')'
triplet_list ::= triplet
               | triplet_list ',' triplet
couple_list ::= couple
              | couple_list ',' couple
triplet ::= maybe ':' maybe ':' maybe
couple ::= maybe ':' maybe
maybe ::=
        | INTEGER
```

FIGURE 2.23 – La grammaire formelle de SLICES dans la représentation BNF.

de padding à utiliser). Dans les phases ultérieures du compilateur, le contexte sera consulté pour connaître les paramètres des différents `shape` et pour détecter les erreurs de typage.

En un deuxième temps, la liste des instructions est parcourue. Il y a deux types d'instructions possibles, des `push` nus qui produisent des données sur un unique canal, et des nids de boucles `for` qui peuvent contenir plusieurs `push` et donc produire des données sur plusieurs canaux sortants.

Dans le premier cas, les primitives `push` contiennent un *iterator* ou un *orderedset* qui sont obtenus par les opérations `grid`, `blocks`, `shape` ou `zip`. Nous avons montré dans les sections précédentes comment compiler un sous-graphe SJD implémentant ces structures. Dans le deuxième cas, on a affaire à un nid de boucles `for` imbriqués. On a montré en § 2.6.4 comment compiler un sous-graphe SJD pour chaque `push` de la boucle. Nous pouvons pour chaque `push` , qu'il soit nu ou à l'intérieur d'un nid de boucles, produire un graphe SJD approprié. Il ne reste plus qu'à connecter ces différents graphes aux nœuds `Input` et `Output`.

Un même `Input`, peut-être utilisé par plusieurs des sous-graphes produits. À l'aide d'un nœud `Duplicate`, nous dupliquons donc chaque `Input` en autant de copies que nécessaire, ce qui nous permet d'alimenter les entrées des sous-graphes produits. Un même `Output`, peut recevoir des données de plusieurs sous-graphes. Nous connectons donc les sorties des sous-graphes (dans l'ordre des `push`) à un unique nœud `Join` qui alimente chaque `Output`. Les consommations du nœud `Join` dépendent de la taille des itérateurs ou ensembles ordonnés produits par chaque sous-graphe.

2.6.6 Exemples : graphes SJD produits par notre compilateur SLICES

Nous allons montrer le fonctionnement du compilateur SLICES vers SJD sur les exemples de programmes slices introduits dans la § 2.5.8.

Considérons tout d'abord la multiplication matricielle en figure 2.24. Le graphe SJD produit par notre compilateur est représenté à droite. Ces figures sont directement produites par notre compilateur et utilisent des notations légérement différentes : les productions et consommations des nœuds sont figurées sur l'étiquette ; les faisceau d'arcs consécutifs de même production et consommation sont remplacés par un arc en rouge gras et un multiplicateur qui indique la taille originale du faisceau. Pour la multiplication, on observe que le graphe produit est exactement le graphe SJD programmé à la main en figure 2.9. L'étage de transposition (en haut de la branche droite) correspond à l'étage de tri généré lors de l'extraction des blocs des colonnes. Les deux couples $D - J$ sont produits lors de la compilation des imbrications de boucles `for`.

L'étage de séparation pair/impair utilisé pour la FFT est représenté en figure 2.25. La branche droite et gauche sont très semblables. Observons la branche gauche qui correspond aux données paires. Le premier `Split` correspond à l'extraction de la région on élimine la donnée I_{15} qui n'est pas paire. Puis on remarque le sous-graphe d'extraction de blocs sans recouvrement : ici on extrait des blocs de taille 1 séparés par des trous de taille 1.

Cette implémentation contrairement à celle faite à la main en SJD utilise une stratégie de décimation : on duplique les données, puis on supprime les données paires à gauche et les impaires à droite. Dans certaines architectures faire une copie de toutes les données et supprimer les données indésirables peut-être efficace. Dans d'autres la duplication de données peut amener un surcoût inutile, on préferera la version SJD qui n'utilise que des `Split` et `Join`. On montrera néanmoins qu'il est possible de récupérer la version SJD à partir de la version SLICES en utilisant les transformations du chapitre 3.

Passons maintenant à l'exemple du filtre de Sobel que nous avons repris en figure 2.26. Nous considérons ici la version utilisant la séparabilité du noyau composée d'une convolution horizontale et d'une convolution verticale. L'extraction des triplets horizontaux, comme verticaux, produit le premier graphe de la figure. On identifie les nœuds d'extraction de région en haut du graphe : on insère des données sur les bords droits et gauches pour alimenter les blocs qui sortent de la figure. Puis, vient l'étage d'extraction de blocs avec recouvrement total.

L'extraction des colonnes, nécessaire avant de pouvoir extraire les triplets verticalement, est

```
shape[5,10] A = IA.input
shape[10,20] B = IB.input

for l in A[0:1:,::] x (:,0:0):
  for c in B[::,0:1:] x (0:0,:):
    push zip(l,c)
```

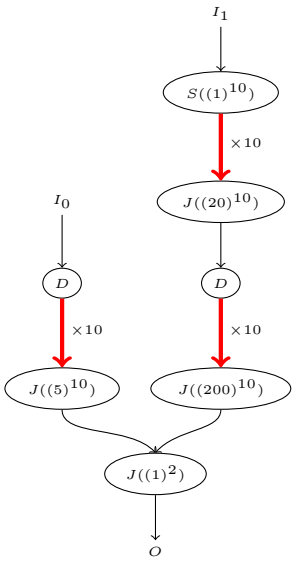


FIGURE 2.24 – Multiplication matricielle en SLICES et le graphe SJD compilé.

compilée vers un simple étage de transposition correspondant à l'étage de tri.

Le filtre de Gauss est représenté en figure 2.27. Le premier étage du graphe constitué des sources et des premiers nœuds `Join`, correspond à l'extraction de région. Ici, on insère un bord épais d'un pixel autour de l'image originale. Puis viennent les étages d'extraction de blocs horizontal et vertical. Enfin en bout de chaîne on identifie l'étage de tri.

Finalement le filtre de Hough générè le graphe en figure 2.28. On observe que le graphe correspond exactement à la partie supérieure du graphe que l'on a réalisé à la main en figure 2.12.

2.6.7 Propriétés des graphes produits par SLICES

Duplications de données

Si on reprend l'implémentation du filtre gaussien en SJD (cf. figure 2.11), on peut observer que l'étage vertical de composition des blocs ignore un certain nombre de blocs de données (éliminées dans les nœuds `Sink`) qui pourtant ont été dupliqués par l'étage horizontal. Pourquoi dupliquer des données qui vont être par la suite détruites ? Le programme est inefficace.

On a construit SLICES de manière à réduire le nombre de copies inutiles nécessaires lors de l'extraction de blocs pour ne pas envoyer des données inutiles en aval. Dans l'extraction des *blocks*[1] les nœuds `Duplicate` ont été placés le plus en aval possible, de manière à factoriser les transformations communes. Par exemple, au lieu de dupliquer les éléments communs à plusieurs blocs, puis d'extraire la région dans chaque copie du bloc ; on commence par extraire la région avant de faire des duplications.

Lors de l'extraction de *blocks*[*d*], on réordonne les étages *blocks*[1] de manière à réduire les données en aval. Dans cet ordre tous les étages sans recouvrement (§ 2.6.1) sont poussés en haut

```
shape[N] I = I.input
for pair in I[::2]:
  push pair
for impair in I[1::2]:
  push impair
```

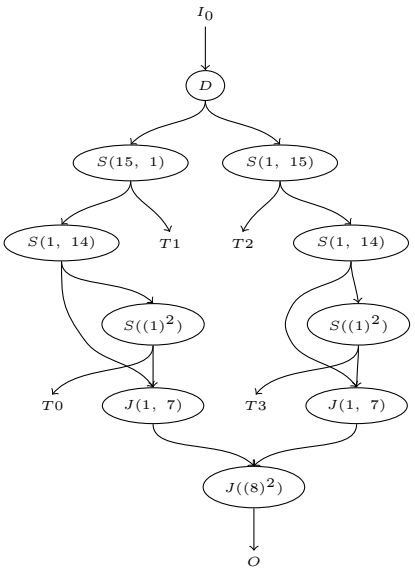


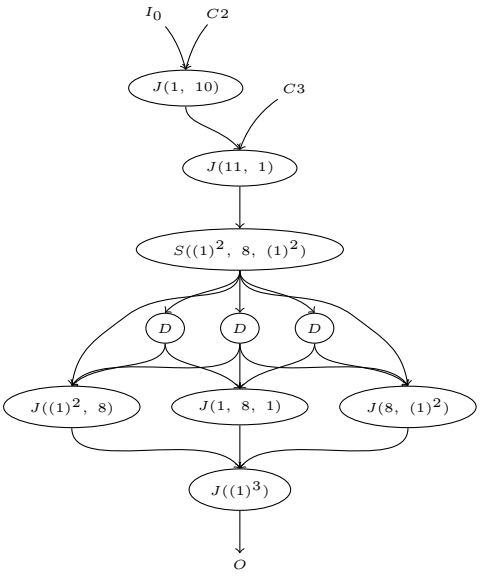
FIGURE 2.25 – Séparation des données paires et impaires, ici au premier niveau de la FFT sur un vecteur de taille 16

```
shape[10,10] I = image.input
for triplet in I[::,::] x (-1:1,0:0):
  push triplet
```

Extraction des triplets horizontaux

```
shape[10] I = columns.input
for triplet in I[::] x (-1:1):
  push triplet
```

Extraction des triplets verticaux




```
shape[10,10] I = image.input
for column in I[::,0:1:] x (0:0,:):
  push columns.column
```

Extraction des colonnes

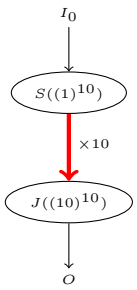


FIGURE 2.26 – Filtre de Sobel (version utilisant la séparabilité du noyau) en SLICES et le graphe SJD compilé.

```
shape[10,10] I = image.input
for block in I[::,::] x (-1:1,-1:1):
  push block
```

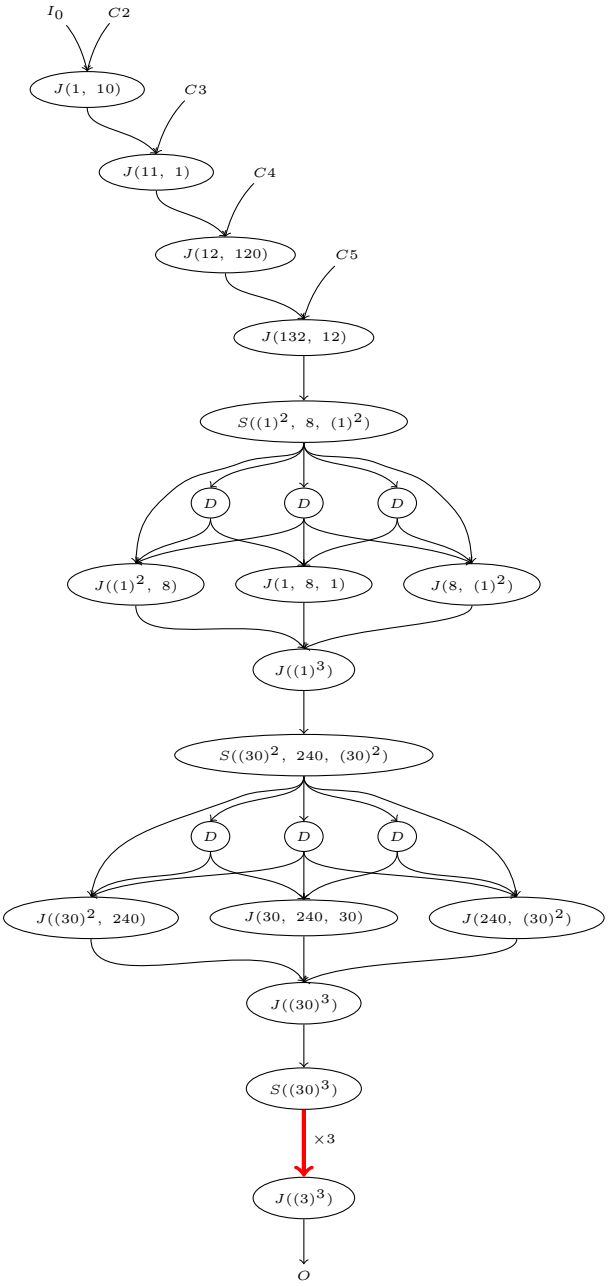


FIGURE 2.27 – Filtre de Gauss en SLICES et le graphe SJD compilé.

```
shape[2,100] points = points.input
shape[alpha] angles = angles.input
for a in angles:
  for p in points[0:1:,::]x(:,0:0):
    push a
    push p
```

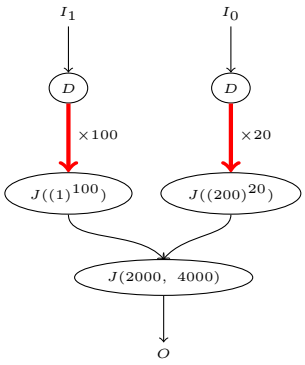


FIGURE 2.28 – Filtre de Hough en SLICES et le graphe SJD compilé.

du pipeline. Ceci est intéressant car les étages sans recouvrement éliminent des éléments et réduisent donc le nombre de données produites en aval.

Complexité des graphes générés

Nous allons donner la complexité pire cas des graphes SJD générés par notre compilateur en nombre de nœuds et en nombre d'arcs. Pour cela nous introduisons tout d'abord quelques grandeurs caractéristiques d'un programme SLICES :

- $ms = max_i(s_i)$, c'est le maximum des dimensions de l'image d'entrée ; en pratique cette valeur peut être très grande (par exemple dans un programme de traitement d'images haute-définition).
- d, c'est le nombre de dimensions dans l'image ;
- p, le nombre de `push` utilisés ;
- $mw = max_i(w_i)$, c'est le maximum des dimensions du motif extrait ;
- $mo = max_x(\overline{Outer(x)})$, c'est le maximum de répétitions rencontré dans les nids de boucles.

Dans les cas sans recouvrement et avec recouvrement partiel, le motif pour extraire un `blocks[1]` unidimensionnel est constitué d'un nombre constant de nœuds $K(= \mathcal{O}(1))$. Dans le cas avec recouvrement total, le nombre de nœuds est en $\mathcal{O}(mw)$ et le nombre d'arcs est en $\mathcal{O}(mw^2)$.

Dans le cas multidimensionnel, il faut considérer que l'on traite d `blocks` unidimensionnel. Il faut aussi prendre en compte l'étage de tri qui nécessite $\mathcal{O}(1)$ nœuds et $\mathcal{O}(mw)$ arcs. Ce qui nous donne une complexité cumulé de $\mathcal{O}(d.mw + 1) = \mathcal{O}(d.mw)$ nœuds et de $\mathcal{O}(d.mw^2 + mw) = \mathcal{O}(d.mw^2)$ arcs.

Il faut maintenant considérer le cas des nids de boucles. Comme précisé en 2.6.4, on peut être amené à dupliquer une valeur $\overline{(Outer(x))}$ fois ; ce qui implique l'utilisation d'une jonction $D - J$

avec autant d'arcs. Un nids de boucles entraîne donc une complexité au pire cas de $2(= \mathcal{O}(1))$ nœuds et de $\mathcal{O}(mo)$ arcs.

Finalement, en supposant que tous les itérateurs travaillent sur des motifs différents, il faut multiplier ces complexités par le nombre de `push` utilisés, ce qui nous donne :

- une complexité en nœuds au pire cas, $C_{noeuds} = \mathcal{O}(p.d.mw)$;
- une complexité en arcs au pire cas, $C_{arcs} = \mathcal{O}(p.d.(mw^2 + mo))$.

Nous remarquons que les complexité en nœuds et en arcs sont indépendantes de la taille des entrées (ms) [3]. Elles dépendent essentiellement de la taille du motif extrait et du nombre de motifs extraits.

C'est un résultat intéressant : cela veut dire que la complexité du programme ne dépend pas de la taille des données traitées (sauf si on itère explicitement dessus). C'est d'autant plus important que la complexité des méthodes d'optimisation de graphes SJD proposées dans le chapitre suivant, sont liées à la complexité en nœuds en du graphe.

2.7 Comparaison des langages

Maintenant que le lecteur s'est familiarisé avec SJD et SLICES nous allons peser les pour et les contres des deux langages.

2.7.1 Comparaison de SJD et SLICES

Expressivité SJD permet d'exprimer tous les arrangements avec répétition des entrées (§ 2.4.2) alors que SLICES est limité aux motifs réguliers (§ 2.5.9). SLICES n'est donc pas un remplacement pour SJD. Mais il permet néanmoins d'exprimer de nombreuses réorganisations de données dans le domaine du traitement du signal. SLICES et SJD peuvent être utilisés ensemble pour exprimer des réarrangements de données complexes, par exemple pour faire un parcours irrégulier de blocs dans une image, on peut charger SLICES de l'extraction des blocs et réordonner les blocs à l'aide d'un petit graphe SJD.

Facilité d'utilisation L'écriture de réorganisations dans SJD est difficile et verbeuse : les nœuds de routage ont une sémantique très bas niveau qui nous oblige à projeter notre découpage de données en une seule dimension avant de pouvoir l'implémenter. SLICES est lui beaucoup plus naturel, puisqu'il s'appuie sur une expression haut-niveau d'un découpage de données en plusieurs dimensions et l'utilisation des familières boucles `for`. Pour s'en rendre compte il suffit de comparer le code 2.1 implémentant la multiplication matricielle dans SJD et la version SLICES. Dans le programme SJD, des index complexes sont employés, et les connexions entre nœuds sont difficiles à visualiser dans la représentation ΣC (profusion de `connect`). Le programme SLICES, offre une représentation compacte où l'itération sur les colonnes et les lignes apparait immédiatement.

Complexité des graphes générés Enfin, le compilateur SLICES réduit le nombre de répétitions dans les graphes générés et garantit que la complexité du graphe n'est pas fonction de la taille de l'image mais fonction du nombre d'itérations imbriquées et du motif extrait. Par exemple l'implémentation du filtre gaussien en SJD proposée en figure 2.11 a une complexité proportionnelle à la taille de l'image puisque l'étage central de réorganisation à autant d'arcs que de colonnes dans l'image. Si on veut appliquer le programme à une image haute-définition, cela peut-être problématique. La version du filtre Gaussien produite par SJD ne dépend que de la taille du motif

3. Bien entendu, si un des nids de boucles itère sur la taille des entrées alors $mo = ms$, c'est le cas pour notre implémentation du filtre de Hough.

extrait. Bien entendu, un programmeur expérimenté peut obtenir les mêmes garanties en écrivant le graphe SJD à la main.

2.7.2 Comparaison entre SJD et StreamIt

SJD et StreamIt sont très proches. D'ailleurs les versions SJD de la multiplication matricielle ou de la FFT ont été directement traduites depuis StreamIt. Néanmoins SJD présente quelques différences :

- les nœuds peuvent être connectés arbitrairement
- on dispose de nœuds `Sink` et `Constant`

Ces différences sont importantes car elles permettent à SJD d'exprimer toutes les réorganisations de données statiques à support fini, comme nous l'avons montré dans le théorème 2.4.5. L'expressivité des nœuds de routage dans StreamIt est plus faible à cause de l'obligation de connecter les nœuds de manière hiérarchique. Cela veut dire par exemple que le premier élément produit sur un graphe sera toujours en première position. Certaines permutations sont impossibles dans StreamIt et doivent être exprimées à l'intérieur d'un filtre de calcul.

2.7.3 SJD et le modèle polyédrique

Nous avons montré que SJD permet de représenter tous les arrangements avec répétitions mais qu'en est-il du modèle polyédrique[Fea92a] ?

Un seul polyèdre Si l'on considère un seul polyèdre, nous pouvons voir avec un exemple très simple que ce dernier ne peut pas exprimer tous les réarrangements avec répétition. Prenons le réarrangement $[1,2,3,4] \rightarrow [3,2,2,7]$ qui produit les éléments d'un stream dans l'ordre $3,2,2,7$. Pour exprimer ce réarrangement dans le modèle de dépendances affines, il faut trouver une fonction f affine telle que le pseudo-code

```
...
   for l = 0,lmax
     for k = 0,kmax
       for j = 0,jmax
         for i = 0,imax
             push I[f(i,j,k,l, ...)]
```

produise les éléments I_3, I_2, I_2, I_7 avec $f = a + b.i + c.j + d.k + e.l + \ldots$

Considérons les valeurs du vecteur d'itération $(i, j, k, l, \ldots)$,

- $f(0,0,0,0,\ldots) = 3$, donc $a = 3$.
- $f(1,0,0,0,\ldots) = 2$, donc $3 + b = 2$ et $b = -1$.
- $f(0,1,0,0,\ldots) = 2$, donc $3 + c = 2$ et $c = -1$.
- $f(1,1,0,0,\ldots) = 7$, donc $3 - 1 - 1 = 1 \neq 7$, absurde.

Union de polyèdres Néanmoins le modèle polyédrique permet de travailler avec des unions finies de polyèdres. Dans ce cas, on peut exprimer tous les réarrangements avec répétition et on retrouve la même expressivité que SJD.

2.7.4 Comparaison entre SLICES et ArrayOL

Array-OL que l'on a déjà présenté dans l'introduction, propose un modèle de programmation très proche de celui de SLICES avec quelques différences. Array-OL permet de décrire des découpages de données sur des tableaux toriques : au lieu d'injecter des éléments sur les bords comme dans SLICES, on prend les éléments de l'autre côté du tableau. Array-OL ne permet pas de décrire des réorganisations de données entrelacées sur plusieurs flux, alors que SLICES en est capable avec les nids de boucles `for`. Par contre, Array-OL permet l'extraction de blocs non parallèles aux axes de l'espace des données ; fonctionalité qui est absente dans SLICES. Les auteurs de [ABD05] montrent qu'il est possible de compiler un programme ArrayOL vers des réseaux de processus de Kahn (KPN). Dumont projette ArrayOL sur un modèle d'execution SDF[Dum05], mais la projection est à gros grain (au niveau global d'ArrayOL) et cache entièrement le parallélisme possible entre les différents motifs extraits. En ce sens la projection de SLICES vers les CSDF est plus fine puisqu'elle expose le parallélisme entre motifs.

2.8 Travaux connexes

ZPL[DCS04] est un langage haut-niveau pour l'expression de programmes parallèles. C'est un des précurseurs des langages de la famille PGAS (*Partitionned Global Address Space*). L'idée dans ZPL est de distribuer l'espace des données sur un ensemble de processeurs virtuels. Tous les opérateurs ZPL travaillent alors de manière collective sur ce découpage des données ; ce qui permet d'exprimer aisément des opérations en parallèle ou des réorganisations de données. Pour décrire les découpages de données ZPL introduit la notion de région. Une région est un tableau à n dimensions qui peut-être déplacé sur l'espace des données en utilisant des vecteurs de déplacement (*stride vectors*). Nous nous sommes fortement inspirés de ZPL pour construire le langage SLICES. Le pouvoir d'expression de ZPL est supérieur à celui de SLICES. En effet les vecteurs de stride dans ZPL sont moins contraints, ils peuvent être irréguliers et même dynamiques. Ceci est à la fois un avantage et un désavantage. En effet, ZPL ne peut pas être statiquement compilé vers SJD puisque les valeurs de certains vecteurs de déplacement ne sont pas connues à la compilation.

L'idée d'utiliser un type *shape* pour restructurer les données en plusieurs dimensions à été empruntée au langage Single Assignment C (SAC)[Sch03]. SAC propose une variante fonctionnelle de C avec des opérations de tableaux multidimensionnels. De nombreuses optimisations ont été proposées dans SAC pour combiner les manipulations de tableaux successives en une seule opération.

La notation des blocs et des grilles a été empruntée aux langages Matlab[Mat96] et Python[vR95].

2.9 Conclusion

Dans ce chapitre nous proposons une méthode hybride pour l'expression du routage de données statiques dans le parallélisme de flots.

Le langage SJD, permet d'exprimer tous les réarrangements statiques à support fini des données. C'est donc une bonne représentation intermédiare pour effectuer les transformations optimisantes que l'on présentera dans le chapitre suivant. De plus SJD est contenu dans le modèle CSDF ce qui garantit une exécution déterministe en mémoire bornée des programmes.

Le langage SLICES, offre un modèle de programmation accessible qui facilite la description de nombreuses application de traîtement du signal. SLICES possède un système de types qui permet de détecter les programmes incorrects à la compilation.

Nous montrons comment compiler SLICES vers le modèle SJD. Les graphes produits par SLICES peuvent être combinés avec des programmes SJD généralistes (codés en ΣC ou en StreamIt par exemple) qui permettent de décrire les filtres nécessaires au calculs sur les données.

Chapitre 3

Transformations de graphes SJD

Sommaire

Dans le chapitre précédent nous avons présenté deux langages, SJD et SLICES, pour exprimer des programmes de flots de données, nous avons particulièrement insisté sur leur capacité à exprimer des routages de données. SJD et SLICES sont complémentaires, nous avons montré qu'ils peuvent être intégrés au sein d'un même programme en compilant dans le frontend de notre compilateur les parties écrites en SLICES vers du SJD pur. Dans ce chapitre nous nous situons en aval du frontend : nous ne travaillons plus qu'avec du code SJD. L'objet de ce chapitre est la transformation automatique de code SJD pour l'optimisation.

On peut vouloir optimiser un programme selon différents critères. Certains critères correspondent à des contraintes de l'architecture, par exemple, un programme ne peut pas utiliser plus de mémoire que celle dont dispose l'architecture. D'autres critères visent à améliorer la performance du programme, par exemple, en augmentant le parallélisme, ou en réduisant le coût des communications d'un programme.

Pour évaluer ces critères, il faut savoir précisément comment les programmes sont exécutés sur la cible. Si on cherche, par exemple, à diminuer la mémoire consommée par un graphe SJD, il nous faut savoir comment celle-ci est allouée à l'exécution : les éléments échangés entre deux nœuds sont-ils copiés sur un tampon ou copiés à travers des registres ? Si un tampon est utilisé, dans quelle mémoire est-il alloué ?

Dans le chapitre 4, nous montrerons dans le cadre de notre implémentation d'un compilateur SJD, comment optimiser la mémoire et les communication d'un programme en explorant l'espace engendré par transformations. Néanmoins dans ce chapitre nous souhaitons nous détacher de l'architecture et du compilateur choisi, c'est pourquoi nous considérerons que l'on cherche à minimiser une fonction économique $\phi : \mathit{Graphes} \mapsto \mathbb{R}$. Dans le chapitre suivant nous montrerons comment choisir ϕ de manière à réduire l'empreinte mémoire d'un programme pour qu'il tienne sur la cible ou de manière à optimiser les coûts de communcations sur la cible.

Puisque nous ne faisons à priori pas d'hypothèses sur le critère à optimiser, nous allons présenter une méthode très générale pour optimiser des graphes. D'abord on va proposer un ensemble de transformations sur les graphes SJD qui n'altèrent pas la sémantique du programme. Elles nous permettent donc de générer des implémentations alternatives de notre programme. Puis nous montrerons comment explorer l'espace des alternatives pour trouver le graphe qui minimise la fonction économique ϕ.

L'espace d'exploration, très grand, sera en pratique exploré au moyen d'heuristiques. Néanmoins nous montrerons qu'au prix de certaines restrictions sur les transformations appliquées il est possible de se ramener à un espace d'exploration fini. Le fait d'avoir un espace d'exploration fini nous permet de trouver l'optimum pour ϕ par une recherche exhaustive et de mesurer la qualité des heuristiques proposées.

Pour résumer les contributions de ce chapitre sont :

- un modèle formel de correction des transformations ;
- des transformations originales sur les graphes de flots de données ;
- un espace d'exploration fini qui nous permet d'évaluer les heuristiques proposées.

Une partie des contributions de ce chapitre ont été publiées dans [dOCLB10b] et sont résumées dans un rapport technique [dOCLB09].

3.1 Transformations de graphes

Qu'entendons nous exactement par transformation de graphe ? Toutes les transformations de graphe que l'on considère vont extraire un bout du graphe et le remplacer par un bout transformé, elles sont donc localisées sur une partie du graphe. Plus formellement, nous adoptons la formulation de [AEH+99]. Une transformation T qui s'applique à un graphe SJD G et qui génère un graphe G' est notée $G \xrightarrow{T} G'$. Une telle transformation va chercher un sous-graphe particulier, noté L, dans le graphe original G ; si L est trouvé il sera remplacé par le sous-graphe R. La transformation $G \xrightarrow{T} G'$ est ainsi définie par :

- un sous-graphe gauche $L \subseteq G$ dont les *arcs frontière* sont les arcs entre les nœuds de L et les nœuds de $G \backslash L$;
- un sous-graphe droit R possédant autant *d'arcs frontière* entrants (resp. sortants) que L possède *d'arcs frontière* entrants (resp. sortants) ;
- une bijection g entre les arcs frontière de L et les arcs frontière de G.

Le graphe G' est alors obtenu en supprimant le sous-graphe L dans G ce qui produit le graphe $G \backslash L$. Les *arcs frontière* entre $G \backslash L$ et L sont donc déconnectés. On utilise alors la bijection g, pour « coller » le graphe R à la place de L, ce qui produit le graphe G'. Considérons par exemple la transformation en figure 3.1. Le sous-graphe gauche L est représenté à gauche de la transformation, ici il s'agit d'un couple $S - J$ avec des consommations et des productions identiques égales à $(2, 2)$. Le sous-graphe L possède un arc frontière entrant i_1 et un arc frontière sortant o_1. Le sous-graphe

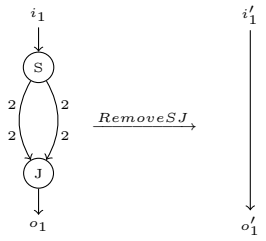


FIGURE 3.1 – Exemple de transformation : un couple de nœuds $S - J$ de même consommations et productions sont supprimés.

R, à droite, ne possède pas de nœuds, et possède un arc frontière qui fait à la fois office d'arc entrant i'_1 et d'arc sortant o'_1. La bijection g est implicitement donnée par la correspondance $i_1 \to i'_1$ et $o_1 \to o'_1$.

Bien sûr, une même transformation peut-être appliquée à plusieurs endroits d'un même graphe G, comme illustré en figure 3.2. On peut également appliquer plusieurs transformations à la suite : par exemple, le graphe en figure 3.2, admet successivement les transformations $RemoveSJ_{S1,J1}$ et $RemoveSJ_{S2,J2}$ qui produisent le graphe F→G→H. L'application de plusieurs transformations à la suite s'appelle une dérivation.

Définition 3.1.1. *Une dérivation d'un graphe G_0 est une chaîne de transformations qui peuvent lui être successivement appliquées : $G_0 \xrightarrow{T_0} G_1 \ldots \xrightarrow{T_n} \ldots$.*

Plusieurs dérivations, peuvent parfois confluer sur le même graphe. Ainsi dans l'exemple précédent, si on applique les transformations dans l'ordre inverse $RemoveSJ_{S2,J2}$ et $RemoveSJ_{S1,J1}$, on arrive au même graphe final. On dira que $RemoveSJ_{S2,J2}$ et $RemoveSJ_{S1,J1}$ commutent. Ce n'est pas vrai pour tous les transformations.

3.2 Correction d'une transformation

Nous souhaitons définir des transformations permettant d'explorer des implémentations alternatives d'un programme original SJD. Bien sur nous ne nous intéressons qu'aux transformations qui conservent la sémantique du programme original ; les transformations qui changent le comportement du programme ne sont pas correctes. Par correction nous entendons que le programme, après transformations, réalise le même calcul que le programme original et que le programme soit correct au sens de la définition 2.4.16 (c'est-à-dire qu'il soit consistant et vivace.).

Nous allons définir formellement ce qu'est une transformation correcte. Pour cela nous nous appuyons sur la sémantique des CSDF qui a été définie en § 2.4. Nous supposons sans perte de généralité que les graphes que nous considérons sont connexes (dans le cas contraire, il suffit de vérifier la conservation de la sémantique sur chaque composante connexe du graphe).

Définition 3.2.1. *Soit un graphe CSDF G, une transformation $G \xrightarrow{T} G'$ est* correcte ssi

$$I(G) = I(G') \quad \Rightarrow \quad O(G) \leq O(G') \tag{3.1}$$

$$G \text{ est consistant} \quad \Rightarrow \quad G' \text{ est consistant} \tag{3.2}$$

$$G \text{ est vivace} \quad \Rightarrow \quad G' \text{ est vivace} \tag{3.3}$$

En d'autres mots, T préserve la sémantique de G si T préserve la consistance, la vivacité et si pour les mêmes entrées, le graphe généré par T, produit *au moins* les mêmes sorties que G. On verra, par la suite, dans la preuve du lemme 3.2.1 que (3.1) ⇒ (3.3).

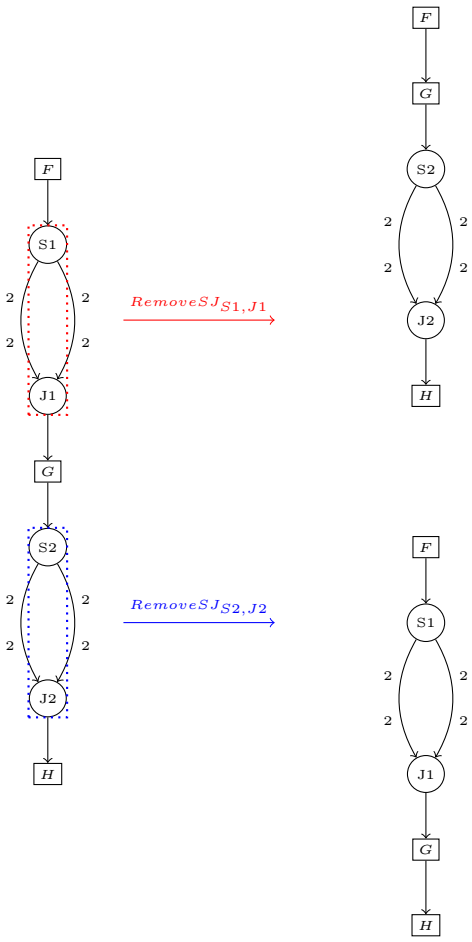


FIGURE 3.2 – Une même transformation peut-être appliquée à différents endroits d'un graphe. Par exemple ici *RemoveJS* peut-être appliqué au couple $S1 - J1$ en rouge ou au coupe $S2 - J2$ en bleu.

La préservation de la consistance et de la vivacité sont des critères nécessaires si nous voulons garantir, après transformation, que les graphes produits sont toujours ordonnançables en mémoire bornée et libres d'interblocages. La préservation des traces nécessite une explication supplémentaire. On peut se demander pourquoi pourquoi la propriété (3.1) n'est pas

$$O(G) = O(G') \tag{3.4}$$

qui implique que les traces soient exactement les même avant et après transformation. La raison est que cette contrainte forte écarterait de nombreuses transformations qui sont pourtant très utiles et qui n'altèrent pas les traces produites par le programme mais le « rythme » de production de ces traces.

Pour expliquer ce phénomène nous allons reprendre l'exemple de la section précédente en figure 3.3. Le couple de nœuds $S - J$ ne change pas l'ordre des éléments puisque les consommations et productions sont les mêmes des deux côtés. On pourrait donc croire qu'ils peuvent être éliminés sans altérer les traces. Pourtant avec la propriété forte (3.4), cette transformation est incorrecte. Le couple $S - J$ impose que les éléments soient toujours consommés par paires (car un nœud CSDF n'est exécuté que si ses consommations sont satisfaites). En éliminant le couple $S - J$ on viole la propriété (3.4) sur les entrées de taille impaire, puisque la sortie de G est tronquée à une taille paire alors que celle de G' est de taille impaire.

C'est cette observation qui nous a motivé à proposer la propriété (3.1) faible. La propriété faible relâche les contraintes sur le nombre d'éléments produits mais impose aux graphes transformés de produire *au moins* autant d'éléments que le graphe original (heureusement, car sinon on aurait pu remplacer tout graphe par le graphe vide qui ne fait rien).

Éléments résiduels Avec la propriété faible, des éléments résiduels peuvent donc être produits par le graphe transformé (p. ex. l'élément 5 dans notre exemple). En pratique cela ne pose aucun problème, en effet le nombre d'éléments résiduels est connu statiquement (il suffit de faire la différence du nombre d'éléments produits sur un ordonnancement stationnaire pour G et pour G') ; on peut donc placer sur chaque sortie un nœud `Split` à une seule branche et une consommation de la taille attendue qui aura pour effet de bloquer les éléments résiduels et de restituer la propriété forte (3.4). Autoriser les éléments résiduels nous permet de relâcher les délais lors de la transformation du graphe quitte à les recréer sur chaque sortie.

Nous remarquerons également que la propriété faible (3.1) correspond à l'équivalence dénotationelle de deux réseaux de Khan. Si l'on considère des programmes qui ne finissent pas, c'est-à-dire avec des entrées infinies, les deux définitions sont équivalentes.

Proposition 3.2.1 (équivalence dans le cas de traces infinies)**.** *Pour des traces en entrée infinies, les propriété faible* (3.1) *et forte* (3.4) *sont équivalentes.*

Condition suffisante locale de correction

La définition 3.2.1 précédente de correction n'est pas très pratique. En effet pour l'établir il faut raisonner sur G et G', c'est-à-dire qu'il faut prendre en compte le contexte $(G \backslash L)$ dans lequel la transformation s'applique. Nous allons montrer qu'il est possible d'établir la correction d'une transformation en n'examinant que les sous-graphes L et R. Nous pourrons ainsi construire un ensemble de transformations qui seront toujours correctes indépendamment de l'endroit du programme où elles seront appliquées.

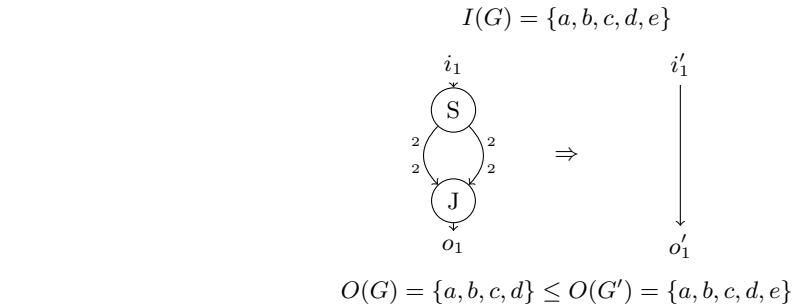


FIGURE 3.3 – Exemple de transformation qui n'est correcte que sous la condition faible (3.1).

Lemme 3.2.1 (Correction locale)**.** *Si une transformation* $G \xrightarrow{T} G'$ *vérifie*

$$\begin{aligned} &\exists b \in \mathbb{N}, \\ \forall I(L) = I(R) \quad &\Rightarrow \quad O(L) \leq O(R) && (3.5) \\ \forall I(L) = I(R) \quad &\Rightarrow \quad \max(\overline{O(R)} - \overline{O(L)}) \leq b && (3.6) \\ L \textit{ est consistant} \quad &\Rightarrow \quad R \textit{ est consistant} && (3.7) \end{aligned}$$

alors la transformation T *est* correcte.

Démonstration.

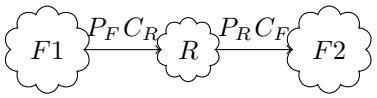
1. (3.5) ⇒ (3.1)

   D'après le théorème 2.4.3 tout sous-graphe d'un CSDF est continu et donc monotone (propriété 2.4.8). De la monotonicité de L,R,$G\backslash L$ and $G'\backslash R$, découle l'implication (3.5) ⇒ (3.1).

2. (3.5)(3.6)(3.7)⇒ (3.2)

   Si L n'est pas consistant, alors G non plus et (3.2) est vrai. Supposons donc que L, R et G sont consistants, et montrons que G' est consistant.

   Cas sans cycle : S'il n'y a pas de cycle entre les sorties de R et les entrées de R (à travers $F \equiv G'\backslash R$). Alors on peut séparer F de part et d'autre de R en $F1$ et $F2$ et le graphe G' s'écrit

   On note c_R et p_R les consommations et productions de R durant une exécution d'un ordonnancement stationnaire du sous-graphe R (qui existe puisque R est consistant). Ceci est vrai également pour F et L. D'après le théorème 2.4.4, s'il existe une solution non nulle au système d'inconnues q_R, q_{F1} et q_{F2} suivant alors G' est consistant,

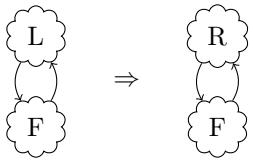
   $$\begin{aligned} P_F.q_{F1} &= C_R.q_R \\ P_R.q_R &= C_F.q_{F2} \end{aligned}$$

   Pour montrer la consistance de G' il suffit de vérifier que

   $$\begin{aligned} q_R &= C_F.P_F \\ q_{F1} &= C_R.C_F \\ q_{F2} &= P_R.P_F \end{aligned}$$

est une solution du système.

Cas avec cycle : Sans perte de généralité, on suppose que L et R sont connectés à travers $F \equiv G\backslash L \equiv G'\backslash R$ avec un seul arc entrant et un seul arc sortant. Dans le cas avec plusieurs arcs entrants et sortants il suffit de faire le raisonnement ci-dessous pour chacun d'eux.



On pose la fonction $l(x)$ qui donne la taille de la trace de la sortie du sous-graphe L pour une entrée de taille x. On définit de même les fonctions $r(x)$ et $f(x)$ pour les sous-graphes R et F.

Par hypothèse nous savons que,

$$\forall k, l(k.c_L) = k.p_L \tag{3.8}$$
$$\forall k, r(k.c_R) = k.p_R \tag{3.9}$$
$$\forall k, f(k.p_L) = k.c_L \tag{3.10}$$

Si nous exécutions p_L fois l'ordonnancement stationnaire du sous-graphe R, nous obtiendrions une sortie de taille $\overline{O(R)} = r(p_L.c_R) = p_L.p_R$. La sortie de R est l'entrée de F dans G', donc $\overline{O(F)} = f(\overline{O(R)}) = f(p_L.p_R) = p_R.c_L$. Si ce scénario était possible, F comme R seraient alimentés par des entrées dont la taille est multiple de leur consommation dans un ordonnancement stationnaire, donc par définition F et R obéiraient un ordonnancement stationnaire.

Pour montrer que ce scénario est réalisable (et que G' admet un ordonnancement stationnaire), il nous faut prouver que :

$$\overline{O(F)} = \overline{I(R)} \Leftrightarrow p_L.c_R = p_R.c_L \tag{3.11}$$

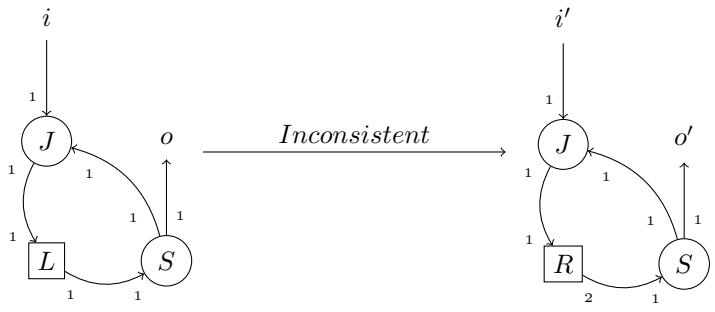
Pour cela on dérive :

$$(3.8) \Rightarrow l(c_L.c_R) = p_L.c_R \tag{3.12}$$
$$(3.9) \Rightarrow r(c_L.c_R) = c_L.p_R \tag{3.13}$$
$$(3.5) \Rightarrow \exists n_1, r(c_L.c_R) = l(c_L.c_R) + n_1 = p_L.c_R + n_1 \tag{3.14}$$
$$(3.5) \Rightarrow \forall k, \exists n_k, r(k.c_L.c_R) = k.p_L.c_R + n_k \tag{3.15}$$
$$(3.6) \Rightarrow \forall k, n_k \leq b \tag{3.16}$$

Puisque $k.c_L.c_R$ et $c_L.c_R$ sont des multiples des consommations stationnaires de R on peut écrire :

$$\begin{aligned} \forall k, k.r(c_L.c_R) &= r(k.c_L.c_R) \\ \forall k, k.(p_L.c_R + n_1) &= k.p_L.c_R + n_k \text{ en substituant par (3.14)(3.15)} \\ \forall k, k.n_1 &= n_k, \text{ or } k.n_1 \leq b \text{ par (3.16)} \\ \text{d'où } n_1 &= 0 \text{ et } \forall k, n_k = 0 \end{aligned} \tag{3.17}$$

Donc, (3.17)(3.14)(3.13) $\Rightarrow$ (3.11), *cqfd.*



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

FIGURE 3.4 – Exemple de transformation qui préserve les traces, n'introduit pas d'interblocage et qui pourtant ne préserve pas la consistance.

3. (3.1) $\Rightarrow$ (3.3)

   Si G n'est pas vivace alors (3.3) est vrai. Supposons donc G vivace : en augmentant la taille de $I(G)$ on peut obtenir un $O(G)$ aussi grand que l'on veut.

   Supposons que G' ne soit pas vivace. Si G' a un interblocage, sa trace sortante $O(G')$ est bornée : $\exists c, \forall I(G'), \overline{O(G')} \leq c$. Par (3.1), on obtient : $\forall I(G), \overline{O(G)} \leq \overline{O(G')} \leq c$, ce qui contredit l'hypothèse de vivacité de G. D'où, (3.1) $\Rightarrow$ (3.3).

   Puisque l'on a montré en 1., que (3.5) $\Rightarrow$ (3.1), alors par transitivité (3.5) $\Rightarrow$ (3.3).

□

Quelle est l'utilité de l'hypothèse (3.6) ($\exists b \in \mathbb{N}, \forall I(L) = I(R) \Rightarrow \max(\overline{O(R)} - \overline{O(L)}) \leq b$) dans le lemme de correction locale ? En fait cette hypothèse est nécessaire pour prouver la consistance de G' dans le cas avec cycles. Prenons pour exemple la transformation de la figure 3.4 qui ne vérifie pas l'hypothèse (3.6) car R produit chaque fois le double d'éléments que L. Cette transformation préserve les traces, $O(G) \sqsubseteq O(G') \sqsubseteq \{1, 1, \ldots\}$, et n'introduit pas d'interblocages. Pourtant, elle ne préserve pas la consistance, au fur et à mesure des exécutions de G' des éléments vont s'accumuler dans le cycle (il faut donc une mémoire infinie pour exécuter ce programme).

3.3 Transformations simplificatrices

Dans la section précédente nous avons définit la correction d'une transformation et montré comment l'établir indépendamment du contexte de la transformation. Nous allons maintenant construire un **ensemble de transformations correctes**. Nous séparons cet ensemble dans les transformations simplificatrices qui suivent et les transformations restructurantes qui feront l'objet de la section suivante. Les transformations simplificatrices sont des transformations qui simplifient le graphe en supprimant des nœuds, des arcs ou des points de synchronisation.

3.3.1 Élimination des délais

Tout nœud `Split` (resp. `Join`) peut être remplacé par la variante sans-délai de même productions (resp. consommations) en préservant la sémantique.

En effet les variantes sans délai ne changent pas les permutations réalisés mais l'ordre d'exécution des nœuds (dès qu'un élément est reçu il est produit sur la sortie adéquate). Ainsi si on remplace un nœud de routage dans G par la variante sans-délai dans G', l'ordre des éléments sur les traces sortantes sera exactement le même, mais la variante sans-délai pourra produire des éléments en avance. Cette propriété s'écrit,

$$\forall I(G) = I(G') \Rightarrow O(G) \sqsubseteq O(G')$$

On remarque également qu'à la fin de chaque cycle entier, le nœud normal N et le nœud sans-délai $\underline{N}$ auront produit exactement le même nombre de données sur chaque sortie. Soit C le nombre maximum d'éléments produits par sortie sur un cycle. Puisque à chaque fin cycle la différence entre les productions de N et de $\underline{N}$ sont nulles, la différence entre deux cycles ne peut jamais dépasser le nombre d'éléments produits par cycle :

$$\forall I(G) = I(G') \Rightarrow max(\overline{O(G')} - \overline{O(G')}) \leq C.$$

Puisque les variantes sans délai sont également consistantes, on a établit toutes les hypothèses du lemme 3.2.1 et la correction de la transformation suit : on peut remplacer les nœuds de routage par leurs variantes sans-délai.

De cela on déduit que toutes les transformations présentées par la suite qui s'appliquent à des nœuds `Split` ou `Join` sont également correctes sur les variantes sans-délai (même si ce n'est pas systématiquement précisé).

3.3.2 Suppressions

Les transformations les plus simples sont les suppressions, elles consistent à éliminer une partie du graphe qui n'est pas utile pour le calcul. Nous distinguons deux types d'éliminations, celles qui éliminent du code mort : du code qui n'a aucun effet observable, et celles qui éliminent des ensembles de nœuds dont la composition est l'identité.

Élimination de code mort

La première suppression considérée est l'élimination de code mort. Un nœud est mort lorsque :

- c'est un nœud pur : il n'a pas d'autres effets de bords que ses productions ;
- ses sorties sont toutes connectées à des nœud `Sink` ; en effet la sémantique des nœud `Sink` garantit que les sorties ne seront pas observées.

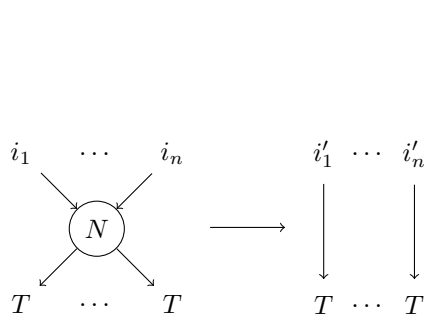
Dans ces conditions un nœud est mort et peut donc être éliminé du graphe en redirigeant ses entrées sur des nœuds `Sink` comme sur la figure 3.5(a) Cette transformation est correcte : les traces sont exactement préservées puisque le nœud supprimé n'agissait pas sur les traces sortantes de G, et R est consistant ; le lemme 3.2.1 nous assure donc la correction.

Cette transformation agit souvent de proche en proche et peut éliminer des pans entiers du graphe. À chaque fois, le nœud supprimé est remplacé par des nœuds `Sink` qui peuvent à leur tour déclencher une élimination de code mort.

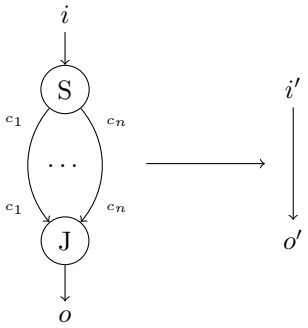
Réduction des identités

Un deuxième type de suppression consiste à éliminer certains couples de nœuds dont l'effet combiné est l'identité. Par exemple la transformation présentée dans l'introduction de ce chapitre *RemoveSJ* (cf. figure 3.1) élimine les couples $S - J$ qui ne changent pas l'ordre des éléments.

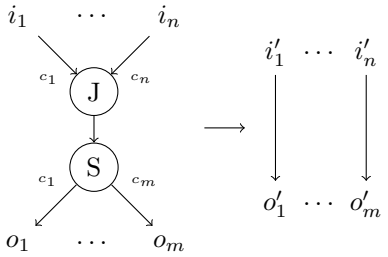
On propose cinq transformations correctes qui éliminent les identités,



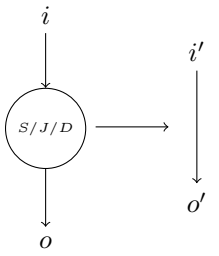
(a) *Deadcode* : Élimination de code mort (N peut remplacer tout nœud pur.)



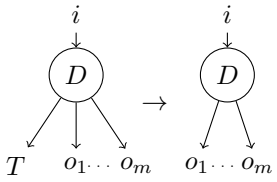
(b) *RemoveSJ*



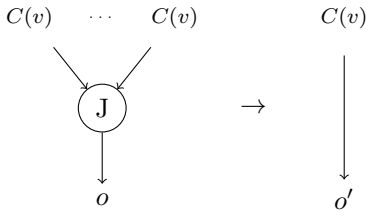
(c) *RemoveJS*



(d) *RemoveI*



(e) *RemoveDT*



(f) *JoinC*

FIGURE 3.5 – Suppressions

- *RemoveSJ* élimine une jonction $S - J$ avec les mêmes consommations et productions, (cf. figure 3.5(b)) ;
- *RemoveJS* élimine une jonction $J - S$ avec les mêmes consommations et productions (cf. figure 3.5(c)) ;
- *RemoveI* élimine un nœud `Split`, `Duplicate` ou `Join` avec une seule entrée et une seule sortie (cf. figure 3.5(d)).
- *RemoveDT* élimine un arc d'un nœud `Duplicate` qui envoie les données vers un puits (cf. figure 3.5(e)) ;
- *JoinC* réunit deux sources constantes qui retournent la même valeur par défaut (cf. figure 3.5(f)).

3.3.3 Compactions

Compaction de nœuds

Comme leur nom l'indique, ces transformations compactent des hiérarchies de nœuds de routage. Par exemple une cascade de nœuds `Duplicate` peut être remplacé par un seul nœud `Duplicate` en utilisant la transformation de la figure 3.6(a). Si un canal i est dupliqué en deux à l'aide d'un premier nœud `Duplicate`, puis que l'une des copies est à nouveau dupliquée par un deuxième nœud `Duplicate`, on obtiendra en tout trois copies de i. On peut tout aussi bien faire directement trois copies à l'aide d'un nœud `Duplicate` d'arité 3.

Les hiérarchies de nœuds `Split` (resp. `Join`) peuvent également être compactées à condition que les productions (resp. consommations) soient aussi en cascade. Sur la figure 3.6(b) un premier nœud `Split` produit $p_1 + p_2 + \ldots + p_k$ données sur le canal alimentant un deuxième nœud `Split`. Ce dernier nœud découpe, à son tour, le flux en blocs de taille $p_1, p_2, \ldots, p_k$. La transformation *CompactSS* remplace les deux `Split` en cascade par un `Split` qui fait directement le découpage au grain le plus fin.

Lorsque la production du premier nœud `Split` n'est pas la somme des productions du deuxième, cette transformation est incorrecte car le comportement cyclo-statique du premier nœud est altéré après fusion : les traces sur les canaux $o_{k+1}, \ldots, o_m$ ne sont pas conservées.

La même transformation existe pour les nœuds `Join`, en remplaçant consommations par productions (cf. figure 3.6(c)).

Compaction de canaux

Considérons deux canaux consécutifs dans un couple $S - J$. Si sur chaque canal, le nœud S produit le même nombre d'éléments par cycle que consomme le nœud J alors on peut fusionner les canaux comme sur la figure 3.7.

Pour que cette transformation soit correcte il faut supprimer les délais dans le couple $S - J$. En effet, en fusionnant les canaux on augmente la production du nœud `Split` sur le canal : par exemple en remplaçant un nœud $S(2, 3, 1)$ par un nœud $S(5, 1)$. Alors que le premier `Split` produit une sortie pour une trace entrante de taille 2, le deuxième `Split` attendra d'avoir reçu 5 éléments avant de produire quelque chose. Cette transformation viole la condition $O(G) \subseteq O(G')$ et est donc incorrecte. Par contre la même transformation est correcte sur des nœuds sans-délai car ceux-ci acheminent les entrées sur les sorties dès réception. Avec des nœuds sans-délai on n'aura pas de rétention des données après transformation.

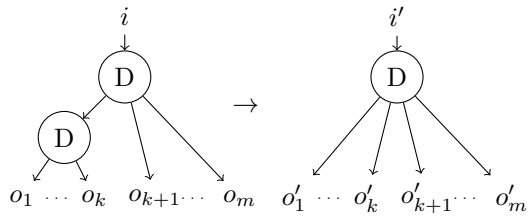
3.3.4 Suppression de synchronisations

Les transformations présentées maintenant sont utiles pour supprimer des points de synchronisation dans le graphe. Par point de synchronisation on entend un point dans le programme où plusieurs flux de données distincts se rencontrent, autrement dit un rendez-vous. À l'exécution, ces rendez-vous imposent une synchronisation des différents flux pour assurer la cohérence des échanges, ce qui peut provoquer des attentes entre plusieurs processeurs.

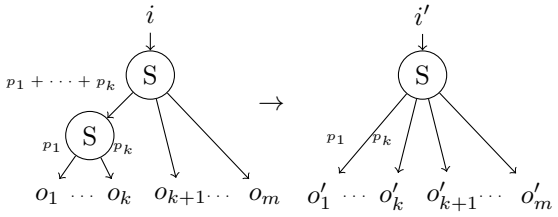
Dans le modèle SJD, les nœuds de routage peuvent créer des rendez-vous entre plusieurs flux. Dans cette section on propose des transformations pour éliminer ou séparer « en petits morceaux » les points de synchronisations de manière à réduire leur coût.

Propagation de constantes

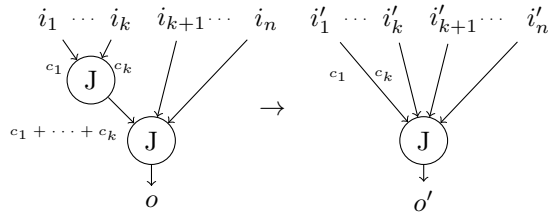
Il peut arriver que des données provenant d'une source constante soient redirigées sur plusieurs canaux à l'aide d'un nœud `Split` ou `Duplicate` comme sur la figure 3.8(a). Puisque la source est constante, les données produites sur les différents canaux sont toutes identiques. Pour éviter la



(a) *CompactDD*



(b) *CompactSS*



(c) *CompactJJ*

FIGURE 3.6 – Transformations compactant des hiérarchies de nœuds de routage. La compaction est toujours représentée sur le premier canal, mais elle peut opérer à n'importe quelle place.

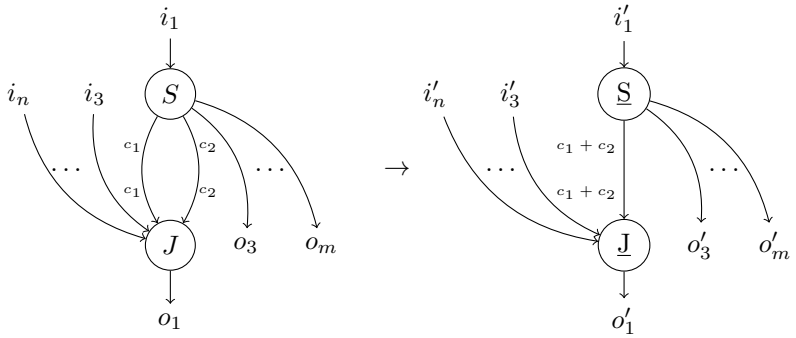


FIGURE 3.7 – La transformation *CoarseSJ* compacte les canaux contigus dans un couple $S - J$.

synchronisation induite par le nœud `Split` ou `Duplicate`, on peut tout simplement dupliquer la source constante et se passer du nœud de routage.

La transformation peut sembler artificielle, il est en effet rare que le programmeur divise une source constante. Par contre durant l'exploration automatique de programmes, l'effet combiné d'autres transformations peut produire ce patron.

Suppression de Synchronisation $J - S$

Les couples $J - S$ permettent de rassembler puis de découper plusieurs flux. Cela permet d'échanger des éléments entre les différents flux. Cependant, il arrive que plusieurs flux soient rassemblés au sein d'un même couple $J - S$ alors qu'ils sont en fait totalement indépendants. On présente des transformations ($SplitJS$,$BreakJ$ et $BreakS$) permettant de séparer explicitement le traitement des flux dans ce cas de figure.

$SplitJS$ casse un couple $J-S$ en deux parties. La transformation est correcte à deux conditions : la somme des consommations de **J** doit être égale à la somme des productions de **S**,

$$\sum_{x=1}^{n} c_x = \sum_{y=1}^{m} p_y = C \tag{3.18}$$

et on doit pouvoir partitionner les consommations et les productions en deux groupes de taille égale,

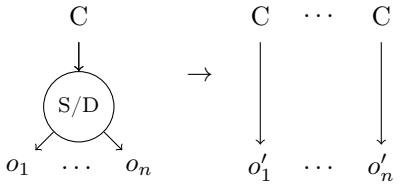
$$\exists (j,k), \sum_{x \leq j} c_x = \sum_{y \leq k} p_k = K \tag{3.19}$$

Le partitionnement sépare les entrées et les sorties en deux faisceaux d'arcs. Puisque **J** et **S** ont les mêmes productions et consommations sur un cycle, le couple admet un ordonnancement stationnaire de vecteur de répétition minimal $[1, 1]$. Sur cet ordonnancement, le nœud `Join` consomme sur le faisceau gauche d'arcs entrants K éléments et $C - K$ sur le faisceau droit ; puis le nœud `Split` envoie K éléments sur le faisceau gauche d'arcs sortants et $C-K$ sur le faisceau droit. Puisqu'il n'y pas de dépendances entre les faisceaux gauches et les faisceaux droits, on peut séparer le routage de données en deux couples $J - S$ (cf. figure 3.8(b)). Cette transformation préserve exactement les traces et est correcte.

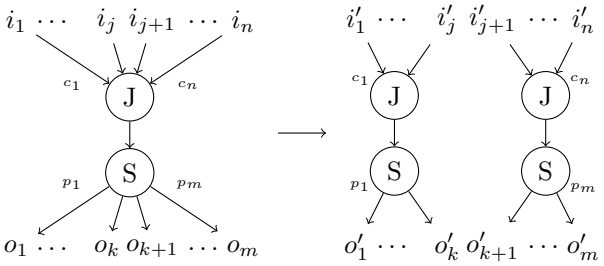
Comme on a dit précédemment, $SplitJS$ ne s'applique qu'aux couples qui admettent une partition des consommations et des productions en groupes de taille égales. Pour les couples $J - S$ qui ne sont pas dans ce cas de figure, nous proposons deux transformations complémentaires. $BreakJ$ et $BreakS$ permettent de séparer les couples $J - S$ qui vérifient (3.18) mais qui ne vérifient pas (3.19).

Examinons le cas d'une jonction $J(4, 1, 1) - S(3, 3)$, les deux nœuds échangent 6 éléments pas cycle, pourtant il n'est pas possible de partitionner les consommations et productions en deux groupes égaux. Cette impossibilité pourrait être levée si l'arc de consommation 4 était cassé en deux arcs de taille 3 et 1. Dans ce cas on pourrait faire le partitionnement $J(\{3\}, \{1, 1, 1\}) - S(\{3\}, \{3\})$. C'est exactement ce que fait la transformation $BreakJ$ (cf. figure 3.8(c)), elle casse l'arc le plus grand en deux arcs plus petits.

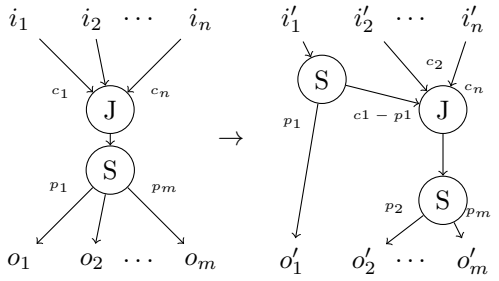
$BreakJ$ ne s'applique que lorsque l'arc le plus grand est sur le nœud `Join`, $c_1 > p_1$. Dans le cas contraire, $p_1 > c_1$, on appliquera plutôt $BreakS$ (cf. figure 3.8(d)) qui fait exactement la même chose mais sur l'arc du nœud `Split`. $BreakS$ et $BreakJ$ sont souvent appliquées à la suite l'une de l'autre. En effet, leur effet combiné égalise de proche en proche les arcs du couple $J - S$ jusqu'à ce que les arcs restants puissent être distribués en deux partitions de taille équivalente (ce qui permettra l'application de $SplitJS$).



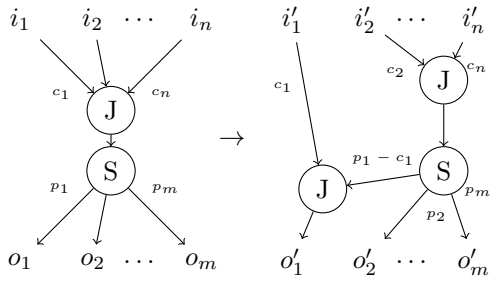
(a) *BreakC*



(b) *SplitJS*



(c) *BreakJ*



(d) *BreakS*

FIGURE 3.8 – Transformations qui suppriment des points de synchronisation.

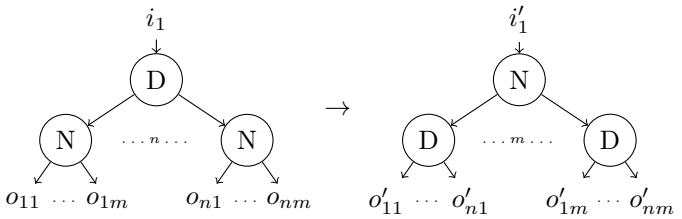


FIGURE 3.9 – La transformation $InvertDN$ factorise plusieurs exécutions identiques du même nœud pur N.

3.4 Transformations restructurantes

Les transformations présentées dans la précédente section rendaient toutes le graphe plus simple : soit en supprimant des nœuds, des arcs ou des points de synchronisation. Les transformations que l'on va présenter maintenant, appelées *restructurantes*, ne simplifient pas forcément les structures de routage ; elles proposent néanmoins des réécritures alternatives qui, selon le programme, sont plus ou moins intéressantes ou qui font apparaître des structures simplifiables par les transformations précédentes.

3.4.1 Factorisation

La première transformation restructurante que l'on va considérer concerne les nœuds `Duplicate`. Supposons que l'on ait, comme sur la figure 3.9, plusieurs copies d'un même nœud pur N qui consomment des entrées identiques.

Puisque N est pur, les productions de chaque copie de N ne dépendent que des entrées. Or les entrées sont exactement les mêmes puisque les nœuds N sont en aval d'un nœud `Duplicate`. Au lieu de répéter le calcul sur chaque copie, on peut faire le calcul une fois et dupliquer le résultat. Cette transformation préserve exactement les traces en sortie et est correcte. La transformation avec $N \equiv D$ ne présente aucun d'intérêt et on se l'interdit, autant faire un $CompactDD$.

Cette transformation est très utile car elle supprime les calculs redondants. Cependant dans certains cas bien spécifiques il peut-être avantageux de faire les calculs séparément (p. ex. coût de communication après calcul bien supérieur au coût de calcul). On reparlera brièvement de ce cas de figure en § 3.6.1.

Factorisation par uniformisation

Rappelons que la factorisation n'est possible que si les enfants de `Duplicate` sont identiques. On peut néanmoins factoriser des nœuds `Split`, dont la somme des productions est égale, en uniformisant les nœuds au préalable de manière très similaire aux transformations $BreakS$ et $BreakJ$. On appelera cette transformation $InvertUniformize$.

Prenons l'exemple de la figure 3.10. Les deux nœuds `Split` ont un cycle de productions différent mais les sommes de leur productions sont identiques. Tout d'abord nous uniformisons les productions des deux nœuds de manière similaire aux transformations $BreakS$ et $BreakJ$. Il suffit de casser alternativement le premier arc d'un des nœuds en un arc plus petit. Sur la figure on a uniformisé les nœuds sous la forme $S(3, 1, 3)$. On se retrouve ainsi avec deux nœuds identiques en aval de D, on peut donc appliquer la factorisation.

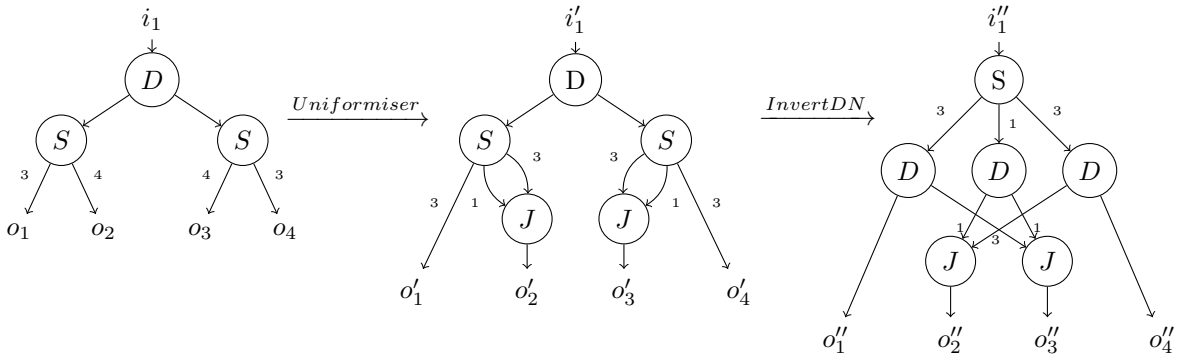


FIGURE 3.10 – La transformation $InvertUniformize$ factorise plusieurs `Split` dont les sommes des productions sont identiques.

3.4.2 Regroupement

Certains programmes traitent de manière indépendante les blocs de données pairs et impairs. Il peut donc être intéressant pour faire apparaître explicitement l'indépendance des deux traitements dans le graphe de séparer explicitement ceux-ci. Pour cela on se sert des transformations de *regroupement* qui séparent les blocs de données selon les restes de congruence modulo d.

Considérons le nœud `Split` de la figure 3.11(a), qui découpe l'entrée en m blocs de taille p. Pour l'exemple on va séparer les blocs pairs et impairs : on prendra $d = 2$. La transformation n'est correcte que si m est divisible par d, dans ce cas il suffit d'introduire un nouveau nœud `Split` qui sépare les blocs impairs (à gauche) et les blocs pairs (à droite). La sortie gauche sera découpée en $m/2$ blocs de taille p qui seront envoyés aux sorties impaires de la forme $o_{2.k-1}$. Les blocs de la sortie droite seront envoyés aux sorties paires de la forme $o_{2.k}$.

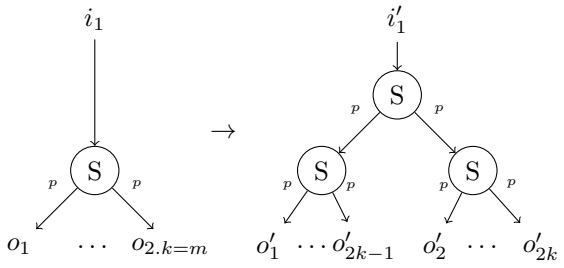
Le mécanisme est exactement le même pour les d plus grands. Les cas $d = 1$ ou $m = d$ ne sont pas très intéressants puisque la transformation est l'identité. La transformation existe aussi pour les nœuds `Join` (cf. figure 3.11(b)). On verra plus tard (§ 3.6.3) une application particulière de ces transformations.

3.4.3 Déroulage

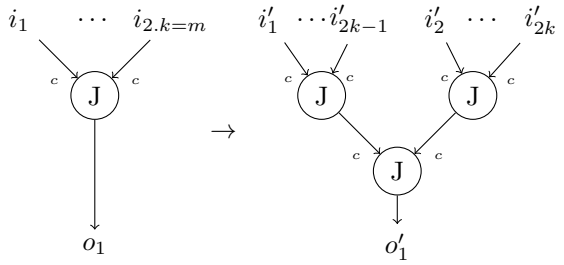
Lorsqu'un nœud d'un programme SJD a fini d'exécuter un cycle, il recommence depuis le début. Les transformations de déroulage consistent à faire apparaître explicitement dans le graphe plusieurs exécutions consécutives du même nœud.

Sur la figure 3.12 on a par exemple déroulé deux fois un nœud `Split`. À l'aide d'un `Split` sans-délai on découpe le flux d'entrée en blocs de la taille des consommation cumulées de S sur un cycle, $\sum_{i=0}^{m} c_i$. Puis on duplique le nœud `Split` que l'on veut dérouler en deux instances : la première reçoit les blocs pairs, la deuxième reçoit les blocs impairs. Enfin, les productions paires et impaires sont rassemblées à l'aide de nœuds `Join` sur chacune des sorties. Les différentes instances dupliquées correspondent aux exécutions successives dans le temps du `Split` original.

Pour séparer l'entrée en différents blocs il est nécessaire d'utiliser un `Split` sans-délai si l'on veut préserver la sémantique sur les entrées dont la taille n'est pas multiple des consommations cumulées sur un cycle. Supposons que l'on ait utilisé un `Split` normal, une entrée de taille p_1 ne produira aucun élément sur la sortie o'_1 de G' alors qu'elle produira p_1 éléments sur la sortie o_1 de G. La transformation est donc incorrecte si l'on utilise des nœuds `Split` normaux pour découper les flux. Par contre, pour les nœuds `Join` en sortie on peut utiliser indifféremment les versions avec



(a) *ReorderS*



(b) *ReorderJ*

FIGURE 3.11 – Les transformations de regroupement séparent explicitement les blocs selon leur congruence $[d]$. Dans cette figure on a représenté les transformations pour $d = 2$.

ou sans délai puisque de toutes façons les données sont produites et consommées en blocs de taille p_i.

On montre que la transformation de déroulage est correcte en utilisant le lemme 3.2.1, en effet :

- Les traces sont exactement préservées lors du déroulage. Nous l'avons montré formellement en vérifiant que les permutations réalisées par R et par L sur un ordonnancement stationnaire sont identiques. Nous omettons les calculs ici car ils sont fastidieux.
- R et L produisent exactement le même nombre d'éléments pour les mêmes entrées ;
- R est consistant.

Bien entendu la transformation de déroulage n'est pas propre aux nœuds `Split`, elle peut être appliquée à tout nœud pur (c'est-à-dire les nœuds de routage et les filtres purs). On donne en figure 3.13 la forme la plus générale de cette transformation. Si les productions sont cyclo-statiques il faut utiliser des nœuds sans-délai pour découper et rassembler les flux, si le cycle ne comporte qu'une seule valeur (comme pour les productions des nœuds `Split`) on peut utiliser les variantes normales. On remarquera que dans la forme générale de la transformation les productions et consommations des nœuds de déroulage sont multipliés par des coefficients k_1, k_2 ; en effet en toute généralité, on n'est pas obligé d'envoyer le même nombre de blocs à chaque nœud. Ainsi on peut envoyer k_1 blocs à la première copie du nœud et $k_2 (\neq k_1)$ blocs à la deuxième copie du nœud.

3.4.4 Transformations dérivées du déroulage

Quel est l'intérêt du déroulage ? D'une part le déroulage permet d'augmenter le parallélisme apparent du graphe puisqu'il fait apparaître explicitement plusieurs exécutions d'un nœud. Il permet donc de transformer du parallélisme de pipeline en parallélisme de tâches (ce qui est utile dans

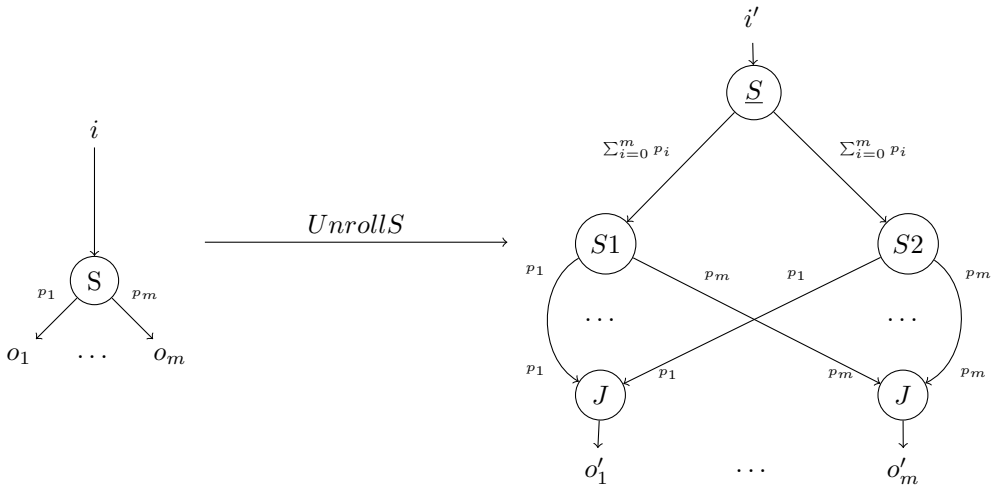


FIGURE 3.12 – Déroulage d'un nœud `Split`.

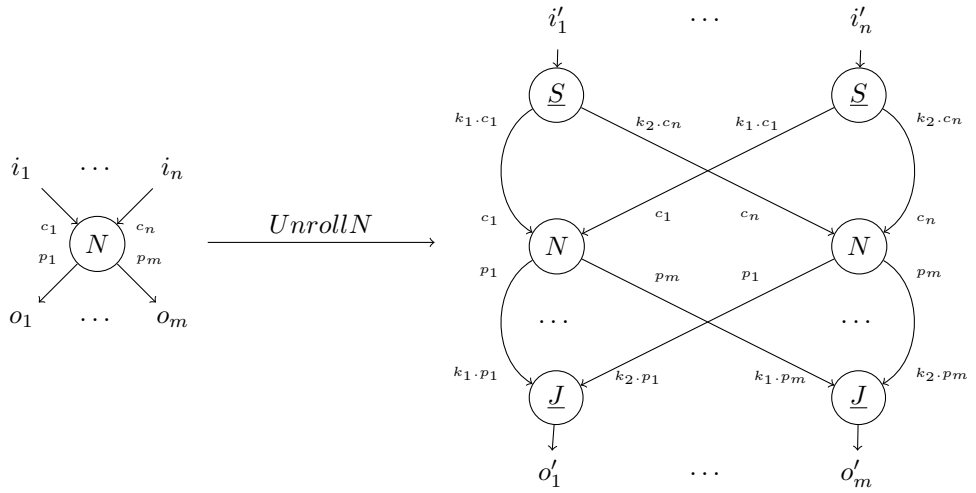


FIGURE 3.13 – Patron général de la transformation de déroulage. Le nœud N peut être remplacé par n'importe quel nœud pur. Les $k_i \in \mathbb{N}$ représentent le nombre de blocs envoyés à chaque instance du nœud déroulé.

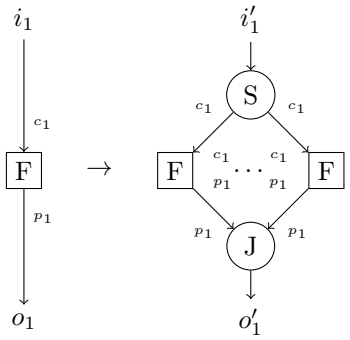


FIGURE 3.14 – Le déroulage de nœuds `Filter` purs *SplitF*, permet de transformer du parallélisme de pipeline en parallélisme de tâches.

le cas des filtres). D'autre part, nous allons montrer dans cette section que le déroulage permet dans certains cas de faire apparaître des simplifications dans le graphe.

Déroulage de filtres : SplitF

On peut dérouler un filtre F pur en appliquant le patron général donné dans la section précédente comme sur la figure 3.14. Cela a pour effet de découper les entrées en plusieurs morceaux, chaque copie de F traitant un des morceaux. Puisque le nœud est pur cela est tout à fait correct : il n'y a ni effets de bords, ni état interne à préserver entre les différentes copies de F.

Cette transformation permet de casser une fonction de calcul en plusieurs morceaux qui peuvent être distribués parmi les différents cœurs de notre architecture. On transforme un parallélisme de pipeline (on pouvait superposer plusieurs exécutions de F dans le temps) en du parallélisme de tâche (on peut distribuer plusieurs exécutions de F sur différents nœuds).

Déroulage de couples $J - S$ avec RemoveJS

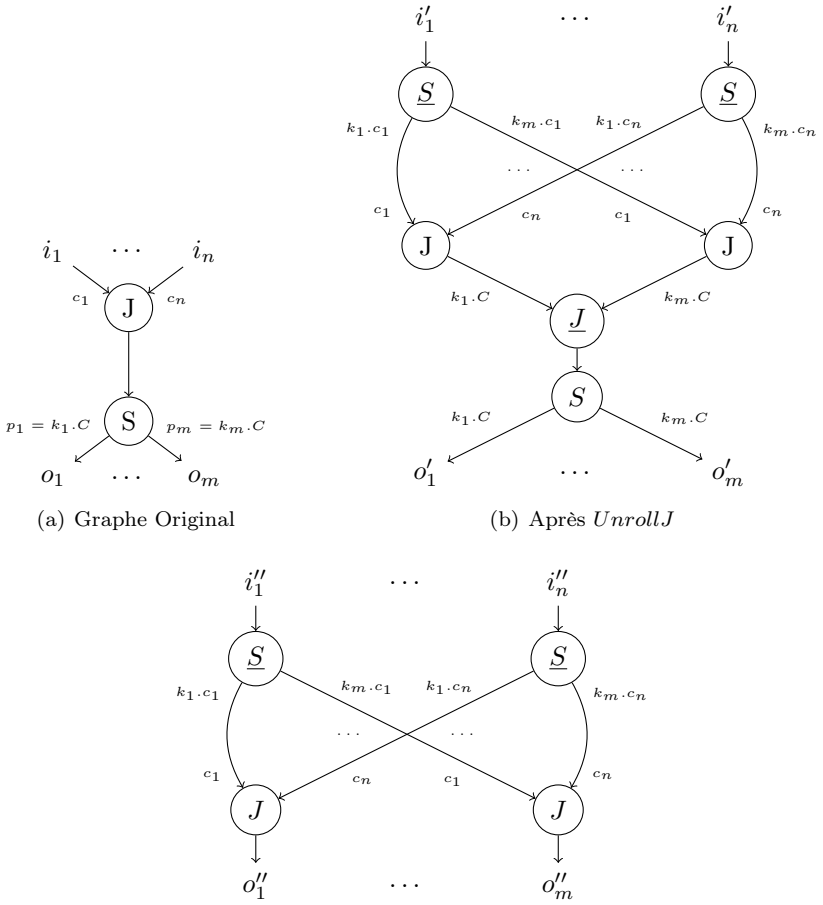
Cette transformation est obtenue en composant deux transformations : le déroulage d'un nœud `Split` ou `Join` et la transformation *RemoveSJ*. Nous allons expliquer la transformation en déroulant le nœud `Join`.

Soit le graphe de la figure 3.15(a), un couple $J-S$ avec des consommations c_i et des productions p_j. La transformation que l'on va construire est possible lorsque les p_j divisent tous la somme des consommations, formellement,

$$\forall j \leq m, \exists k_j \in \mathbb{N} \text{ tel que } p_j = k_j.C \text{ avec } C = \sum_{i=1}^{n} c_i$$

Cette condition signifie que chaque exécution du nœud S consomme les éléments produits sur un ou plusieurs cycles entiers de J. La première étape de cette transformation consiste à dérouler J selon les k_j ci-dessus, comme en figure 3.15(b). Apparaît alors un couple $\underline{J} - S$ de même productions et consommations qui peut donc être supprimé avec la transformations *RemoveJS* ; ce qui produit le graphe en figure 3.15(c).

Les `Split` sans-délai résultants peuvent dans le cadre de cette transformation être remplacés par des `Split` normaux. En effet le délai supplémentaire qu'impliquent les `Split` normaux était déjà présent dans le graphe original à cause du `Split` de consommations $k_i.C$. Les variantes normales garantissent aussi la préservation des traces.



(a) Graphe Original (b) Après $UnrollJ$

(c) Après $RemoveJS$

FIGURE 3.15 – Déroulage de couples $J - S$, $UnrollRemove$. Dans cet exemple on déroule le nœud J.

Cette transformation est aussi possible lorsque les c_i divisent la somme des productions,

$$\forall i \leq n, \exists k_i \in \mathbb{N} \text{ tel que } c_i = k_i.P \text{ avec } P = \sum_{j=1}^{m} p_j$$

Il faut alors dérouler le nœud `Split` à la place du nœud `Join`, comme sur la figure 3.16.

Ces transformations sont intéressantes : en déroulant un des nœuds elles mettent en évidence une simplification possible et permettent d'inverser les étages de `Split` et de `Join`. Dans certains programmes, comme la multiplication matricielle SJD du chapitre précédent, cette inversion permet de paralléliser le graphe avec moins de points de synchronisation (§ 4.4.2).

Déroulage de couples $J - S$ avec BreakJ/BreakS/SplitJS

Dans cette section on présente la transformation $UnrollBreak$. La construction de cette transformation est très semblable à celle de $UnrollRemove$, la différence est que au lieu de composer le déroulage avec la simplification $RemoveJS$, on utilise les transformations de suppression de points de synchronisation $BreakJ$, $BreakS$ et $SplitJS$.

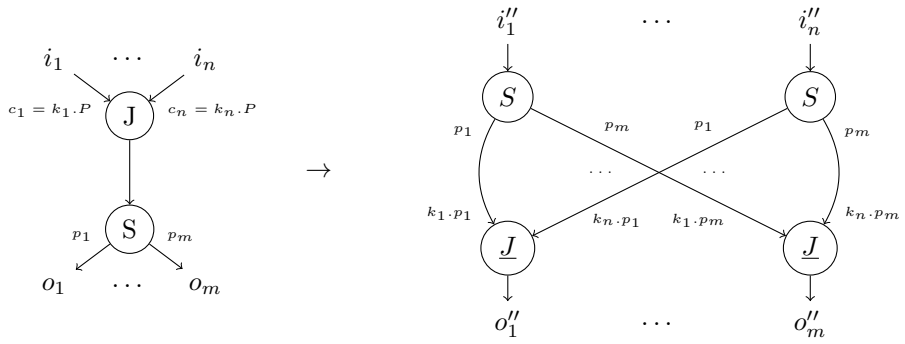


FIGURE 3.16 – Déroulage de couples $J - S$, *UnrollRemove*. On déroule le nœud S.

Soit un couple $J - S$, dont la somme des consommations du `Join` s'écrit C et la somme des productions du `Split` s'écrit P. *UnrollBreak* s'applique aux couples $J - S$ qui vérifient :

$$\exists k \in \mathbb{N}, P = k.C \text{ ou } C = k.P$$

Supposons que l'on soit dans le cas $P = k.C$, dans ce cas on va dérouler le nœud `Join` avec $UnrollJ$ d'un facteur k. Comme sur la figure 3.15(b), on se retrouve avec un nœud de déroulage de la forme $J(k.c_1, \ldots, k.c_n)$. Or $k.c_1 + \ldots + k.c_n = k.C = P$, après transformation on retrouve donc un couple $S - J$ dont les productions et consommations sont les mêmes. Comme on a montré en § 3.3.4, ces couples peuvent être « cassés » à l'aide des transformations $BreakJ$, $BreakS$ et $SplitJS$.

Déroulage de couples $S - J$: FactorSJ

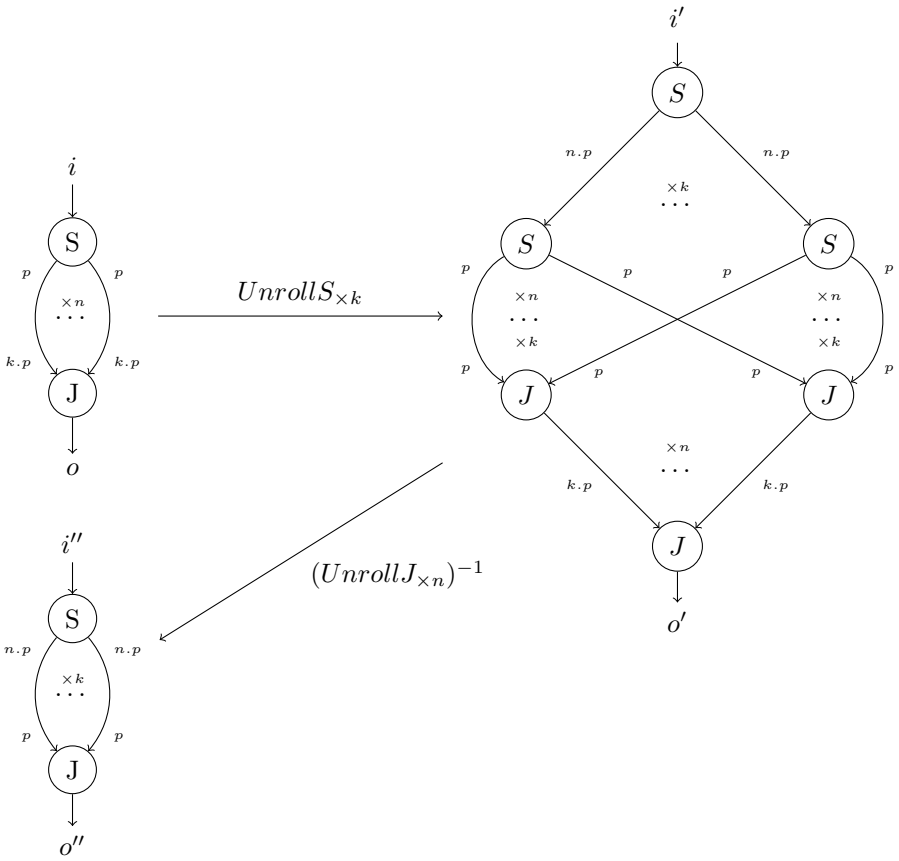
On s'intéresse maintenant au déroulage au sein d'un couple $S - J$. Partons d'un graphe comme celui de la figure 3.17(a) que l'on a déjà utilisé au chapitre 2 pour transposer des images. Ici on remarque que les éléments consommés par le nœud `Join` sont k fois plus nombreux que les éléments produits par le nœud `Split`. Si on déroule le nœud `Split` k fois, on se retrouve avec le graphe intermédiaire de la figure 3.17(a). Les rôles de k et de n sont symétriques dans le graphe intermédiaire, on reconnaît en effet la partie droite de la transformation $UnrollJ_{\times n}$, que l'on peut appliquer dans le sens inverse (on pourrait parler d'enroulage).

L'inverse des transformations ne satisfait pas toujours les hypothèses du lemme de correction 3.2.1. En effet les transformations pour lesquelles $O(G)sqsubsetO(G')$, ne peuvent pas être utilisées dans le sens contraire : la transformée d'un nœud `Split` en son homologue sans-délai est correcte mais la transformée inverse est incorrecte. Cependant, les transformations de déroulage, qui nous intéressent ici, préservent exactement les traces ($O(G') = O(G)$). Les deux sens sont donc corrects. On peut ainsi intervertir les rôles joués par n et k dans le graphe intermédiaire. Cette transformation est particulièrement intéressante lorsque $n > k$, puisqu'elle réduit alors le nombre de canaux nécessaires pour réaliser la transposition de données.

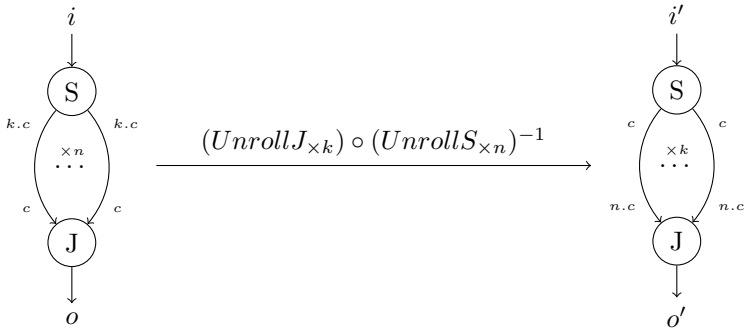
La même transformation existe pour le cas inverse où le nœud `Split` consomme k fois le nombre d'éléments produits par `Join` (représentée en figure 3.17(a)) ; en fait c'est l'inverse de la première transformation.

3.4.5 D'autres transformations ?

Pour construire les transformations restructurantes et simplificatrices que nous avons présentées, nous avons étudié des transformations locales entre un nœud et ses enfants et un nœud et ses parents. Nous avons fait le choix de nous intéresser aux transformations locales, et de transformer



(a) Déroulage de S



(b) La transformation inverse est également correcte

FIGURE 3.17 – Déroulage de couples $S - J$: $FactorSJ$.

le graphe de proche en proche. Mais l'analyse de la structure globale du graphe permet peut-être d'appliquer des transformations fort intéressantes. De la même manière, nous considérons que les nœuds `Filter` sont des boîtes noires qui sont pures ou impures. Mais il existe des transformations basées sur l'analyse du code de la fonction f associée au nœud `Filter` : par exemple StreamIt propose des optimisations spécifiques pour les fonctions f affines très intéressantes.

Ainsi, l'ensemble de transformations présenté ici n'est pas un catalogue exhaustif de toutes les transformations correctes sur les SJD. Il représente néanmoins un ensemble de transformations *utiles* pour générer des implémentations alternatives d'un programme SJD.

3.5 Optimisation des graphes générés par SLICES

La méthode que nous avons présentée dans le chapitre 2 compile des graphes SJD corrects et dont la complexité en nœuds et arcs est maîtrisée. Cependant le compilateur SLICES ne considère pas les optimisations possibles entre deux étages distincts. Or dans certains programmes il est possible de fusionner deux étages en simplifiant le graphe.

Simplification des graphes Nous ajoutons en sortie du compilateur SLICES un module de simplification qui va appliquer les transformations simplificatrices au graphe SJD produit de manière à débarasser celui-ci des nœuds inutiles. Prenons pour exemple le graphe généré pour le filtre Gaussien, que l'on reprend sur la figure 3.18. On peut regrouper les deux étages d'extraction de région en cascade en utilisant la transformation $CompactJJ$. On peut également fusionner l'étage d'extraction de blocs inférieur avec l'étage de tri en utilisant la transformation $RemoveJS$. De cette manière on se ramène à une version qui n'a nul besoin d'un étage de transposition explicite (contrairement à la version SJD de la § 2.3.6).

Des simplifications similaires sont aussi possibles sur les graphes d'extractions des triplets horizontaux et verticaux.

Transformation des graphes Pour simplifier les graphes de notre compilateur nous n'utilisons que les *transformations simplificatrices* puisque ces dernières ne font que supprimer des nœuds ou des arcs. Les *transformations restructurantes* sont plus complexes : dans certains cas elles peuvent aboutir à un graphe plus performant alors que dans d'autres elles peuvent le rendre moins efficace. C'est pourquoi il faut trouver un moyen de choisir un ensemble de transformations restructurantes optimisant, c'est l'objet du § 3.7. Nous allons cependant ici donner un exemple où les transformations restructurantes permettent de simplifier le graphe : la séparation des données paires et impaires pour la FFT.

Rappelons que le graphe produit pour un étage de séparation pair/impair de la FFT utilisait deux branches : une pour décimer les données impaires et l'autre pour décimer les données paires. Le compilateur SLICES ne fait pas d'optimisation entre des motifs différents, il n'a donc pas cherché à fusionner les deux branches. Avec les transformations proposées ci-dessus il est possible de simplifier l'étage de réorganisation par la dérivation en figure 3.19. Nous retrouvons ainsi un simple $S-J$ dont les branches suivent l'ordre de la permutation réalisée (l'ordre n'apparait pas sur la figure). Les transformations de factorisation ($InvertUniformize$ et $InvertND$) ont permis de faire remonter les opérations similaires entre la branche paire et impaire dans le graphe. Puis avec $UnrollJS$ on a déroulé les executions de la branche impaire et paire sur un vecteur ; ce qui a rendu explicite que la décimation des données sur la branche paire correspond aux données nécessaires sur la branche impaire.

Dans la suite nous montrons comment explorer les dérivations des transformations restructurantes et simplificatrices de manière à permettre des simplifications semblables.

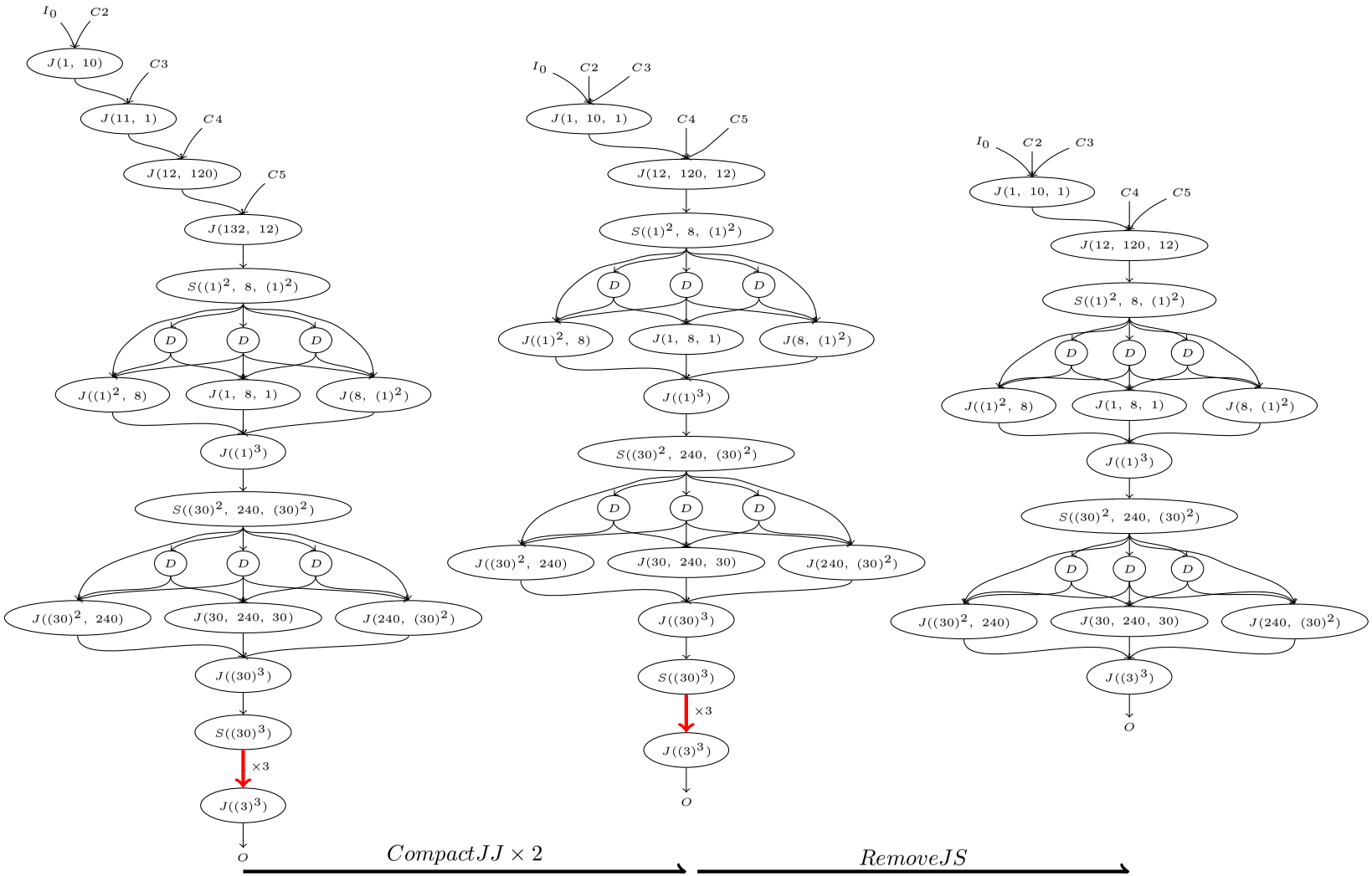


FIGURE 3.18 – Simplification du graphe généré par SLICES pour le filtre Gaussien.

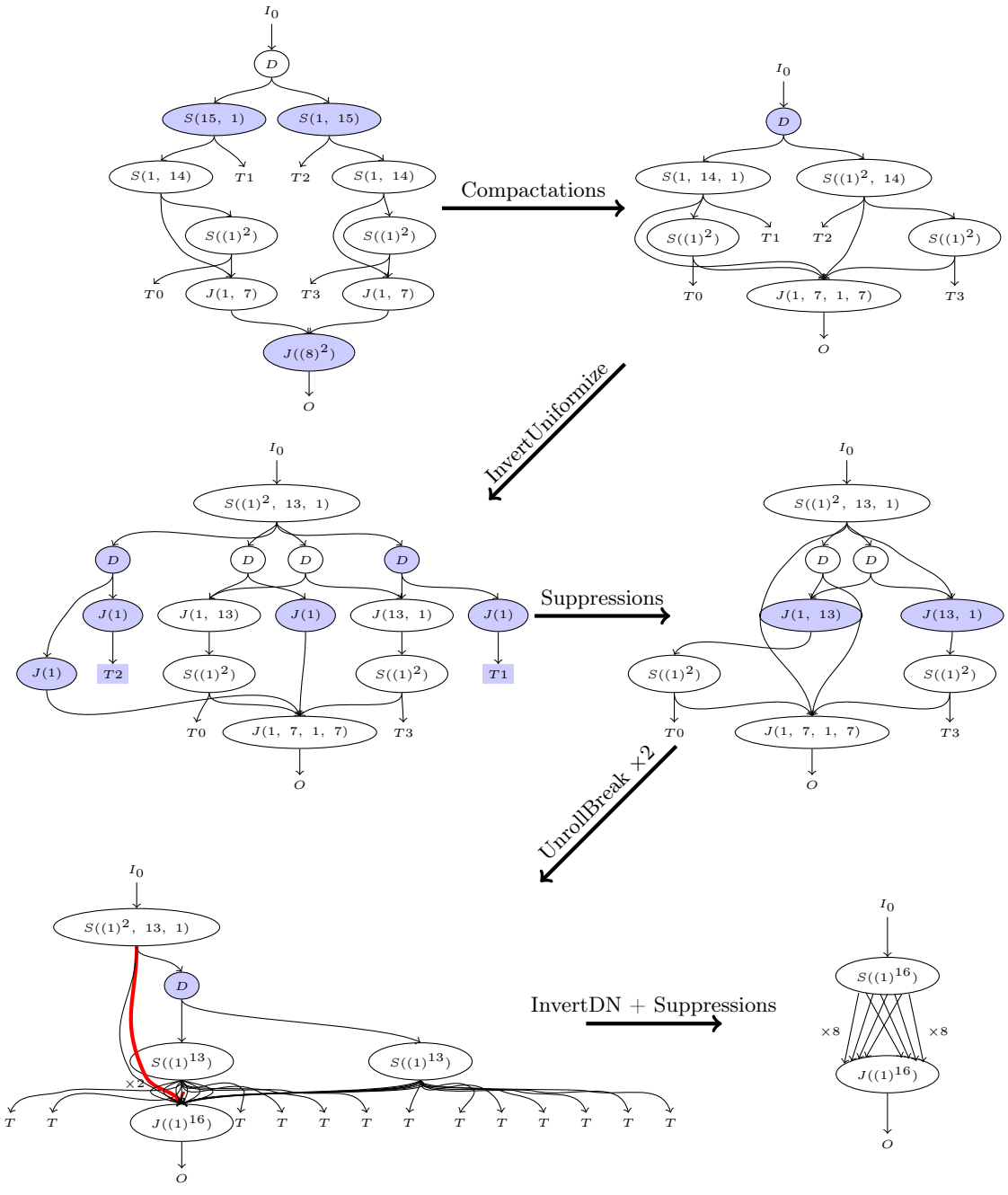


FIGURE 3.19 – Transformation du graphe généré pour l'étage de réorganisation des données de la FFT.

3.6 Espace d'exploration engendré

On a défini un ensemble de transformations correctes sur les graphes SJD. On souhaite maintenant étudier l'espace engendré par leurs dérivations qui contient différentes implémentations sémantiquement équivalentes au programme d'origine. Notre but étant d'explorer cet espace pour trouver le graphe qui optimise la fonction économique ϕ.

On rencontre trois types de problèmes pour explorer l'espace engendré qui seront traités dans les sections suivantes :

- Considérer une transformation et son inverse lors de l'exploration peut-être problématique car l'exploration pourrait boucler.
- L'espace engendré est infini : en effet certaines transformations peuvent augmenter la taille du graphe arbitrairement. Par exemple, on peut dérouler un nœud autant de fois que l'on veut.
- Certaines transformations ($ReorderS, ReorderJ, Unroll$) sont paramétrables, il faut par exemple pour $ReorderS$ choisir le paramètre d de congruence. Or il existe à priori un nombre infini de paramètres possibles.

On va montrer que l'espace d'exploration est fini sous certaines contraintes d'application des transformations.

3.6.1 Sens des transformations

Comme nous l'avons vu en § 3.4.4, certaines transformations sont inversibles, c'est-à-dire correctes dans les deux sens. Les transformations inversibles sont (lorsque l'on ne remplace pas les nœuds de routage par les variantes sans-délai) : $BreakC$, $SplitJS$, $ReorderS$, $ReorderJ$, $Unroll$, $SplitF$, $UnrollRemove$, $FactorSJ$, $InvertDN$, $InvertUniformize$.

Le fait de considérer une transformation et son inverse ne rend pas forcément l'espace engendré infini, mais complique considérablement l'exploration. En effet, on peut construire des dérivations infinies : $T \circ T^{-1} \circ T \circ T^{-1} \circ \ldots$. On pourrait interdire l'application juxtaposée d'une transformation et de son inverse, mais cela ne résoudrait pas pour autant le problème car on pourrait avoir des dérivations de la forme $T_0 \circ T_1 \circ \ldots \circ T_0^{-1} \circ \ldots$. Si nous souhaitons prendre en compte les transformations inverses dans l'exploration, il nous faut garder une trace des graphes déjà visités durant l'exploration pour s'assurer de ne pas « tourner en rond ». Cela est possible, mais coûteux : établir un isomorphisme de graphe est un problème non trivial [1][RC77]. Notre premier prototype considérait les deux sens des transformations durant l'exploration. Pour garder la trace des graphes rencontrés nous utilisions la méthode de hashage de graphe présentée dans [Por08] qui nous évitait d'avoir à résoudre l'isomorphisme de graphe entre le graphe courant et tous les graphes déjà visités. L'approche était néanmoins coûteuse. Nous nous sommes donc demandés s'il est toujours utile de considérer les deux sens d'une transformation ?

- Pour les transformations $BreakC$ et $SplitJS$, le sens inverse ne présente aucun intérêt : en effet, pourquoi rajouter des nœuds de synchronisation supplémentaires au graphe ?
- Pour la transformation $FactorSJ$ qui réduit le nombre d'arcs dans un couple $S-J$, les deux sens sont utiles. Par contre pour chaque couple $S-J$ seul le sens qui réduit le nombre d'arêtes $k < n$ nous intéresse. Donc il n'y a pas de problème à considérer les deux sens (puisque seul un des deux sera employé pour chaque couple $S-J$).

1. à ce jour il n'existe pas d'algorithme polynomial ni de preuve de NP-complétude pour l'isomorphisme de graphes.

- Les transformations $ReorderS^{-1}$, $ReorderJ^{-1}$, $Unroll^{-1}$, $SplitF^{-1}$ et $UnrollRemove^{-1}$ regroupent des nœuds ensemble en grossissant le grain du parallélisme. Cela peut-être utile lorsque le degré de parallélisme exprimé est supérieur au nombre de cœurs disponibles ; il peut-être intéressant alors de fusionner les tâches qui sont sur le même cœur de manière à amortir le coût de changement de contexte d'une tâche. Cependant la réduction de parallélisme est une opération facile ; il suffit une fois que le placement des tâches est connu de fusionner les tâches qui sont affectées au même cœur. Dans notre approche nous avons fait le choix de ne pas considérer ces transformations durant l'exploration de l'espace d'implémentation ; et de réaliser la fusion des tâches plus tard dans la chaîne de compilation. Nous présenterons l'étape de fusion des tâches de notre compilateur en § 4.1.6.

- Finalement, la transformation $InvertDN^{-1}$ et ses dérivées, présentent généralement peu d'intérêt puisqu'elles ajoutent des calculs redondants au graphe de l'application. Il y a cependant un cas où cela peut se révéler utile : lorsque le coût de la communication après calcul est bien supérieur au coût du calcul lui même ; il peut être alors préférable de faire des calculs redondants pour diminuer les coûts de communication. Nous avons décidé dans notre compilateur pour éviter d'avoir à prendre en compte les transformations inverses (et donc de devoir calculer et conserver des fonctions de hashage de tous les graphes visités) de ne **pas considérer cette possibilité**.

3.6.2 Déroulage et finitude de l'espace engendré

Maintenant que nous avons résolu le problème du sens des transformations, nous allons interdire certaines transformations de manière à rendre l'espace d'exploration engendré fini.

La transformation qui rend l'espace d'exploration infini est le déroulage. En effet, le déroulage peut s'appliquer à n'importe quel nœud et permet d'augmenter arbitrairement le nombre de nœuds du graphe. On pourrait tout simplement interdire les transformations de déroulage, pourtant les transformations issues du déroulage sont souvent intéressantes.

En § 3.4.4 nous avons présenté des transformations qui correspondaient à des utilisations particulièrement utiles du déroulage :

- $SplitF$, pour transformer du parallélisme de pipeline en parallélisme de tâches.

- $UnrollRemove$ et $UnrollBreak$, pour éliminer des points de synchronisation dans les couples $J - S$.

- $FactorSJ$, pour diminuer le nombre d'arcs dans les couples $S - J$.

La construction de ces transformations n'est pas innocente ; ce sont des applications du déroulage qui nous permettent de conserver un espace d'exploration fini. Nous arrivons ainsi a un compromis : on interdit la transformation générale de déroulage, $UnrollN$, responsable de l'infinitude de notre espace d'exploration ; néanmoins on conserve les applications les plus intéressantes du déroulage.

Comment se fait-il que ces transformations ne génèrent pas un espace d'exploration infini ? Nous allons donner une première explication intuitive ; la preuve formelle de finitude sera l'objet de la § 3.6.4.

$SplitF$ crée un espace infini puisque c'est $UnrollN$ appliquée aux nœuds `Filter`. $SplitF$ est paramétrable, en effet on peut décider du nombre de fois où l'on déroule le filtre. Or diviser un filtre en deux, puis diviser chacune des copies en deux, c'est équivalent à diviser le filtre en quatre. Sans limiter le nombre de variantes engendrées, nous interdisons donc à un filtre d'être divisé plus d'une fois par $SplitF$. Par ailleurs en § 3.6.3 nous montrons que pour les transformations paramétrables seul un nombre fini de paramètres est considéré.

$FactorSJ$ et $UnrollRemove$, garantissent toutes les deux la finitude de l'espace d'exploration car ce sont la composition d'une transformation qui augmente le nombre de nœuds ($UnrollN$) avec

une transformation qui le diminue ($UnrollN^{-1}$ ou $RemoveJS$). L'effet « explosif » de $UnrollN$ est ainsi contrebalancé par une transformation réductrice ce qui garantit que l'espace engendré reste fini.

3.6.3 Comment paramétrer les transformations ?

Le dernier problème que l'on rencontre est le choix des paramètres dans les transformations paramétrables. Les transformations paramétrables sont $SplitF$, $ReorderS$ et $ReorderJ$. Nous ne pouvons pas essayer tous les paramètres ($\mathbb{N}$ est infini), il nous faut donc choisir les plus intéressants.

SplitF

La transformation $SplitF$ va augmenter le parallélisme de tâches dans l'application. Le paramètre de la transformation $SplitF$ est fixé librement par le concepteur en fonction du degré de parallélisme qu'il veut obtenir. Lorsque le degré de parallélisme est supérieur au nombre de cœurs disponibles dans l'architecture ; celui-ci sera réduit par fusion (§ 4.1.6).

ReorderS et ReorderJ

Ces deux transformations permettent d'éliminer des points de synchronisation en séparant explicitement des blocs d'éléments par leur congruence de modulo d. Puisqu'il n'est pas possible de considérer toutes les valeurs de d, nous adoptons la stratégie suivante pour le choix de d :

- Pour les nœuds `Split`, nous choisirons d égal au nombre de nœuds en aval de S ; si par exemple le nœud `Split` alimente trois nœuds, on s'intéressera au regroupement par congruence 3 ; dans nos expériences c'est un choix qui déclenche souvent des optimisations intéressantes ;
- De manière similaire, pour les nœuds `Join`, nous choisirons d égal au nombre de nœuds en amont de J ;
- Finalement, on considérera un cas particulier que l'on rencontre fréquemment. Lorsque l'on rencontre un couple $J-S$ dont le nœud J consomme c éléments sur chacune de ses n entrées et dont le nœud S produit c éléments sur chacune de ses m sorties, avec $d = pgcd(n, m) \neq 1$; on peut effectuer la simplification représentée en figure 3.20.

3.6.4 Preuve de finitude de l'espace engendré

Dans la section précédente nous avons borné l'espace engendré par les transformations en faisant des hypothèses restrictives : nous interdisons la transformation de déroulage (en gardant tout de même les transformations qui en sont issues), nous ne considérons que le sens le plus utile des transformations inversibles, finalement pour les transformations paramétrées nous choisissons un ensemble fini de paramètres intéressants.

Nous allons maintenant prouver formellement, qu'avec les hypothèses ci-dessus, l'espace de transformation engendré est fini. Nous nous servons pour cela d'un ordre bien fondé sur les multi-ensembles [DM79] que nous rappelons ci-dessous.

Ordre bien fondé dans les multi-ensembles

On considère d'abord l'ensemble $\mathbb{N} \cup \{+\infty\}$, qui est l'ensemble des naturels auquel on adjoint un élément $+\infty$ plus grand que tous les autres. Cet ensemble possède un ordre bien fondé $<$.

On note $\mathcal{M}(\mathbb{N} \cup \{+\infty\})$ l'ensemble des multi-ensembles finis d'éléments de $\mathbb{N} \cup \{+\infty\}$. Un multi-ensemble (parfois appelé sac) est tout simplement une liste finie et non ordonnée d'éléments ;

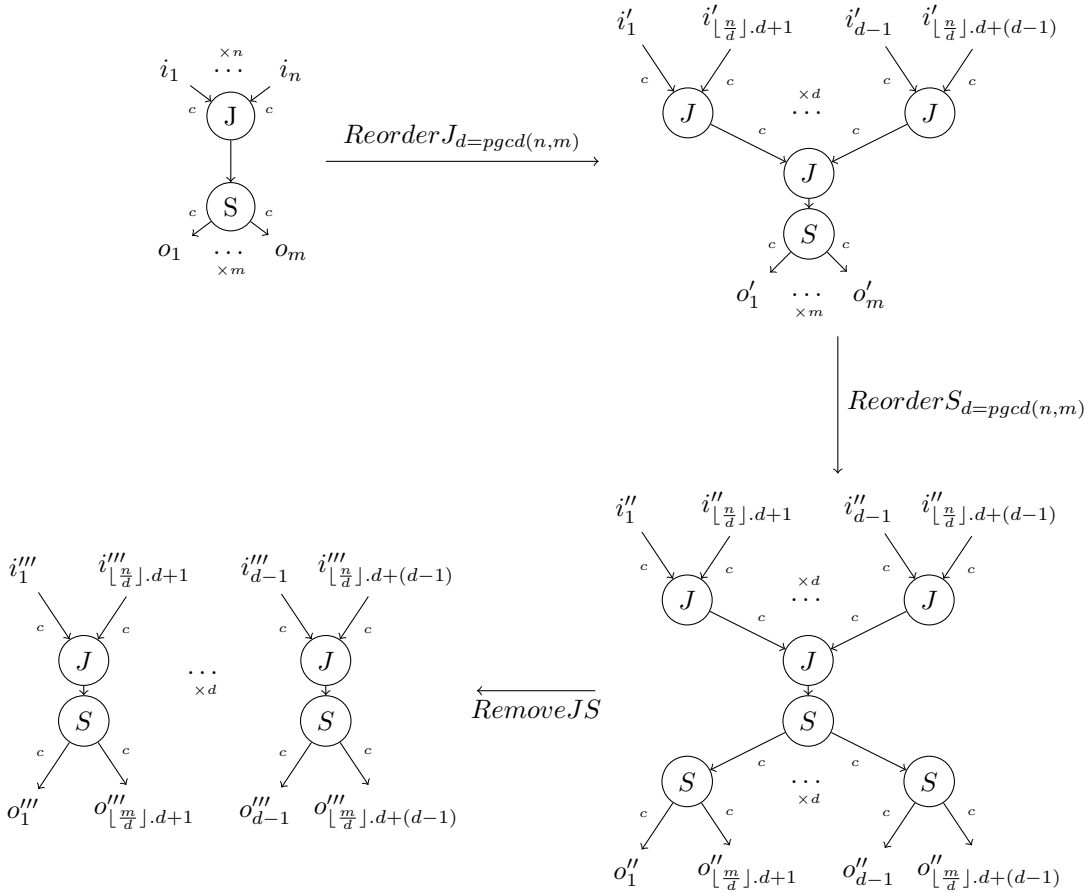


FIGURE 3.20 – *ReorderJS* : lorsque les nœud S et J travaillent sur des blocs de même taille, il est possible de simplifier leur expression en réordonnant les éléments selon le plus grand commun diviseurs des périodes $pgcd(n, m)$.

contrairement à un ensemble un multi-ensemble peut contenir plusieurs copies du même élément. Par exemple $A = \{10, 0, 0\}$ est un multi-ensemble de $\mathbb{N} \cup \{+\infty\}$.

Dershowitz montre [DM79] qu'il est possible de définir un ordre bien fondé partiel $\sqsubset$ sur $\mathcal{M}(\mathbb{N} \cup \{+\infty\})$ de la façon suivante, $\forall M, N \in \mathcal{M}(\mathbb{N} \cup \{+\infty\}), M \sqsubset N$ si pour les multi-ensembles $X, Y \in \mathcal{M}(\mathbb{N} \cup \{+\infty\})$, où $X \neq \{\}$ et $X \subseteq M$, on a

$$N = (M/X) \cup Y$$

et

$$\forall (x, y) \in (X, Y), x > y.$$

En d'autres termes on peut réduire un multi-ensemble en enlevant un élément ou en le substituant par un ou plusieurs plus petits éléments. Par exemple $B = \{9, 8, 7, 7, 0, 0\}$ est plus petit que A puisque l'on a enlevé l'élément 10 et l'on a ajouté les éléments plus petits $9, 8, 7, 7$.

Poids d'un graphe

Nous définissons la fonction σ qui à chaque nœud associe un poids dans $\mathbb{N} \cup \{+\infty\}$,

$$\sigma(N) = \begin{cases} \sum_{k=1}^{n} c_k & \text{si } N = \mathbf{J}(c_1 \dots c_n) \\ \sum_{k=1}^{m} p_k & \text{si } N = \mathbf{S}(p_1 \dots p_m) \\ +\infty & \text{si } N = \mathbf{F} \text{ jamais déroulée} \\ 1 & \text{si } N = \mathbf{F} \text{ déroulée} \\ 0 & \text{sinon} \end{cases}$$

Nous étendons cette fonction aux graphes par la fonction ρ : Graphes $\mapsto (\mathcal{M}(\mathbb{N} \cup +\infty), \sqsubset)$ définie de la manière suivante, $\rho(G) = \{\sigma(n) : \forall n \text{ nœud de G}\}$.

Nous définissons également la fonction $\eta(G)$: Graphes $\mapsto \mathbb{N}$ telle que $\eta(G)$ soit le nombre d'arcs du graphe G.

Nous posons enfin la fonction $\tau(G)$: Graphes $\mapsto (\mathcal{M}(\mathbb{N} \cup \{+\infty\}) \times \mathbb{N}, \prec)$ définie par $\tau(G) = (\rho(G), \eta(G))$. L'ordre $\prec$ est construit comme l'ordre lexicographique sur $\mathcal{M}(\mathbb{N} \cup +\infty) \times \mathbb{N}$ dérivé de $\sqsubset$ et de $<$. Il est bien fondé, puisque $\sqsubset$ et $<$ le sont, et c'est un ordre partiel puisque $\sqsubset$ est partiel.

Transformation contractante

Définition 3.6.1 (Transformation contractante)**.** *Une transformation T est contractante* ssi $G \xrightarrow{T} G' \Rightarrow \tau(G') \prec \tau(G)$.

Autrement dit une transformation est contractante dans deux cas exactement :

- quand elle diminue le poids du graphe en substituant un groupe de nœuds par un groupe de nœuds de poids strictement inférieur ;
- quand elle ne diminue ni augmente le poids du graphe mais diminue le nombre d'arcs.

Lemme 3.6.1. *Si $\tau(R) \prec \tau(L)$ alors T est contractante.*

Démonstration. Par définition $G \backslash L \equiv G' \backslash R$, donc le poids et le nombre d'arcs de $G \backslash L$ sont invariants par T. □

Nous allons montrer que toutes les transformations, à l'exception de $UnrollN$, avec les hypothèses restrictives sur le sens et les paramètres, sont contractantes. Pour cela nous allons écrire les effets des transformations sur $\tau(R)$ et $\tau(L)$. Nous écrirons tout d'abord les effets des transformations sur les nœuds sans considérer la structure du graphe.Pour une transformation $G \xrightarrow{T} G'$ nous écrirons les nœuds de L à gauche et les nœuds de R à droite que nous

séparons avec le symbole $\vdash$. Puis nous écrirons en dessous les multi-ensembles $\rho(R), \rho(L)$ associés et montrerons que $\rho(L) \sqsubset \rho(R)$. Lorsque $\rho(R), \rho(L)$ ne sont pas comparables nous montrerons que $\eta(R) < \eta(L)$. Nous abuserons de l'opération puissance pour noter un nœud qui apparait plusieurs fois dans un sous-graphe et un poids qui apparait plusieurs fois dans un multi-ensemble. Nous montrerons pour chaque transformation que $\tau(R) \prec \tau(L)$.

RemoveSJ, RemoveJS, RemoveI, RemoveDT, JoinC Ces transformations ne font que supprimer des nœuds, elles sont donc contractantes.

Deadcode

$$\begin{array}{rcr} \mathbf{NT}^m & \vdash & \mathbf{T}^n \\ \{\sigma(N), (0)^m\} & \sqsupset & \{(0)^n\}\mathbf{C}(d_m) \end{array}$$

La transformation supprime un nœud **N** pur avec m sorties et n entrées. Deux cas se présentent :

- Si $\mathbf{N} = \mathbf{D}$, alors par construction $n = 1$ et $m >= 1$ et $\sigma(D) = 0$. On supprime, au moins, un nœud de poids 0.
- Sinon $\sigma(N) > 0$ et on remplace un nœud de poids strictement positif par plusieurs nœuds de poids nul.

T est donc contractante.

CompactDD

$$\begin{array}{rcr} \mathbf{D}(n)\mathbf{D}(m) & \vdash & \mathbf{D}(n+m) \\ \{(0)^2\} & \sqsupset & \{0\} \end{array}$$

La transformation est contractante.

CompactSS

$$\begin{array}{rcr} \mathbf{S}(p_1 \dots p_m)\mathbf{S}(p_1 + \dots + p_m p_{m+1} \dots p_{m'}) & \vdash & \mathbf{S}(p_1 \dots p_{m'}) \\ \{(p_1 + \dots + p_m), (p_1 + \dots + p'_m)\} & \sqsupset & \{(p_1 + \dots + p'_m)\} \end{array}$$

La transformation est contractante. La preuve pour **CompactJJ** est la même.

CoarseSJ

$$\begin{array}{rcr} \mathbf{S}(c_1 c_2 p_3 \dots p_m)\mathbf{J}(c_1 c_2 c_3 \dots c_n) & \vdash & \mathbf{S}(c_1 + c_2 p_3 \dots p_m)\mathbf{J}(c_1 + c_2 c_3 \dots c_n) \\ \{(c_1 + c_2 + p_3 + \dots + p_m), (c_1 + \dots + c_n)\} & = & \{(c_1 + c_2 + p_3 + \dots + p_m), (c_1 + \dots + c_n)\} \\ \eta(L) = (n-2) + (m-2) + 2 & > & \eta(R) = (n-2) + (m-2) + 1 \end{array}$$

La transformation est contractante : même si les poids de L et R sont égaux, le nombre d'arcs de R est plus petit.

BreakC

$$\begin{array}{rcr} \mathbf{CS}(p_1 \dots p_m) & \vdash & \mathbf{C}^m \\ \{0, (p_1 + \dots + p_m)\} & \sqsupset & \{(0)^m\} \end{array}$$

La transformation est contractante.

SplitJS

$$\begin{array}{rcl} \mathbf{J}(c_1 \dots c_n)\mathbf{S}(p_1 \dots p_m) & \vdash & \mathbf{J}(c_1 \dots c_j)\mathbf{J}(c_{j+1} \dots c_j)\mathbf{S}(p_1 \dots p_k)\mathbf{S}(p_{k+1} \dots p_m) \\ \{(\sum_{i=1}^{n} c_i), (\sum_{i=1}^{m} p_i)\} & \sqsupset & \{(\sum_{i=1}^{j} c_i), (\sum_{i=j+1}^{n} c_i), (\sum_{i=1}^{k} p_i), (\sum_{i=k+1}^{m} p_i)\} \end{array}$$

$(\sum_{i=1}^{n} c_i)$ est plus grand que $(\sum_{i=1}^{j} c_i)$ et que $(\sum_{i=j+1}^{n} c_i)$. On observe la même chose pour les p_i. Les éléments de L ont été substitués par plusieurs éléments plus petits : la transformation est contractante.

BreakS

$$\begin{array}{rcl} \mathbf{J}(c_1 \dots c_n)\mathbf{S}(p_1 \dots p_m) & \vdash & \mathbf{J}(c_1(p_1 - c_1))\mathbf{J}(c_2 \dots c_n)\mathbf{S}((p_1 - c_1)p_2 \dots p_m) \\ \{(c_1 + \dots + c_n), (p_1 + \dots + p_m)\} & \sqsupset & \{(p_1), (c_2 + \dots + c_n), (p_1 + \dots + p_m - c_1)\} \end{array}$$

La transformation est contractante puisque l'on a remplacé le poids $(c_1 + \dots + c_n)$ par le poids plus petit $(c_2 + \dots + c_n)$ et l'on a remplacé le poids $(p_1 + \dots + p_m)$ par les poids plus petits (p_1) et $(p_1 + \dots + p_m - c_1)$. La démonstration est la même pour $BreakJ$.

InvertDN Il existe trois cas de figure pour la transformation $InvertDN$. N peut être au choix un nœud `Join`,

$$\begin{array}{rcl} k > 2, \mathbf{D}(k)\mathbf{J}(c_1)^k & \vdash & \mathbf{D}(k)\mathbf{J}(c_1) \\ \{0, (c_1)^k\} & \sqsupset & \{(0)^k, (c_1)\} \end{array}$$

un nœud `Split`,

$$\begin{array}{rcl} k > 2, \mathbf{D}(k)\mathbf{S}(p_1 \dots p_m)^k & \vdash & \mathbf{D}(k)^m\mathbf{S}(p_1 \dots p_m) \\ \{0, (p_1 + \dots + p_m)^k\} & \sqsupset & \{(0)^k, (p_1 + \dots + p_m)\} \end{array}$$

ou un nœud `Filter` de poids $\sigma_F = +\infty \mathrm{r} mou 1 > 0$,

$$\begin{array}{rcl} k > 2, \mathbf{D}(k)\mathbf{F}(c_1, p_1)^k & \vdash & \mathbf{D}(k)^m\mathbf{F}(c_1, p_1) \\ \{0, (\sigma_F)^k\} & \sqsupset & \{(0)^k, (\sigma_F)\} \end{array}$$

Dans les trois cas on crée $(m-1)$ nœuds $\mathbf{D}$ de poids $(0,0)$ et on enlève $k-1 \geq 1$ nœuds de poids strictement positif. La transformation est contractante.

InvertUniformize Cette transformation est composée de deux étapes, $InvertUniformize = InvertDN \circ Uniformize$, elle s'applique lorsque tous les enfants de D sont des nœuds `Split` de même poids P.

L'étape d'uniformisation remplace chaque `Split` par un `Split` de même poids mais avec éventuellement plus d'arcs et crée des `Join` pour regrouper les arcs plus petits. Par constructions les `Join` crées sont de poids strictement plus petit que chaque nœud `Split` original. On notera les poids des `Join` : $C_1, \dots, C_l < P$. $InvertDN$, comme on a montré précédemment, enlève $k-1 \geq 1$ nœuds de poids P. On a donc l'inéquation :

$$\{(0), (P)^k\} \quad \sqsupset \quad \{(0)^k, (P), (C_1), \dots, (C_l)\}$$

La transformation est contractante.

ReorderS/ReorderJ

$$\begin{array}{ccc} \mathbf{S}(\underbrace{p \dots p}_{m=d.k}) & \vdash & \mathbf{S}(\underbrace{p \dots p}_{k})^{d}\mathbf{S}(\underbrace{p \dots p}_{d}) \\ \{p.d.k\} & \sqsupset & \{(p.k)^{d}, p.d\} \end{array}$$

Comme expliqué en § 3.4.2 on n'applique pas la transformation lorsque $d = 1$ ou $k = 1$, car c'est l'identité. Donc $d > 1$ et $k > 1$, ce qui amène à $f.k.p > p.f$ et $f.k.p > p.k$ et prouve que la transformation est contractante.

SplitF

$$\begin{array}{ccc} \mathbf{F}(c_1 p_1) & \vdash & \mathbf{S}(\underbrace{c_1 \dots c_1}_{f})\mathbf{J}(\underbrace{p_1 \dots p_1}_{f})\mathbf{F}(c_1 p_1)^{f} \\ \{+\infty\} & \sqsupset & \{0, (f.c_1), (f.p_1)\} \end{array}$$

avec $f > 1$. On substitue une fonction jamais déroulée, donc de poids infini par une fonction déroulée de poids nul et des nœud `Split` et `Join`. La fonction est donc contractante.

UnrollRemove Ici on considère le cas où l'on déroule le nœud **S**. La preuve est la même lorsque l'on déroule le nœud **J** en permuttant le rôle des nœuds **J** et **S**.

$$\begin{array}{ccc} \mathbf{J}((k_1.P) \dots (k_n.P))\mathbf{S}(p_1 \dots p_m) & \vdash & \mathbf{S}(p_1 \dots p_m)^{n}\mathbf{J}(k_1.p_1 \dots k_n.p_1) \dots \mathbf{J}(k_1.p_m \dots k_n.p_m) \\ \{(P.\sum_{i=1}^{n} k_i), P\} & \sqsupset & \{(P)^{n}, (p_1.\sum_{i=1}^{n} k_i), \dots, (p_m.\sum_{i=1}^{n} k_i)\} \end{array}$$

Par construction, tous les k_i et p_j sont strictements positifs. Le poids $(P.\sum_{i=1}^{n} k_i)$ est donc strictement plus grand que tous les poids dans $\rho(R)$. La transformation est donc contractante.

UnrollBreak Ici on considère le cas où l'on déroule le nœud **J**, donc $P = k.C$. La preuve est la même lorsque l'on déroule le nœud **S** en permutant le rôle des nœuds **J** et **S**.

La première étape de $UnrollBreak$ consiste à dérouler J avec $UnrollJ$ avec un factor k,

$$\begin{array}{ccc} \mathbf{J}(c_1 \dots c_n)\mathbf{S}(p_1 \dots p_m) & \vdash & \mathbf{J}(\underbrace{C \dots C}_{k})(\mathbf{J}(c_1 \dots c_n))^{k}\mathbf{S}(\underbrace{c_1 \dots c_1}_{k}) \dots \mathbf{S}(\underbrace{c_n \dots c_n}_{k}) \\ \{C, P\} & & \{(P)^{2}, (C)^{k}, (c_1.k), \dots, (c_n.k)\} \end{array}$$

Comme attendu la transformation $UnrollJ$ n'est pas contractante. Néanmoins on se retrouve avec un couple $J - S$ de même poids P, on peut donc appliquer soit $BreakS$ ou $BreakJ$ qui comme on l'a montré remplacent les deux nœuds de poids P par des poids strictements plus petits.

L'effet combiné des deux transformations est donc contractant puisque on a substitué le nœud de poids maximal dans L et on l'a remplacé par des nœuds strictement plus petits.

FactorSJ Ici on considère le cas où l'on déroule le nœud **S**. La preuve est la même lorsque l'on déroule le nœud **J** en permuttant le rôle des nœuds **J** et **S**.

$$\begin{array}{ccc} \mathbf{S}(\underbrace{p \dots p}_{n})\mathbf{J}(\underbrace{k.p \dots k.p}_{n}) & \vdash & \mathbf{S}(\underbrace{n.p \dots n.p}_{k})\mathbf{J}(\underbrace{p \dots p}_{k}) \\ \{(p.n), (k.p.n)\} & \sqsupset & \{(k.p.n), (p.k)\} \end{array}$$

Comme discuté en § 3.6.1, la transformation $FactorSJ$ est toujours appliquée dans le sens qui diminue le nombre d'arcs, donc $n > k$. On substitue le poids $p.n$ par le poids strictement plus petit $p.k$, la transformation est contractante.

ReorderJS $ReorderJ$,$ReorderS$ et $RemoveJS$ sont contractantes donc $ReorderJS = ReorderS \circ ReorderJ \circ RemoveJS$ est également contractante.

Finitude

Nous avons montré que toutes les transformations considérées sont contractantes. Nous allons maintenant montrer que les dérivations d'un ensemble de transformations contractantes sont finies.

Lemme 3.6.2. *(Dérivations finies) Soit $\mathcal{T}$ un ensemble de transformations contractantes. Soit G_0 un graphe quelconque. Toutes les dérivations de G_0 dans $\mathcal{T}$ sont finies.*

Démonstration. Supposons l'existence d'une dérivation infinie $G_0 \xrightarrow{T_0} G_1 \ldots \xrightarrow{T_n} \ldots$. Par hypothèse, pour tout n, T_n est contractante : $\tau(G_n) \succ \tau(G_{n+1})$. On peut donc construire une suite infinie strictement décroissante : $\tau(G_0) \succ \tau(G_1) \succ \ldots \succ \tau(G_n) \ldots$. Ceci est absurde puisque l'ordre $\succ$ est bien fondé. □

Considérons maintenant l'espace des graphes engendré par les transformations de $\mathcal{T}$ à partir d'un graphe G_0. Cet espace contient toutes les dérivations possibles de G_0 dans $\mathcal{T}$. Par le lemme précédent on sait que les dérivations sont finies ; on pose β le nombre maximal de transformations dans une dérivation de G_0.

De plus, pour tout graphe G, il existe un nombre fini de transformations que l'on peut lui appliquer, puisque

- notre ensemble $\mathcal{T}$ de transformations est fini ;
- les transformations paramétrables considérent un nombre fini de paramètres ;
- le nombre de nœuds du graphe (et donc d'endroits où l'on pourrait appliquer une transformation) est fini.

Les graphes de l'espace engendré s'écrivent sous la forme :

$$G_0 \xrightarrow{T_0} G_1 \xrightarrow{T_1} \ldots \xrightarrow{T_f} G_f \text{ avec } f \leq \beta$$

or il existe un nombre fini de choix pour T_0, T_1, et T_f donc l'espace engendré est fini.

3.7 Exploration de l'espace d'implémentation

Nous venons de montrer que, sous certaines hypothèses, l'espace engendré par nos transformations est fini. Nous allons maintenant expliquer comment l'on explore cet espace.

3.7.1 Quelles transformations s'appliquent à un graphe ?

Pour pouvoir explorer l'espace engendré par les transformations il faut générer les dérivations de G_0. Pour cela il faut déterminer quelles transformations s'appliquent à G_0 ? On a vu en figure 3.2 qu'une même transformation peut s'appliquer à différents endroits. Notre problème est l'identification de tous les endroits de G_0 où la transformation s'applique. En toute généralité, il faudrait résoudre le problème de l'isomorphisme de sous-graphe : c'est-à-dire repérer tous les sous-graphes de G qui sont identiques à L. Ce problème est NP-complet[Coo71] et les algorithmes pour le résoudre sont coûteux.

En pratique avec nos transformations on peut faire plus simple. Nous appellerons nœud entrants (resp. sortants) de L ou de R, les nœuds qui sont connectés à des arcs frontière entrants (resp. sortants). On peut remarquer que pour toutes les transformations que l'on considère L possède soit un seul nœud entrant ($Deadcode$, $RemoveSJ$, etc.), soit un seul nœud sortant ($CompactJJ$). On appellera ce nœud unique, le foyer de la transformation.

Pour trouver tous les endroits du graphe qui contiennent L, on utilise l'algorithme 1. Cet algorithme essaye toutes les combinaisons entre les nœuds de G et les transformations possibles. Si un nœud N est identique au nœud foyer de la transformation T alors les nœuds voisins de N sont comparés au nœuds voisins de T, ce qui nous permet de confirmer ou d'infirmer la présence de L à cet endroit.

Algorithm 1 MATCHING$(G, \mathcal{T})$

```
1 : M ← {}
2 : for all N ∈ G_nuds do
3 :   for all T ∈ 𝒯 do
4 :     if N ≡ T.foyer then
5 :       if MatchNeighbours(N, T) then
6 :         M ← M + {T}
7 :       end if
8 :     end if
9 :   end for
10 : end for
11 : return M
```

La complexité pour comparer deux graphes lorsqu'un nœud foyer est identifié est faible, en effet la position des autres nœuds du graphe est connue. Soit n le nombre de nœuds dans G, v le nombre d'arcs de G et t le nombre de transformations. Le pire cas se produit avec $InvertDN$ qui doit vérifier que tous les fils du nœud foyer D sont identiques ; ce qui nous donne une complexité au pire cas de $\mathcal{O}(v)$. Puisque qu'il faut vérifier toutes les combinaisons de transformations et de nœuds, $C_{\text{MATCHING}} = \mathcal{O}(v.n.t)$. Mais le nombre de transformations considéré est une constante et le nombre d'arcs majore le nombre de nœuds donc la complexité finale au pire cas s'écrit

$$C_{\text{MATCHING}} = \mathcal{O}(v.n) = \mathcal{O}(v^2)$$

En pratique la complexité moyenne est bien meilleure puisque les nœuds qui n'admettent aucune transformation sont très vite éliminés grâce au test du nœud foyer.

3.7.2 Exploration exhaustive

Une première possibilité pour explorer l'espace d'implémentation est de parcourir exhaustivement toutes les variantes de l'espace en conservant la meilleure alternative ($G_{best} = argmin_G \phi(G)$). Comme nous savons que l'espace d'exploration est fini, nous pouvons utiliser un algorithme de parcours d'arbre classique. Ici nous explorerons en profondeur d'abord selon l'algorithme 2 qui se compose de cinq phases :

1. Trouver l'ensemble de transformations qui peuvent s'appliquer au graphe G.
2. S'il reste une transformations T à essayer, appliquer la transformation $G \xrightarrow{T} G'$.
3. Calculer la valeur de ϕ du graphe G' et éventuellement actualiser le meilleur candidat.
4. Faire un appel récursif pour explorer les dérivations de G'.
5. Restaurer le graphe G, avant d'appliquer une nouvelle transformation.

Algorithm 2 EXHAUSTIVE(G)

$\mathcal{T} \leftarrow \{RemoveSJ, RemoveJS, RemoveI, \ldots\}$
$G_{best} \leftarrow G$
$\phi_{best} \leftarrow \phi(G)$
for all $T \in$ MATCHING$(G, \mathcal{T})$ **do**
  $G \leftarrow T(G)$
  if $\phi(G) < \phi_{best}$ **then**
    $G_{best} = G$
    $\phi_{best} = \phi(G)$
  end if
  EXHAUSTIVE(G)
  $G \leftarrow$ RESTORE(G)
end for

Mémoire pour l'exploration Dans cette recherche nous transformons les graphes sur place. On ne fait jamais de copies de graphes. Lorsque l'on applique une transformation le graphe courant est modifié en remplaçant L par R. Bien entendu, lorsque l'on remonte une branche durant l'exploration il faut pouvoir restaurer le graphe (à son état avant transformation) pour pouvoir essayer les branches alternatives. Pour cela, lorsqu'une transformation est appliquée elle sauvegarde sur une pile les différences entre G et G' (dans un format très compact), ce qui permet à la fonction RESTORE de revenir à G. Cette manière de faire est nécessaire pour éviter une explosion de la mémoire. En effet si à chaque branche une copie du graphe était réalisée on serait limité par la mémoire à des profondeurs d'exploration très faibles.

3.7.3 Complexité de l'algorithme exhaustif

La complexité de l'algorithme est directement liée à la taille de l'espace exploré.

Le nombre de transformations qui peuvent s'appliquer est fonction du nombre de nœuds du graphe (en considérant le pire cas où chaque nœud admet au moins une transformation). À chaque étape de l'exploration, on peut donc avoir un facteur de branchement proportionnel au nombre de nœuds du graphe.

Mais la taille du graphe change au fur et à mesure de l'exploration : le nombre de nœuds peut diminuer, par exemple avec une compaction, ou augmenter, par exemple avec un déroulage. On peut faire une estimation grossière de l'augmentation maximale du nombre de nœuds après une transformation en posant $w(G) = max(n, v, \sum_{e \in G} prod(e))$. Nous pouvons vérifier pour toutes les transformations utilisées $G \xrightarrow{T} G'$ que $w(G') \leq 2.MAXARCS.w(G)$. $MAXARCS$ est la taille du déroulage maximal et sera introduit en § 3.7.5.

Soit $w_0 = w(G_0)$, on peut montrer par induction que pout tout graphe G_k obtenu après k transformations de G_0, $w(G_k) \leq (2.MAXARCS)^k.w_0 = w_k$.

Nous avons montré en § 3.6.4 que la longueur des dérivations d'un graphe initial donné est bornée. Supposons que la dérivation la plus longue pour G_0 soit de longueur β. Alors on peut majorer la taille de l'espace exploré par :

$$\prod_{k=0}^{\beta} w_k = \prod_{k=0}^{\beta} w_0.(2.MAXARCS)^k = w_0^{\beta}.\frac{1-(2.MAXARCS)^{\beta}}{1-(2.MAXARCS)}$$

Ainsi la taille de l'espace exploré dépend dans le pire cas exponentiellement de la longueur des dérivations. Il est difficile d'estimer le paramètre β pour un G_0 donné. Néanmoins on sait qu'il existe des graphes pour lesquels la taille β dépend du nombre de nœuds initial. Pour cela considérons le sous-graphe $L_{FactorJS}$ de la transformation $FactorJS$ (par exemple). Maintenant construisons le

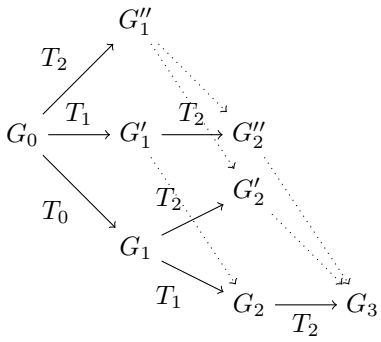


FIGURE 3.21 – Arbre d'exploration obtenu lorsque $T_0 \lessdot T_1 \lessdot T_2$ commutent deux à deux. Les branches coupés par dominance figurent en pointillées.

graphe G_0 formé en concatenant m fois bout à bout le sous-graphe $L_{FactorJS}$ sur un pipeline. Le graphe obtenu possède $2.m + 2$ nœuds (il faut rajouter un nœud `Input` et un nœud `Output`) et admet une dérivation de taille m (obtenue en appliquant m fois la transformation $FactorJS$).

On peut donc dire que dans le pire des cas, la taille de l'espace exploré dépend au moins exponentiellement du nombre de nœuds dans le graphe initial.

3.7.4 Règles de dominance

Il est possible durant l'exploration d'emprunter des branches confluentes. Par exemple sur la figure 3.2 en début de chapitre on observe que $RemoveSJ_{S1,J1} \circ RemoveSJ_{S2,J2} \equiv RemoveSJ_{S2,J2} \circ RemoveSJ_{S1,J1}$. Les branches confluentes sont dues à des transformations qui commutent : $T_1 \circ T_2 \equiv T_2 \circ T_1$. La détection des branches confluentes permet de réduire considérablement la complexité de la recherche exhaustive : puisque les branches confluent sur un même graphe, on peut en couper une des deux.

Lemme 3.7.1. *(condition suffisante de commutativité) Une condition suffisante pour qu'une transformation T_2 commute avec la transformation précédente T_1 est que les nœuds de L_2 n'aient pas été produits ou remplacés par T_1 ($R_2 \cup L_1 = \emptyset$).*

Démonstration. Soit G_2 le graphe obtenu par application de T_1 et de T_2 : $G_0 \xrightarrow{T_1} G_1 \xrightarrow{T_2} G_2$. Puisque les nœuds de L_2 n'ont pas été produits ou remplacés par T_1, ils étaient déjà présents dans G_0 ($L_2 \subseteq G_0$) et ils sont différents des nœuds de L_1 ($L_1 \cup L_2 = \emptyset$).

Puisque $L_2 \subseteq G_0$ le graphe G_0 admet la dérivation $G_0 \xrightarrow{T_2} G_1'$ avec $G_1' = (G_0 \backslash L_2) \cup R_2$. Comme $L_1 \cup L_2 = \emptyset$ et $L_1 \subseteq G_0$ alors $L_1 \subseteq G_1'$, donc le graphe G_1 admet la dérivation $G_1' \xrightarrow{T_1} G_2'$. Comme les transformations opèrent à des endroits distincts du graphe $G_2 \equiv G_2'$. □

Maintenant que l'on sait détecter certaines des transformations qui commutent on peut couper une des deux branches confluentes. Pour cela on introduit un ordre arbitraire de dominance ($\lessdot$) entre les transformations (on pourrait par exemple prendre l'ordre alphabétique entre les noms des transformations). On dira que T_1 domine T_2 lorsque $T_2 \lessdot T_1$. Cet ordre va nous permettre de choisir arbitrairement une branche parmi les différentes branches confluentes : une transformation ne sera pas appliquée si elle est commute avec une transformation antérieure qui la domine. Par exemple si on a trois transformations qui commutent on obtiendra l'arbre d'exploration en figure 3.21.

Implémentation On souhaite implémenter ces règles de dominance dans l'algorithme de recherche de manière à diminuer la complexité de l'exploration. Lors de l'exploration d'une branche par l'algorithme 2, la pile utilisée pour faire les RESTORE contient la dérivation correspondant au chemin depuis la racine (G_0) jusqu'au nœud courant. Supposons que la pile renseigne la dérivation suivante : $[T_0, T_1, \ldots, T_n]$, on souhaite savoir si T_{n+1} doit être appliquée. Pour cela il nous faut savoir si T_{n+1} commute avec une transformation antérieure qui la domine.

Pour établir la commutativité, on va utiliser la condition suffisante du lemme 3.7.1, on veut donc vérifier pour tout $i \leq n$, si $R_i \cup L_{n+1} = \emptyset$. Une possibilité consiste à garder sur la pile une trace des nœuds R_i produits par chaque transformation de manière à pouvoir vérifier la condition.

En pratique dans notre implémentation on a utilisé une condition suffisante plus faible mais beaucoup plus rapide à vérifier : chaque nœud du graphe est décoré à sa création avec une date strictement croissante sur une dérivation. Avant chaque transformation on conserve la date du nœud le plus vieux dans le graphe. On se retrouve donc avec une pile de la forme $[(d_0, T_0), (d_1, T_1), \ldots, (d_n, T_n)]$ où d_i représente la date du nœud le plus vieux dans le graphe avant application de T_i. Nous noterons d_l la date du nœud le plus vieux dans L_{n+1}. Pour savoir si T_{n+1} et T_i commutent il suffit de vérifier que $d_l < d_i$. Si $d_l < d_i$ alors tous les nœuds de L_{n+1} ont une date plus petite que d_i : ils étaient déjà présents avant l'application de T_i.

3.7.5 Simplifications

Même en utilisant les règles de dominance de la section précédente la taille de l'espace est tellement vaste que le temps d'exploration est prohibitif pour certaines instances de graphe très grandes.

Simplification systématique du graphe Une première possibilité pour réduire la taille de l'espace d'exploration consiste à considérer que les transformations simplificatrices de la § 3.3 sont toujours désirables puisqu'elles enlèvent des structures inutiles (les dérivations où elles ne sont pas appliquées sont donc souvent moins efficaces).

On modifie l'algorithme de recherche 2 de manière à considérer uniquement les transformations restructurantes du § 3.4 à chaque nouvelle branche. Les transformations simplificatrices sont alors systématiquement appliquées sur chaque nouvelle variante produite. Cela diminue la complexité d'exploration pour deux raisons :

- On considère moins de transformations dans l'arbre de recherche donc on crée moins de branches à chaque nouvelle variante ;
- On applique toutes les transformations simplificatrices à chaque nouvelle variante rencontré, or les transformations simplificatrices enlèvent des nœuds inutiles ce qui diminue la taille du graphe et réduit donc la complexité des algorithmes de matching et le nombre de transformations possibles par graphe.

Bien entendu, avec cette approche on perd des variantes puisque l'on applique les transformation simplificatrices systématiquement. Cependant les variantes perdues ne sont, dans la plus part des cas, pas les meilleures puisqu'elles contiennent des sous-graphes qui pourraient être supprimés par une transformation simplificatrice.

Nœuds avec beaucoup d'arcs Les transformations dérivées de $Unroll$, comme $UnrollBreak$, vont augmenter le nombre d'arcs de R en fonction des consommations/productions des nœuds de L. Les dérivations qui utilisent ce type de transformations vont donc produire des graphes avec beaucoup plus d'arcs et de nœuds qui vont, à leur tour, admettre plus de transformations. Ce type de transformations peuvent augmenter énormément la complexité de l'exploration qui devient fonction des productions et des consommations sur les arcs.

Pour mitiger cette explosion combinatoire, nous interdisons les transformations qui créent plus de $MAXARCS$ nouveaux arcs. Où $MAXARCS$ est un seuil réglable lors de l'exploration. Les compilateurs de langages impératifs utilisent une technique semblable. Par exemple le compilateur `gcc` (version 4.5) utilise un seuil semblable pour limiter le nombre de fois qu'une boucle peut-être déroulée.

3.7.6 Recherche par faisceau : Beamsearch

Une autre possibilité pour réduire la taille de l'espace exploré consiste à utiliser une heuristique de recherche. Nous nous sommes intéressés à l'algorithme Beamsearch utilisé pour la première fois dans [Low76].

L'algorithme Beamsearch, est une variante du Best-First search, qui considère à chaque étape le faisceau des s meilleurs candidats. C'est donc un algorithme glouton qui essaye d'optimiser localement la fonction ϕ, mais doté d'une certaine flexibilité puisque l'algorithme garde s candidats à chaque étape, ce qui lui permet de s'échapper de certains optimums locaux. L'algorithme 3 est une implémentation de la stratégie Beamsearch pour notre exploration de graphes.

Algorithm 3 BEAMSEARCH(G, s)

```
1 : T ← {RemoveSJ, RemoveJS, RemoveI, ...}
2 : G_best ← G
3 : φ_best ← φ(G)
4 : beam ← {}
5 : for all T ∈ MATCHING(G, T) do
6 :    G ← T(G)
7 :    if φ(G) < φ_best then
8 :       G_best = G
9 :       φ_best = φ(G)
10 :   end if
11 :   INSERTSORTED(beam, φ(G), T)
12 :   G ← RESTORE(G)
13 : end for
14 : for all T ∈ beam[0 : s − 1] do
15 :   G ← T(G)
16 :   BEAMSEARCH(G)
17 :   G ← RESTORE(G)
18 : end for
```

Nous verrons dans le chapitre 4, que la stratégie de Beamsearch permet d'explorer des instances de programmes de taille importante de manière efficace. On mesurera également quel est l'écart entre la solution trouvée par la recherche exhaustive et la solution trouvée par Beamsearch.

3.8 Travaux connexes

Les transformations qui préservent la sémantique des programmes sont au cœur des problématiques de la compilation.

Transformations dans StreamIt Les modèle de transformations le plus proche à nos travaux, est celui de StreamIt. StreamIt propose un ensemble de transformations sur les graphes pour adapter la granularité du programme et éliminer des Spliters et Joiners inutiles. Trois classes de transformations sont proposées :

- Les *fusion transformations* qui grossissent le grain des tâches, en combinant plusieurs filtres (verticalement ou horizontalement).
- La *fission transformations* qui sépare les filtres en plusieurs morceaux indépendants pour faciliter la parallélisation.
- Les *reordering transformations*, qui éliminent des Spliters et des Joiners inutiles pour faciliter la fission et la fusion.

Dans notre modèle nous avons conservé la transformation de fission que nous appelons $SplitF$. Comme on l'a dit en § 3.6.1, les transformations de fusion, sont réalisés après la phase d'exploration dans le backend (§ 4.1.6). Finalement nous avons conservé les *reordering transformations* présentées dans [GTK+02] : le *filter hoisting* sur les nœuds duplicate devient $InvertDN$, les transformations *synchronization removal* deviennent $RemoveJS$ et $RemoveSJ$ et on propose une variante de la *splitjoin hierarchical division* avec $ReorderS$ et $ReorderJ$. On a bien sur étendu le modèle de transformation de StreamIt qui se restraignait aux graphes SJD hiérarchiques en proposant un ensemble de nouvelles transformations. Pour explorer l'espace des transformations StreamIt utilise la technique du recuit simulé.

Transformations dans ArrayOL Un certain nombre de transformations haut-niveau ont été proposées dans le formalisme ArrayOL [Sou01][Dum05],

- La *fusion* permet d'unir deux tâches ArrayOL qui se suivent. Il faut recalculer les motifs d'entrée de manière à ce que les deux tâches aient assez d'élements pour s'exécuter. Après fusion il faut augmenter le nombre d'éléments consommés de manière à satisfaire la consommation de l'ensemble des nœuds fusionnés. On verra par la suite, (§ 4.1.6) que pour fusionner des nœuds le backend SJD est aussi obligé d'effectuer cette opération. Néanmoins dans le cadre d'ArrayOL cette opération est complexe du fait que le formalisme est multidimensionnel.
- Le *changement de pavage*, permet de changer la taille du pavage utilisée par une tâche. Elle est utilisée pour réduire le recalcul d'une tâche, c'est-à-dire des exécutions succéssives d'une tâche sur les mêmes données (cela arrive lorsque les motifs du tableau d'entrée se supperposent). Le changement de pavage élimine donc les calculs redondants, en ce sens cette transformation s'apparente à notre transformation de factorisation.
- Le *tiling* dans ArrayOL est une opération proche du tiling de boucles. Cette transformation augmente la localité des données, en changeant la granularité des entrées d'une tâche. Dans nos transformations, il n'y a pas de réel équivalent du tiling : en effet comme le modèle SJD est unidimensionnel et non-hiérarchique la localité des donnés est, en principe, implicite. On pourrait, à la limite, établir un parallèle avec les transformations $SplitJS$ et $BreakS/BreakJ$ qui cassent des jonctions $S-J$ en jonctions plus petites, en réduisant ainsi la granularité des blocs traités par les $S-J$ sur un cycle complet.

Transformations de graphes de flots dans le modèle polyédrique Dans le modèle polyédrique proposé par Feautrier[Fea92a][Fea92b], les dépendances de données sur des nids de boucles sont écrites sous la forme d'inéquations affines qui encadrent un polytope dans l'espace des itérations. En appliquant des transformations affines sur ce polytope il est possible de réordonnancer le nids de boucles, par exemple, pour augmenter le parallélisme[LL97] ou pour réduire la mémoire[QR00]. De plus, Lim et Lam ont montré[LL97] que les transformations de fusion de boucles, la fission de boucles, les réordonnancements de boucle et les transformations unimodulaires peuvent être exprimées dans le modèle polyédrique.

Certains auteurs ont eut l'idée d'appliquer le modèle de transformations polyédriques aux programmes de flots de donnés. Pour cela ils considérent que les exécutions succéssives d'un filtre

forment une itération sur l'espace des données entrantes. En plongeant les programmes de flots de données dans le modèle polyédrique, ils peuvent bénéficier de toutes les transformations qu'il contient. Ainsi dans [TLA02] les auteurs montrent comment exprimer les dépendances des graphes StreamIt dans le modèle polyédrique ce qui leur permet de benéficier de certaines optimisations comme la fission de nœuds, ou la propagation de décimation (semblable à la transformation *DeadCode*). Dans [LDWL06], les auteurs montrent comment optimiser des programmes Brook en utilisant des techniques de partitionnement affine dans le modèle polyédrique. Les auteurs transforment chaque opérateur de stream en un système d'inéquations qui le caractérise. Certains opérateurs comme *streamGroup*, introduisent des opérateurs modulo qui sont non-linéaires, néanmoins les auteurs montrent que dans certains cas il est possible de réduire le système d'inéquations à un système linéaire, ce qui leur permet d'utiliser tout l'attirail de transformations contenu dans le système polyédrique, et notamment le partitionnement affine qui permet de trouver un partitionnement des données maximisant le parallélisme de l'application. Enfin dans [MAB+10] les auteurs utilisent le modèle polyédrique pour optimiser du code SPM (Streaming Programming Model), une variante du langage C qui permet d'exprimer des flots de données. Néanmoins les optimisations sont réalisées au niveau des dépendances de boucles et non pas au niveau des dépendances de flots.

L'application du modèle polyédrique aux programmes de flots de données est une alternative très intéressante à notre ensemble de transformations de graphes. D'une part cela permet de réutiliser toutes les transformations classiques définies sur les nids de boucles ; d'autre part, le modèle polyédrique permet de trouver le meilleur ordonnancement de manière efficace lorsque la fonction économique que l'on cherche à optimiser est linéaire. Néanmoins les transformations de graphes SJD présentent un avantage : l'exploration permet de prendre en compte les fonctions économiques non linéaires. D'autre part, dans notre approche, l'optimisation se fait directement sur les graphes, ce qui permet de prendre en compte facilement leur structure dans la fonction économique.

Autres travaux Dans [PPL95], les auteurs décrivent la transformation d'élimination de code mort (*Deadcode*) dans le contexte des graphes SDF. Des transformations sur des graphes de flots de données ont été proposées pour optimiser la conception de circuits [CPRB92][VBI08] : elles travaillent à la granularité de l'opération arithmétique.

3.9 Conclusion

Le but de ce chapitre était de transformer les graphes SJD en préservant leur sémantique de manière à trouver des implémentation alternatives optimisées. Nous avons défini formellement les transformations correctes sur les CSDF et donné un critère permettant de déterminer facilement la correction d'une transformation. Cela nous a permis de construire un ensemble de transformations correctes sur les graphes SJD.

Nous nous sommes alors intéressés à l'espace engendré par ces transformations. Bien sur il est possible d'explorer l'espace engendré de nombreuses autres façon. La littérature est extrêmement riche en algorithmes et heuristiques de recherche. Nous avons ici fait le choix de construire un ensemble de transformations qui engendre un espace fini. Ceci nous permet d'explorer l'espace exhaustivement et donc de pouvoir évaluer l'heuristique proposée. Cela nous permet également d'utiliser l'algorithme Beamsearch qui ne termine que si les dérivations explorées sont finies. Cependant on aurait pu se passer de l'hypothèse de finitude de l'espace engendré et travailler avec un espace infini. Dans ce cas on aurait du forcément utiliser une méthode heuristique pour explorer l'espace avec une condition de terminaison qui garantisse la convergence. De nombreuses méthodes existent comme le recuit simulé, recherche Tabou, Beamsearch avec une profondeur maximale, etc. C'est d'ailleurs le choix qui est fait dans StreamIt, qui utilise une méthode de recuit simulé pour choisir quelles optimisations appliquer au graphe.

Dans ce chapitre nous n'avons pas défini précisément le critère à optimiser ; nous supposons que celui ci est modélisé par une fonction économique ϕ. Dans le chapitre suivant, nous étudierons deux

applications concrètes de l'exploration. D'abord, nous nous intéresserons à la réduction de l'empreinte mémoire des programmes, puis nous verrons comment réduire le coût des communications inter-processeur.

3.10 Perspectives

L'étude des transformations SJD nous a conduit à nous poser deux questions qui restent à ce jour ouvertes ; nous en discutons ci-dessous.

Classe de complexité de l'exploration Nous avons montré que la taille de l'espace engendré par les tranformations peut augmenter exponentiellement en fonction du nombre de nœuds du graphe initial. La production de tous les graphes de l'espace se fait donc au mieux en complexité exponentielle. Néanmoins, en pratique il nous suffit de trouver le graphe qui optimise ϕ. Existe-t-il des fonctions ϕ, non triviales, pour lesquelles la production du meilleur candidat peut se faire par un algorithme polynomial ? Pour-quelles fonctions ϕ, la production du meilleur candidat se réduit à un problème de décision NP-difficile ?

Caractérisation de l'espace engendré Il serait intéressant de pouvoir caractériser l'espace engendré. La méthode d'exploration exhaustive nous garantit de trouver le candidat qui optimise ϕ à *l'intérieur de l'espace engendré par nos transformations*. Néanmoins, il se peut qu'il existe un meilleur candidat qui n'est pas produit par notre ensemble de transformations. Pour une fonction ϕ donnée existe-t-il une classe de graphes pour lesquels nous sommes sûrs de retrouver le meilleur candidat ?

Chapitre 4

Backend SJD et validation expérimentale

Sommaire

Dans le chapitre précédent nous avons défini un ensemble de transformations sur les graphes SJD et montré comment explorer l'espace de programmes que l'on engendre pour optimiser une fonction économique ϕ. Dans ce chapitre nous allons étudier deux applications particulières de cette méthode :

- la réduction de l'empreinte mémoire d'un programme SJD (§ 4.3.2 et § 4.3.4)
- la réduction des communications et son impact sur le temps d'exécution (§ 4.4).

Mais avant d'étudier ces deux applications on présentera l'architecture et le compilateur avec lesquelles elles seront évaluées (§ 4.1).

4.1 Backend SJD

Le backend SJD a été implementé en Python 2.6, il compile un graphe SJD en un programme C exécutable sur un ou plusieurs cœurs.

4.1.1 Architecture cible

Nous avons essayé de nous détacher d'une architecture spécifique dans la construction du backend SJD ; de manière à le porter facilement sur différentes plateformes. Néanmoins nous avons fait quelques hypothèses minimales que nous allons présenter :

- Le nombre de cœurs de l'architecture est statiquement connu à la compilation.

- Chaque cœur est censé avoir une mémoire locale qui est utilisé pour allouer les tampons contenant les données produites par les nœuds qu'il exécute. Néanmoins plusieurs nœuds peuvent se partager la même mémoire locale.

- Pour transférer des données entre deux mémoires locales, on dispose d'un mécanisme de copie de mémoire à mémoire (DMA).

4.1.2 Vue d'ensemble du backend

En entrée du backend, dont l'organisation générale est représentée en figure 4.1, on réalise l'optimisation décrite dans le chapitre précédent, c'est-à-dire on applique un ensemble de transformations pour optimiser une fonction économique ϕ. Cette étape produit un graphe SJD qu'il faut maintenant transformer en du code exécutable sur la cible.

Lorsque l'on execute un programme SJD sur l'architecture on peut exploiter deux types de parallélisme : du parallélisme de tâches, en distribuant les nœuds du graphe parmi les différents cœurs, et du parallélisme de pipeline en superposant les exécution des nœuds dans le temps sur un pipeline logiciel. Dans notre approche nous considérons les deux types de parallélisme pour augmenter le parallélisme disponible dans les applications.

Ainsi, les deux étapes suivantes de notre Backend sont l'étape de *partitionnement* qui distribue les nœuds du programme SJD parmi les différents processeurs disponibles, et l'étape *d'ordonnancement* qui choisit un ordre d'exécution qui satisfait les dépendences de données entre les nœuds tout en créant un pipeline logiciel efficace.

Une fois que le programme SJD est partitionné et ordonnancé, on génère pour chaque cœur un programme qui va exécuter les nœuds qu'il contient. Les échanges de données entre différents cœurs se font : soit à travers une mémoire partagée ; soit lorsque chaque processeur possède une mémoire distincte à travers un transfert DMA entre les deux mémoires. Dans ce dernier cas, il faut également ordonnancer et générer le code qui déclenchera les transferts DMA. Toutes ces opérations seront réalisés dans la phase de *génération de code*. Dans notre compilateur la génération de code produit du code C qui sera transformé en langage machine par un compilateur C tiers.

Le code qui execute l'ordonnancement des tâches et des communications a un léger surcoût pour chaque execution d'un nœud et pour chaque transfert DMA déclenché. Nous avions rapidement mentionné (§ 3.6.1) l'intérêt de fusionner les tâches et les communications pour réduire ces surcoûts. La fusion des tâches a un autre intérêt : en mettant ensemble le code de différents nœuds on permet au compilateur C de faire des optimisations inter-nœuds qui se révèlent très efficaces. La fusion des communications est également nécessaire pour amortir le coût initial d'un transfert DMA en regroupant ensemble plusieurs petits transferts. Pour toutes ces raisons avant de générer le code, notre compilateur comprend des phases de *fusion des tâches* et de *fusion des communications*.

4.1.3 Partitionnement

La première étape de notre backend est le partitionnement. Il s'agit de répartir les nœuds d'un programme SJD sur les cœurs disponibles. Bien entendu, nous cherchons à obtenir le partitionnement le plus efficace possible. Plusieurs facteurs sont à prendre en compte pour optimiser la performance lors du partitionnement :

- Il faut répartir le travail des nœuds équitablement sur les différents cœurs. Car si la charge de travail est déséquilibré, les nœuds les moins chargés devrons attendre à vide les nœuds les plus chargés.

- Il faut diminuer les communications entre les cœurs. En effet, si les communications entre cœurs sont plus coûteuses que le calcul effectué par chaque cœur, la performance du système sera limitée par le coût des communications.

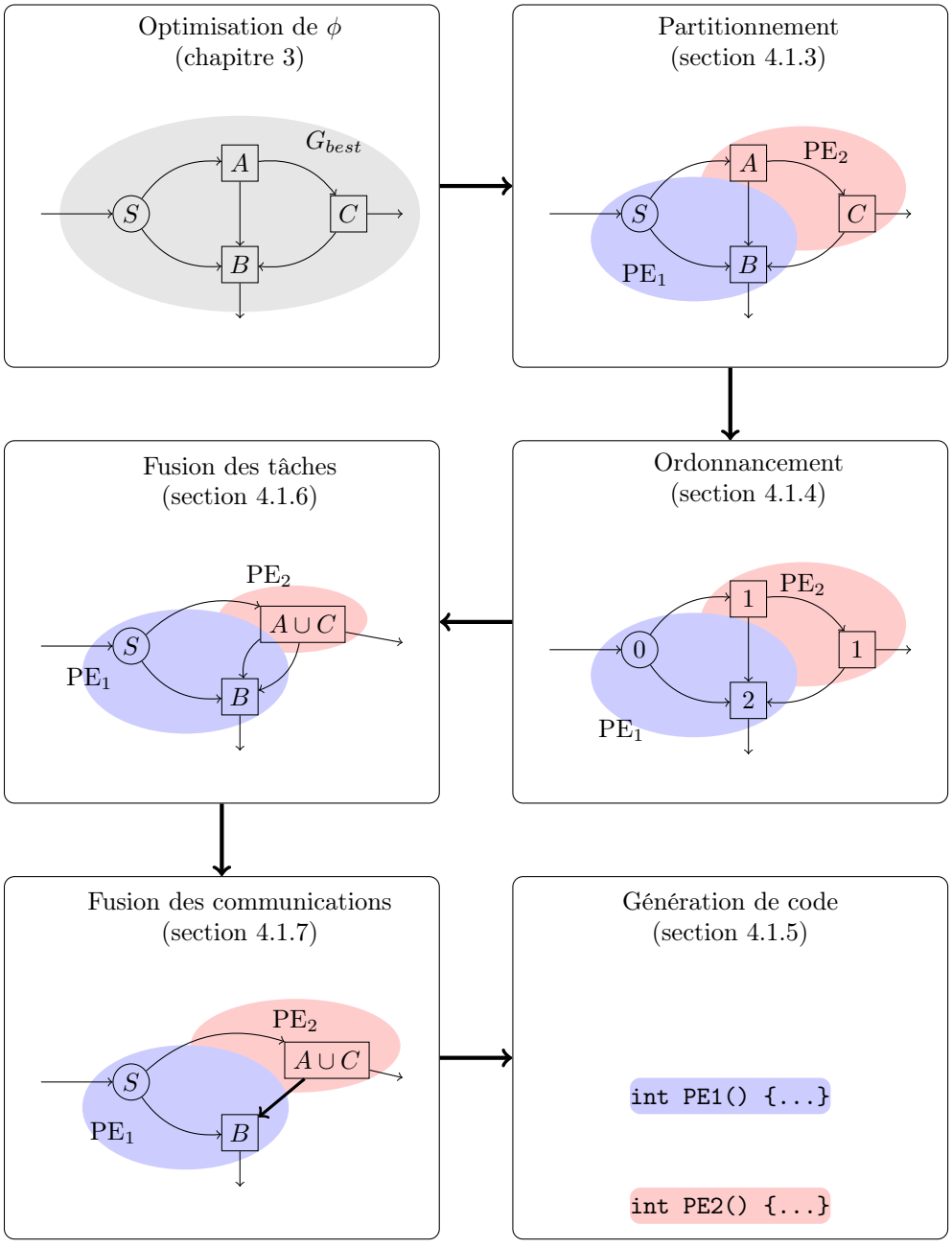


FIGURE 4.1 – Les différentes étapes du backend : de l'optimisation de ϕ du graphe SJD à la génération de code C.

- Il est intéressant de mettre sur la même partition des nœuds voisins dans le graphe. Carpenter[CRA09] arguë qu'en plaçant des nœuds voisins sur la même partition, le compilateur C peut faire des optimisation plus intéressantes. Dans nos expérimentations, on constate le même phénomène, puisque les nœuds voisins sur les mêmes cœurs peuvent être fusionnés, ce qui permet par exemple d'échanger des données plus rapidement à travers des registres.
- Il faut respecter les contraintes mémoire de l'architecture cible.

De nombreuses méthodes de partitionnement existent. Dans le contexte du partitionnement de graphes de flots de données, Udupa[UGT09b] propose une méthode de partitionnement basée sur la programmation linéaire en nombre entiers, Thies[Thi09] propose une méthode heuristique basée sur la programmation dynamique, [CRA09] propose une heuristique gloutonne itérative qui tend à placer les nœuds voisins sur le même cœur.

L'étude détaillée du partitionnement de graphes de flots de données sort du cadre de cette thèse. Nous allons donc réutiliser des méthodes de partitionnement existantes. Nous avons pourvu notre backend de deux partitionneurs. Les deux partitionneurs ont pour contrainte d'équilibrer la charge de travail des cœurs. Le partitionneur par défaut, que nous présentons ci-dessous, est optimisé pour réduire les communications inter-partition et utilise le partitionneur de graphes METIS[KK98]. Le deuxième partitionneur optimise l'empreinte mémoire du graphe ; il a été proposé par Choi[CLC+09] et il sera présenté en § 4.3.4.

Partitionnement optimisant les communications

Pour pouvoir équilibrer la charge de travail des nœuds sur les différents cœurs il faut pouvoir mesurer la charge de travail de chaque nœud. De même pour réduire les communications entre partitions, il faut connaître le coût des communications.

On cherche un partitionnement, $\mathcal{P}$, des nœuds. $\mathcal{P}$ est un ensemble de partitions ayant pour cardinal le nombre de cœurs.

Coût d'un nœud Le coût d'un nœud représente le temps d'exécution d'un nœud sur son ordonnancement stationnaire minimal. Le coût du nœud n sera noté w_n. Dans les méthodes de partitionnement considérées ici, w_n est constant. Pour mesurer w_n notre compilateur réalise un profilage de l'application. Le code de chaque nœud est compilé et exécuté de manière isolée sur la cible sur un nombre élevé d'itérations de son ordonnancement stationnaire minimal, on mesure le temps total de l'exécution et on le divise par le nombre d'itérations.

La charge de la partition correspondant au cœur $P \in \mathcal{P}$ s'écrit,

$$W_P = \sum_{n \in P} w_n$$

Coût des communications On va se placer dans le modèle de Hockney[Hoc94] qui distingue deux facteurs de coût dans une communication point à point : la latence, c'est-à-dire le coût initial pour établir la communication noté c_0 et exprimé en secondes, et la bande passante noté bw qui représente la vitesse de transfert des éléments exprimée en octets par seconde. Le coût des communications sur l'arc e reliant les nœuds n et m s'écrit sous la forme $c_{nm} = c_0 + \frac{s(e).\beta(e)}{bw}$ où $\beta(e)$ représente le nombre d'éléments échangés sur l'arc e sur une exécution de l'ordonnancement stationnaire minimal (cf. définition 2.4.13 en page 38) et $s(e)$ représente le taille en octets du type des éléments de e. On suppose également qu'un arc reliant deux nœuds assignés au même cœur a un coût de communication nul.

Pour un partitionnement donné $\mathcal{P}$, on notera $inter(\mathcal{P})$ l'ensemble des arcs dont le nœud producteur et le nœud consommateur appartiennent à des partitions distinctes. Le coût de communication du programme correspond à la somme des coûts entre tous les paires distinctes de partitions : $C(\mathcal{P}) = \sum_{e \in inter(\mathcal{P})} c_e = \sum_{e \in inter(\mathcal{P})} c_0 + \frac{s(e).\beta(e)}{bw}$.

Nous souhaitons avec ce premier partitionneur résoudre le problème d'optimisation suivant :

$$\text{Minimiser } C, \text{sous la contrainte} \forall P, Q \in \mathcal{P}, |W_P - W_Q| \leq \epsilon \tag{4.1}$$

où ϵ est un seuil constant qui représente le déséquilibre entre le travail des partitions que l'on tolère.

Pour résoudre ce problème nous utilisons le partitionneur de graphes METIS développé par Karypis[KK98]. METIS partitionne des graphes non dirigés : les nœuds et les arcs peuvent être affectés de poids. Dans notre cas nous passons le graphe SJD à partitionner à METIS, en attribuant au nœud n le poids w_n et à arête entre n et m le poids c_{nm}. On peut alors configurer METIS pour résoudre le problème (4.1).

4.1.4 Ordonnancement

Pour ordonnancer les nœuds nous utilisons la méthode *Stream Graph Modulo Scheduling* (SGMS) proposée par Kudlur[KM08]. Cette méthode propose de cacher les latences de communications en superposant les calculs et les transferts de données. Pour cela les exécutions des nœuds sont ordonnancés sur un pipeline logiciel selon un principe proche de l'ordonnancement Modulo[Rau94] souvent utilisé pour ordonnancer les instructions sur les pipelines matériels. La méthode SGMS est composé de trois phases successives :

- La fusion des cycles transforme le graphe SJD en graphe acyclique, en remplaçant chaque cycle par un nœud unique.

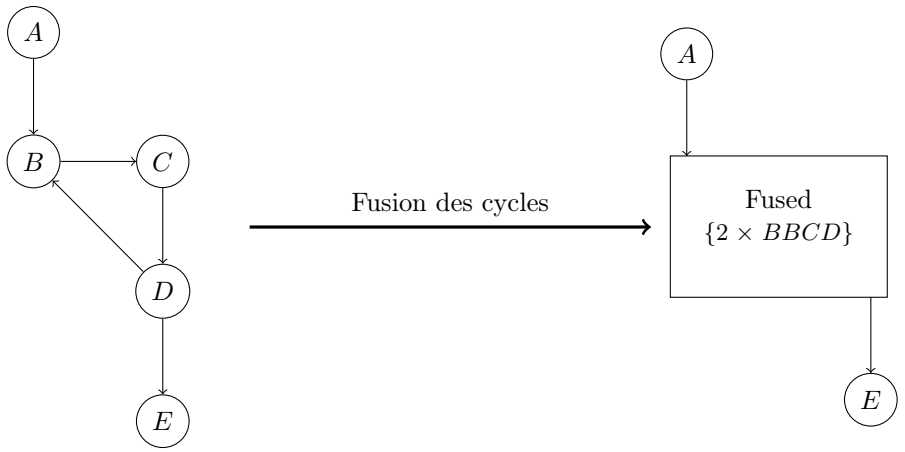
- L'uniformisation des consommations transforme le graphe de manière à obtenir un vecteur de répétition unitaire.

- L'attribution des étages ordonnance le graphe résultant.

Fusion des cycles La méthode SGMS ne permet pas de traîter directement les cycles. C'est pourquoi pour chaque cycle du graphe les nœuds composant le cycle sont fusionnés. **Dans la version actuelle de notre compilateur nous n'avons pas implémenté la fusion des cycles ; nous ne générons donc pas le code pour les programmes avec cycles.** [1]

Bilsen[BELP95] montre comment construire l'ordonnancement stationnaire minimal d'un graphe CSDF cyclique correct. Pour implémenter la fusion des cycles, on construirait donc selon sa méthode l'ordonnancement stationnaire minimal du graphe d'origine. Puis on remplacerait chaque cycle par un nœud unique qui à chaque exécution lancerait un ordonnancement stationnaire minimal des nœuds du cycle.

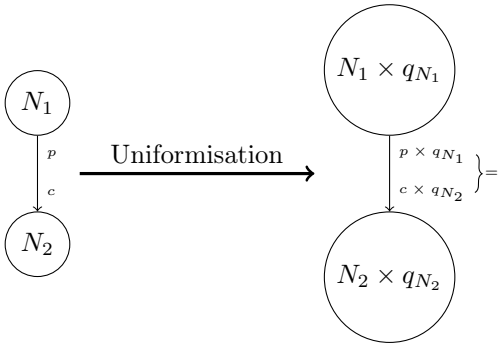
1. Néanmoins toutes les étapes précédant la génération de code prennent en compte les cycles. On peut ainsi réaliser des mesures concernant la mémoire ou les communications pour les programmes avec cycles. Mais on ne peut pas pour l'instant executer ces programmes.



Ordonnancement stationnaire :
$\{A(2 \times BBCD)E\}$

Ordonnancement stationnaire :
{A Fused E}

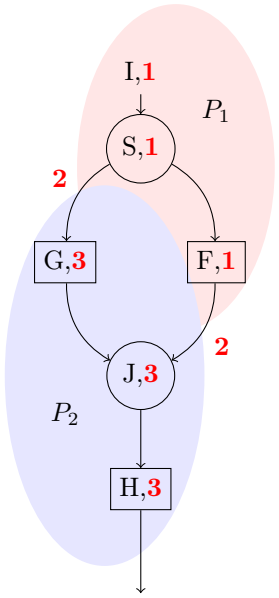
Uniformisation des consommations Avant de calculer l'ordonnancement SGMS il faut uniformiser les consommations dans le graphe. Cette étape très simple consiste à remplacer chaque nœud N par un nœud équivalent N' qui exécute q_N fois N,



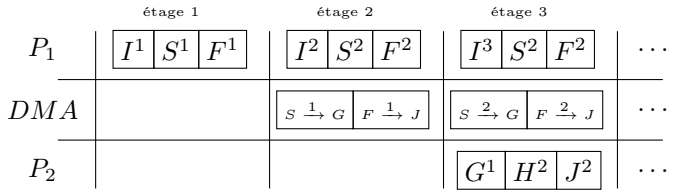
D'après la définition 2.4.10 de l'ordonnancement stationnaire (cf. page 38), $p \times q_{N_1} = c \times q_{N_2}$. Dans le graphe obtenu après uniformisation chaque nœud produit exactement ce que consomme le nœud suivant ; le graphe résultant possède un vecteur de répétition minimal de $[1, 1, \ldots]$.

Attribution des étages Une fois que l'on a fusionné les cycles et que l'on a uniformisé les consommations, chaque nœud n est affecté à un étage $stage_n \in \mathbb{N}$. Les étages sont activés graduellement, du plus petit au plus grand, en formant un pipeline logiciel. Pour attribuer les étages on applique les réglès suivantes,

- Respect des dépendances de données : si un nœud n produit des données qui sont consommées par le nœud m, alors $stage_n \leq stage_m$.
- Masquage du temps de communications : soit n, m deux nœuds appartenant à des partitions distinctes et connectés à travers l'arc e. Si la communication entre n et m nécessite un transfert DMA, alors on ordonnancera celui-ci à l'étage $stage_e$ tel que $stage_n < stage_e < stage_m$. De cette manière on s'assure que le temps de transfert de e pourra être masqué par le temps de calcul de n et de m.



(a) Un graphe SJD partitionné sur deux cœurs, les étages attribués par l'ordonnanceur SGMS figurent en rouge.



(b) Exécution de l'ordonnancement SGMS du graphe, on remarque que les transferts DMA et les calculs sont supperposés sur un pipeline logiciel.

FIGURE 4.2 – Exemple d'ordonnancement SGMS.

Pour trouver des étages qui obéissent aux règles ci-dessus il suffit de parcourir les nœuds du graphe (maintenant acyclique) dans un ordre topologique. L'étage de chaque nœud est calculé en prenant le maximum des étages des parents et en rajoutant éventuellement 1 si ceux-ci appartiennent à une partition distincte. L'algorithme détaillé ains qu'une discussion complète sur SGMS sont disponibles sur[KM08].

Nous donnons un exemple d'ordonnancement SGMS sur la figure 4.2 : en haut, on a figuré un graphe SJD partitionné sur deux cœurs, les étages SGMS en rouge ont été obtenus par l'algorithme précédent ; en bas, on a déroulé l'exécution de l'ordonnancement résultant.

4.1.5 Génération de code

Après l'ordonnancement du graphe SJD, le backend va fusionner les tâches et les communications et enfin générer le code. Nous allons déférer la présentation des deux phases de fusion (§ 4.1.6 et § 4.1.7) car nous souhaitons d'abord présenter la génération de code.

Gonflement du grain des tâches Après la phase d'uniformisation des consommations (§ 4.1.4), chaque tâche s'exécute sur un ordonnancement stationnaire unitaire. Pour cela on a lié ensemble

plusieurs exécutions de la même tâche.

Lors de l'exécution d'un programme on peut augmenter la taille de données sur laquelle les tâches travaillent par un procédé similaire. Il suffit de remplacer chaque appel d'une tâche par *coarse* exécutions consécutives de la tâche. On multiplie ainsi la taille des données consommées et le travail d'une tâche par *coarse*.

Dans [STRA05] et [UGT09b] les auteurs font remarquer l'importance de trouver un facteur de *coarse* adéquat. En effet, le facteur *coarse* influe sur les performances d'un programme pour plusieurs raisons :

- Lorsqu'un cache est présent, la valeur de *coarse* change la taille des données traitées. Si une valeur de *coarse* trop importante est choisie les entrées d'une tache ne tiennent plus dans le cache, ce qui provoque des fautes de cache.

- La valeur de *coarse* change la taille des données traitées, en changeant la valeur de *coarse* on change donc le rapport entre le travail d'un tâche et le coût des communications. Puisque le coût des communications et le travail d'une tâche sont superposés. Il faut donc choisir la valeur de *coarse* pour laquelle le travail d'une tâche et le coût des communications sont les plus proches.

- Enfin le temps d'un transfert DMA compte toujours un coût fixe correspondant à la latence du bus c_0, en augmentant le facteur *coarse* on augmente la charge utile de chaque paquet envoyé.

Dans la version actuelle de notre compilateur, le concepteur fournit à la chaîne la valeur souhaité de *coarse*. Pour trouver la valeur idéale de *coarse*, on pourrait mettre en place une phase de profilage, où l'application est exécutée à plusieurs reprises en faisant varier la taille de *coarse*. Ce processus permettrait de déterminer automatiquement la valeur la plus performante.

Une fois la valeur de *coarse* déterminée, on gonfle la tâche. Si le code du nœud F est représenté par la fonction f. On remplacera f par :

```
for (int j=0; j<coarse; j++) {f;}
```

Génération de code pour l'ordonnancement Chaque partition sera compilée vers un thread dont le code est rassemblé en un seul fichier C. Chaque thread possède une boucle principale qui est chargée d'ordonnancer les executions des nœuds selon les étages assignés en § 4.1.4. Nous allons expliquer l'organisation générale d'une tâche en reprenant la partition P_1 de l'exemple de la figure 4.2(a).

L'ordonnancement de la partition P_1 est simple : à l'étage 1 il faut exécuter I, S et F et à l'étage 2 il faut exécuter DMA_{SJ} et DMA_{FJ}. Le code résultant est transcrit dans l'extrait 4.1. La ligne 2, initalise deux variables `start` et `end`, qui décrivent l'intervalle des étages du pipeline actifs. Ainsi à l'initialisation seul l'étage 1 est actif. La ligne 4 déclare une boucle qui se charge d'exécuter l'ordonnancement modulo sur le nombre d'itérations spécifiées par l'utilisateur (en fait divisé par la valeur de *coarse*). `max_stage` représente le nombre d'étages dans l'ordonnancement, ici 3. On exécute la boucle `max_stage + iterations` fois : en effet la dernière itération doit traverser tout le pipeline, il faut donc ajouter la longueur de celui-ci. Le bloc qui commence à la ligne 6 vérifie si l'étage 1 doit être exécuté, pour cela il teste si 1 est compris entre `start` et `end`. Le cas écheant les nœuds I,S et F sont exécutés. De même pour l'étage 2, pour lequel les transferts DMA sont déclenchés. En ligne 18 on élargit le nombre d'étages actifs jusqu'à ce que tous les étages soient activés. Lorsqu'on atteint la dernière itération en ligne 21, on commence à vider le pipeline en sens inverse. Enfin en ligne 24 on fait une synchronisation avec toutes les autres partitions avant de commencer le tick suivant de l'ordonnancement.

```
1  void *work_P_1(void * arg) {
2    int start = 1; int end = 2;
3
4    for (int i=0; i < max_stage + iterations; i++) {
5
6      if (start <= 1 && end > 1) {
7        /* Etage 1 */
8        fire_I(i - 1);
9        fire_S(i - 1);
10       fire_F(i - 1);
11     }
12     if (start <= 2 && end > 2) {
13       /* Etage 2 */
14       DMA_SJ(i - 2);
15       DMA_FJ(i - 2);
16     }
17
18     if (i < max_stage)
19       end += 1;          /* Remplissage du pipeline */
20
21     if (i >= iterations)
22       start += 1;        /* Vidage du pipeline */
23
24     barrier();           /* On attend que toutes les partitions aient */
25   }                      /* fini le pas courant.                      */
26 }
```

Extrait de code 4.1 – Code généré pour la partition P_1.

Compilation de la mémoire Soit un arc e sur lequel le nœud n produit des données et le nœud m les consomme. À chaque execution de n, celui-ci consomme sur e un nombre de données égal à $\beta(e) = cons(e).q_n.coarse$. Comme l'ordonnancement SGMS forme un pipeline logiciel, plusieurs exécutions de n et de m peuvent être exécutées simultanéement. Pour maintenir la cohérence de données entre ces différentes exécutions on utilise un tampon circulaire. Choi[CLC+09] montre qu'il suffit de prévoir $(stage_m - stage_n + 1)$ cases dans le tampon circulaire pour garantir la cohérence des données. À l'exécution i du nœud n celui ci consommera les données de la case numéro $[i \ mod.(stage_m - stage_n + 1)]$. Ainsi pour chaque arc e il faut prévoir un tampon de taille $b(e) = (stage_m - stage_n + 1).\beta(e).coarse$.

Lors de la génération du code, le backend génère pour chaque arc e des macros `__push__e` et `__pop__e` qui permettent de consommer et de produire des éléments sur e sans avoir à faire des calculs d'index fastidieux pour savoir à quel endroit du tampon circulaire on se situe.

Génération du corps des nœuds Le code des nœuds de routage n'est pas fournit par l'utilisateur. Le backend produit les nœuds de routage automatiquement.

Pour un nœud `Join` $J(p_1 \dots p_m)$ la fonction générée est :

```
__push__(__pop__1()); } ×p_1
__push__(__pop__1()); }
...
__push__(__pop__m()); } ×p_m
__push__(__pop__m()); }
```

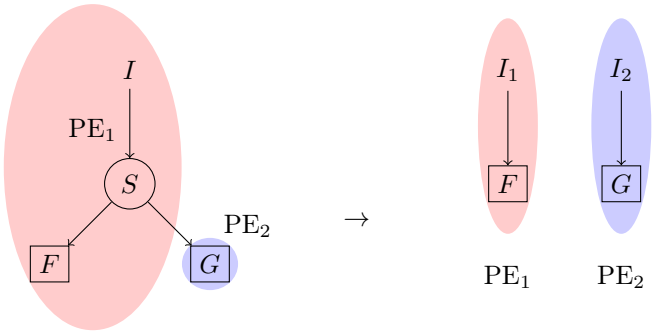


FIGURE 4.3 – Optimisation pour la génération du code des sources . Lorsque les données de la source sont immédiatement découpées par un nœud `Split`, on élimine le goulet d'étranglement.

Pour un nœud `Split` $S(c_1 \dots c_n)$ la fonction générée est :

```
__push__1(__pop__());  } ×c_1
__push__1(__pop__());  }
...
__push__n(__pop__());  } ×c_n
__push__n(__pop__());  }
```

Les nœuds `Duplicate` $D(m)$ sont traités spécialement, au lieu de copier les données sur m tampons différents. Les consommateurs vont tous lire sur le même tampon de manière à éviter les copies inutiles et réduire l'espace mémoire utilisé. Bien sur, une copie distante sera réalisée pour les consommateurs sur d'autres cœurs qui nécessitent un transfert DMA.

La génération du code des nœuds `Filter` est relativement simple puisque leur code nous est fourni par le programmeur. On doit tout de même insérer aux endroits adéquats les appels à `__push__` et `__pop__` pour réaliser l'interface entre le code utilisateur et les données des arcs entrants et sortants.

Une propriété importante dans le code produit par la phase de génération du corps des `Filter`, est que les `__pop__` et les `__push_` générés ne sont jamais imbriqués dans une structure de boucle (for, while). Par ailleurs notre compilateur rejette les corps de fonctions contenant des instructions (goto). Cette propriété est importante lors de la fusion des tâches que l'on présente ci-dessous.

Génération du code des sources Les sources correspondent aux entrées d'un programme. Dans le contexte du traîtement du signal embarqué, les entrées peuvent provenir d'un capteur : par exemple, le flux vidéo capté par une caméra embarquée. Pour simplifier, on suppose ici que les données des sources sont lisibles sur une plage d'adresses mémoires. Le code des sources réalise la copie des données de ces plages mémoire sur son tampon sortant.

On considère cependant une optimisation particulière lorsque les données de la source sont immédiatement découpées par un nœud `Split` et redistribuées à plusieurs partitions, comme sur la figure 4.3. Les nœud `Split` et `Input` sont placés sur un des cœurs par le partitionneur (supposons qu'il s'agisse du cœur PE_1 pour l'exemple). Le problème est que le code généré pour `Split` lit tout les éléments : ceux dont le cœur PE_1 à besoin mais aussi tous les éléments qu'il doit envoyer aux autres cœurs. PE_1 devient dans ce cas de figure un goulet d'étranglement par lequel transitent toutes les entrées de l'application. Ceci est inefficace : chaque cœur devrait lire dans la mémoire uniquement les entrées qui lui sont propres. Pour rémédier à cette situation, nous traitons ce cas de figure spécialement en générant une source factice pour chaque partition qui ne lit que les éléments dont la partition à besoin.

4.1.6 Fusion des tâches

La génération de code de la section précédente se révèle inefficace sur des programmes à grain très fin et cela pour deux raisons :

- les fonctions de travail des nœuds étant très courtes mais très nombreuses, en proportion, beaucoup de temps est passé dans l'ordonnanceur à appeler les fonctions.
- entre chaque paire de nœuds, les `__push__` et `__pop__` respectifs doivent lire et écrire les éléments sur des tampons, même si les écritures se font sur le cache c'est tout de même plus coûteux que de passer par des registres.

La fusion des tâches consiste à remplacer plusieurs nœuds dans le graphe SJD par un seul nœud. Deux nœuds peuvent être fusionnés à condition d'appartenir à la même partition et d'être ordonnancés sur le même étage.

Fusion de deux nœuds Supposons que l'on fusionne deux nœuds n et m.La fusion de nœuds se fait en plusieurs étapes représentées sur la figure 4.4.

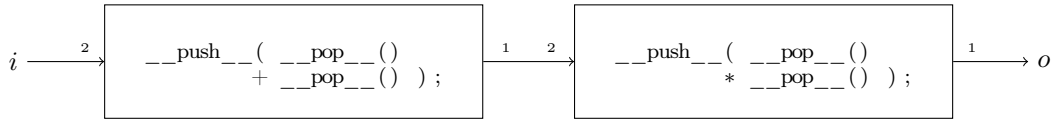
- Nous rappelons que la phase d'uniformisation de l'ordonnanceur SGMS a remplacé le corps du nœud n par le corps déroulé q_n fois (§ 4.1.4). On a donc un graphe dont les consommations et productions sont égales entre n et m comme sur la figure 4.4(b).
- Puisque il y a autant de consommations que de productions, le nombre de `__push__` de n est exactement le nombre de `__pop__` de m. On remplace donc chaque paire push, pop par une même variable locale, comme sur la figure 4.4(c).
- Enfin on fusionne les deux nœuds en concaténant leurs corps, comme sur la figure 4.4(d).

Le code résultant après fusion est bien plus efficace, en effet les communications entre n et m sont exprimées à travers des variables locales. Si le nombre de variables est faible, le compilateur C pourra assigner ces variables sur des registres et éviter les copies sur les tampons de m et de n.

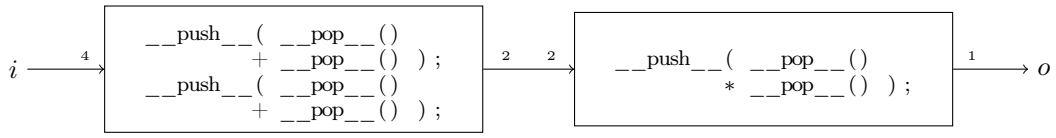
Fusion des nœuds du graphe On réalise la fusion d'un ensemble de nœuds de proche en proche, il est important de faire la fusion dans un ordre particulier pour ne pas introduire des cycles dans le graphe. En effet la fusion simple que l'on a présentée dans le paragraphe précédent n'est correcte qu'entre deux nœuds sans cycle. Si des cycles sont présents on est obligés de dérouler l'exécution du cycle en entrelaçant les executions de n et de m ce qui complique inutilement le procédé.

L'algorithme pour réaliser la fusion de proche en proche sans introduire de cycle est très simple :

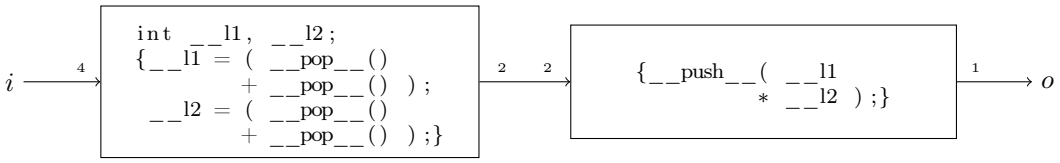
1. Nous rappelons que le graphe obtenu après l'ordonnancement SGMS est acyclique. En faisant un parcours topologique du graphe, on peut donc décorer chaque nœud n du graphe avec un numéro $d(n)$ de manière à que l'ordre des fils de n soit strictement supérieur à l'ordre de n. Un tel ordre de dominance vérifie la propriété suivante $d(n) \leq d(m)$ *ssi* il n'existe pas de chemin entre m et n.
2. On considère tour à tour, chaque ensemble de nœuds à fusionner, c'est-à-dire des ensembles de nœuds connexes, appartenant à la même partition et au même étage. On classe l'ensemble de ces nœuds par ordre de dominance croissant.
3. On sélectionne les deux nœuds n et m de dominance minimale que l'on fusionne. Le nœud crée f est remis dans l'ensemble à fusionner avec une dominance, $d(f) = max(d(n), d(m))$. On répète cette opération jusqu'à ce qu'il ne reste plus qu'un seul nœud.
4. On considère l'ensemble suivant à fusionner.



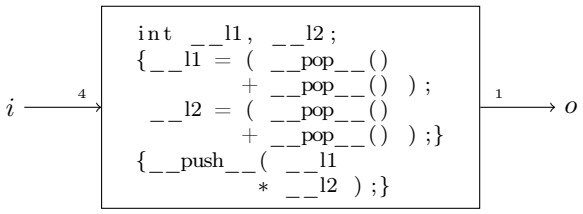
(a) Graphe original.



(b) Après uniformisation des communications.

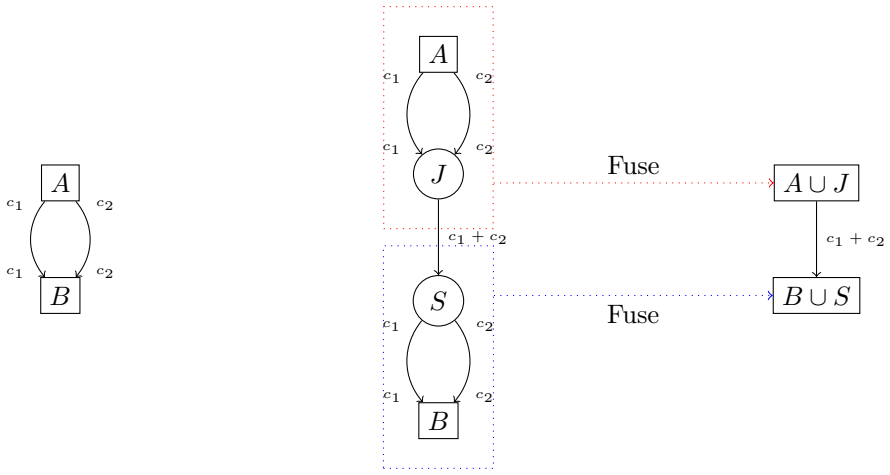


(c) Les push et les pop internes sont substitués par des variables locales.



(d) Fusion des nœuds.

FIGURE 4.4 – Exemple de la fusion entre deux nœuds.



(a) Graphe original

(b) On rajoute des nœuds J et S factices.

(c) On fusionne les deux ensembles de nœuds.

FIGURE 4.5 – Fusion des communications.

Il est facile de montrer que cet algorithme n'introduit jamais de cycles lors de la fusion de n et de m.

Démonstration. Par hypothèse on sait que $\forall x \in G, d(n) \leq d(m) \leq d(x)$. Supposons qu'après l'étape (3) de fusion il existe un cycle dans le graphe. Puisque le graphe était par hypothèse acyclique, le cycle a été introduit lors de la fusion de n et m. Cela veut dire qu'il existait dans G un chemin entre m et n ou un chemin entre n et m autre que l'arc ou les arcs $n \to m$.

Nous allons montrer que les deux cas de figure sont absurdes.

- *Chemin entre m et n :* on sait que $d(n) \leq d(m)$, donc il n'existe pas de chemin entre m et n.
- *Chemin entre n et m autre que $n \to m$:* supposons qu'un tel chemin existe, alors il existe des nœuds $p_1, \ldots, p_l$ distincts de n et m, tels que $n \to p_1 \to \ldots \to p_l \to m$, donc $d(p_l) < d(m)$, ce qui est absurde car m et n sont les minimums pour l'ordre de dominance dans G.

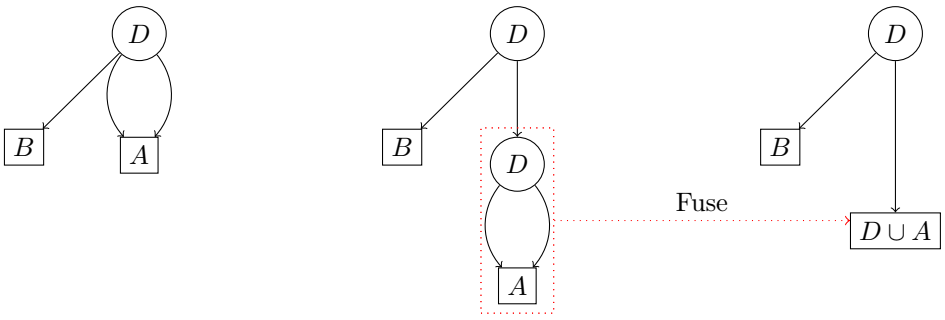
□

Taille du code généré La phase de fusion des tâches peut faire grandir beaucoup la taille du code. Pour certains programmes comme la multiplication matricielle, on génère des codes très longs qui ralentissent la compilation. Dans [STRA05] des méthodes de fusion de code qui prennent en compte l'augmentation de la taille du code sont présentées.

4.1.7 Fusion des communications

Lorsque pour deux nœuds A et B plusieurs arcs $A \to B$ existent, il est possible de les fusionner pour augmenter la charge utile des transferts DMA entre A et B.

Nous allons montrer comment la fusion des communications est implémentée en prennant l'exemple du graphe de la figure 4.5. La première étape consiste à rajouter un nœud `Join` factice qui va regrouper les productions de A et un nœud `Split` factice qui va regrouper les consommations de B. Puis on va réutiliser l'algorithme de fusion de nœuds décrit dans la section précédente pour fusionner ensemble A avec J, et B avec S. Le graphe résultant en figure 4.5(c), ne possède par



(a) Graphe original.

(b) Ajout d'un nœud `Duplicate` factice.

(c) Fusion des communications.

FIGURE 4.6 – Fusion des communications pour les nœuds `Duplicate`. L'avantage d'utiliser cette méthode pour les nœuds `Duplicate`, c'est que la copie des données est faite après le transfert de données entre D et A, ce qui réduit le volume des communications.

construction qu'un seul arc de communications entre les deux nœuds. Bien entendu, cette méthode est toujours valable avec un nombre arbitraire d'arcs entre A et B.

Fusion des nœuds Duplicate La méthode de fusion considère spécialement les nœuds `Duplicate` car il est possible de réaliser une optimisation. Supposons que l'on soit dans le cas du graphe de la figure 4.6(a). On souhaite fusionner les communications émanant d'un nœud `Duplicate`. Si on utilisait la méthode précédente on copierait deux fois le même flux entre D et A ce qui est innéficace. On propose donc une nouvelle méthode de fusion qui ne copie qu'une fois les données et réalise les copies localement sur $D \cup A$. D'une part cela réduit le volume de communications entre les deux nœuds. D'autre part puisque la copie des données se fait localement sur $D \cup A$, elle ne coûte rien après l'optimisation locale des nœuds `Duplicate` que l'on a présenté en § 4.1.5.

4.2 Validation expérimentale

Maintenant que nous avons présenté le backend SJD que nous avons implémenté, nous allons présenter dans la suite deux applications de la méthode d'exploration des transformations de graphes du chapitre 3. Nous allons étudier différents problèmes qui feront intervenir des fonctions économiques ϕ différentes.

Tout d'abord nous allons utiliser les transformations pour **réduire la mémoire nécessaire** pour exécuter un programme SJD. Dans le cadre des systèmes embarqués dans lequel on se place, c'est un problème important car les cibles sont souvent très contraintes en mémoire. Bien entendu l'empreinte mémoire d'un programme SJD dépend du modèle d'exécution choisi. Nous allons mesurer la réduction de mémoire dans deux cas de figure :

- Nous montrerons comment réduire la mémoire dans le cas d'une exécution sur un seul cœur (§ 4.3.2).
- Nous montrerons comment réduire la mémoire dans le cas d'une exécution parallèle sur plusieurs cœurs (§ 4.3.4).

Puis nous allons utiliser les transformations pour **réduire les communications** entre les différentes partitions d'un programme SJD. Au risque de nous répéter, nous rappellons que la

performance d'un programme parallèle dépend de nombreux facteurs ; en particulier pour les programmes avec des volumes importants de communications le temps d'exécution est dominé par le coût des transferts. Réduire les communications entre partitions est donc un problème important pour améliorer la performance des programmes. Nous montrerons en § 4.4 que les transformations proposées permettent de réduire les communications sur plusieurs programmes. Nous montrerons, l'impact de cette réduction sur les performances d'un certain nombre de ces programmes.

Avant de présenter ces résultats, il nous faut introduire l'ensemble des *benchmarks* qui serons utilisés pour conduire les expérimentations.

4.2.1 Benchmarks

Dans cette section nous présentons l'ensemble des programmes sur lesquels on va mesurer expérimentalement la réduction mémoire sur monocœur et multicœur. Parmis ces programmes on retrouvera les exemples du chapitre 2 mais aussi des programmes tirés de la suite StreamIt [Gro]. En effet notre méthode s'applique aussi bien aux graphes SLICES et aux graphes SJD.

Programmes produits par SLICES On va considérer un ensemble de programmes dont les réorganisations de données ont été produites par SLICES :

- **MM-COARSE**, c'est la multiplication matricielle produite par SLICES avec des matrices 12×12 (cf. figure 2.9 en page 29), le programme est comparable au code `matrixmult.str` distribué avec StreamIt.
- **MM-FINE**, même programme que MM-COARSE où le nœud filtre chargé de réaliser le produit scalaire ne travaille plus à l'échelle du vecteur, mais à l'échelle du scalaire. L'accumulation des sommes est réalisée grâce à un cycle dans le graphe.
- **GAUSS**, filtre de gauss dont le code a été produit par SLICES (même graphe que sur la figure 2.27 en page 61, en ajoutant le filtre en sortie du graphe), on considère deux versions GAUSS-10x10 qui travaille sur des images de 100 pixels et GAUSS-100x100 qui travaille sur des images de 1000 pixels. Nous ne prenons pas le filtre de Sobel car il est très proche de celui de Gauss.
- **HOUGH**, filtre de hough dont le code a été produit par SLICES (cf. figure 2.12 en page 35). Comme pour GAUSS on considère des versions sur une image 10×10 et une image 100×100.

Programmes SJD tirés de la suite StreamIt Les programmes qui suivent ont été tirés de la suite StreamIt et retranscrits en SJD,

- **FFT**, la transformée de Fourier rapide sur des vecteurs de taille 16, représentée en figure 2.10. Correspond au code `fft5.str` distribué avec StreamIt.
- **DES**, l'algorithme de chiffrage DES. Le code `des.str` distribué avec StreamIt travaille sur des vecteurs de taille 16. Nous considérons également une version avec des vecteurs de taille 8.
- **BITONIC**, tri bitonic. Le code `bitonic.str` distribué avec StreamIt travaille sur des vecteurs de taille 32. Nous considérons également une version avec des vecteurs de taille 8.
- **DCT**, transformée en cosinus discrète correspond au code `des.str` de StreamIt.
- **FM**, démodulateur FM correspondant au code `FMRadio.str` de StreamIt.
- **CHANNEL**, vocodeur correspondant au code `ChannelVocoder7.str` de StreamIt.

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

TABLE 4.1 – Charactéristiques des benchmarks : nombre de nœuds, nombre d'arcs, nombre de nœuds de routage, nombre de filtres, nombre d'entrées et de sorties,nombre de cycles et types des données transitant sur les arcs.

Taille des programmes Dans la table 4.1 nous donnons le nombre de nœuds filtres, de nœuds de routage, d'arcs et de cycles et le type des données manipulées pour chacun de ces programmes.

4.2.2 Système hôte pour la compilation

Le backend présenté en début de chapitre qui réalise entre autres l'exploration de transformations décrite dans le chapitre 3 a été implémenté en Python. Pour compiler les benchmarks, nous avons exécuté l'exploration de transformations sur l'interpréteur `python-2.6` sur un ordinateur doté de $5GB$ de mémoire vive et d'un processeur Intel Pentium 4 cadencé à $3.8GHz$. Les temps d'exploration sont mesurés avec l'appel système `gettimeofday` sur un noyau linux `linux-2.6.32`.

4.3 Réduction de la mémoire

Nous appelerons empreinte mémoire la quantité de mémoire nécessaire pour implémenter les tampons de communications entre nœuds. Nous ne considérons pas la mémoire necessaire au stockage des instructions et à l'exécution du corps des filtres.

Nous allons d'abord nous intéresser à la réduction de la mémoire en monoprocesseur (§ 4.3.1 - § 4.3.3), puis sur multiprocesseur (§ 4.3.4 - § 4.3.6).

4.3.1 Encadrement des besoins mémoire en monoprocesseur

La taille mémoire d'un graphe G s'écrit $mem(G) = \sum_{e \in G} mem(e)$ où $mem(e)$ est la taille du tampon nécessaire au canal e. Nous remarquons que $mem(e)$ dépend de l'ordonnancement choisi. Supposons deux nœuds $A \xrightarrow{e} B$ et $prod(A) = cons(B) = 1$. Si on execute ce SDF avec l'ordonnancement stationnaire $AABB$, la nombre maximum d'éléments accumulés sur le tampon e est 2. Si par contre on execute $ABAB$, la taille maximale atteinte par le tampon est 1. Nous souhaitons évaluer la réduction mémoire par exploration indépendamment de l'ordonnancement, nous allons donc travailler sur un encadrement de $mem(e)$.

Soit $minmem(e)$ la taille minimale necessaire de la file e sur tous les ordonnancement stationnaires possibles. Nous allons encadrer cette borne. Nous savons qu'il existe un ordonnancement stationnaire minimal dont le vecteur de répétition est noté $\vec{q}$ sur lequel les tampons mémoire pour e nécessite $prod(A) \times q_A = \beta(e)$ éléments (cf. définition 2.4.13 en page 38). Dans [BML97] une borne minimale de $minmem$ est calculée, cette borne est appelée *Buffer memory lower bound*,

$$bmlb(e) = \frac{prod(A).cons(B)}{pgcd(prod(A), cons(B))}$$

On peut ainsi encadrer la mémoire minimale nécessaire pour implémenter les communications sur un arc e par

$$bmlb(e) \leq minmem(e) \leq \beta(e)$$

et par conséquent,

$$\sum_{e \in G} bmlb(e) \leq minmem(G) \leq \sum_{e \in G} \beta(e)$$

Si le graphe est consistant et vivace, il est toujours possible de construire un ordonnancement[BELP95] dont la consommation mémoire est $beta(G)$. La borne supérieure est donc atteinte. Par contre, la borne inférieure $bmlb$ n'est pas toujours atteignable. Dans [BML97] les auteurs montrent qu'il existe des graphes pour lequels le $bmlb$ ne peut être atteint.

4.3.2 Choix de ϕ et exploration en monoprocesseur

Cet encadrement nous permet d'estimer la mémoire minimale théorique sur un seul processeur. Nous souhaitons utiliser la méthode d'exploration du chapitre 3 pour réduire ces bornes. Pour cela il nous faut définir une fonction économique ϕ. Pour réduire la mémoire théorique minimale $minmem$ nous allons essayer réduire sa borne supérieure, $\sum_{e \in G} \beta(e)$, nous posons donc

$$\phi(G) = \sum_{e \in G} \beta(e).$$

Rappelons que $\beta(e)$ représente le nombre d'éléments échangés sur e pendant une execution de l'ordonnancement stationnaire minimal. La fonction économique guide donc l'exploration de manière à réduire le maximum de β sur le graphe.

Nous explorons les benchmarks selon les deux méthodes d'exploration proposées :

- la méthode exhaustive (§ 3.7.2) avec les simplifications décrites en § 3.7.5 ;
- la recherche par faisceau (§ 3.7.6) avec des tailles de faisceau allant de 1 à 5.

Chaque exploration (exhaustive ou par faisceau) dispose au maximum de 60 minutes de temps d'exécution sur un ordinateur de bureau. Si une exploration n'a pas terminé au bout de 60 minutes, elle est interrompue.

Exploration exhaustive L'exploration exhaustive engendre toutes les dérivations possibles du programme initial et choisit celle qui minimise $\phi(G) = \beta(G)$. Nous mesurons pour ce meilleur candidat les valeurs β et $bmlb$. L'exploration exhaustive parcours tout l'espace des variantes, elle trouve donc le meilleur candidat possible par nos transformations

Exploration par faisceau L'heuristique de recherche par faisceau explore une fraction des candidats engendrés. Pour une taille de faisceau nulle, aucune transformation n'est appliquée, l'espace ne contient que le programme original. Pour une taille de faisceau 1, l'algorithme est équivalent à l'heuristique gloutonne Best First Search, une seule dérivation est considérée. Pour des tailles supérieures, l'espace considéré grandit au fur et à mesure que l'on augmente la taille du faisceau. Chaque exploration par faisceau nous retourne le meilleur candidat dans l'espace exploré, pour lequel nous mesuront β et *bmlb*. Bien entendu, le candidat trouvé par la recherche par faisceau ne peut jamais être meilleur que celui trouvé par la recherche exhaustive.

4.3.3 Résultats monoprocesseur

Dans cette section nous allons évaluer la réduction mémoire obtenue par l'exploration exhaustive et par l'exploration par faisceau. Nous observons que dans la plus part des cas avec une largeur de faisceau de 5, l'exploration par faisceau trouve le meilleur candidat, avec des temps d'exploration faibles.

Programmes qui n'admettent pas de transformations

Nous nous intéressont tout d'abord aux programmes CHANNEL, FM et DCT. Les résultats de l'exploration pour ces programmes sont représentés en figure 4.7. Les lignes en pointillés représentent les bornes *bmlb* et *beta* obtenues pour le meilleur candidat lors de la recherche exhaustive. En abcsisse on a figuré la taille du faisceau (0 correspond au programme original). Pour chaque taille de faisceau on trace une barre bleu qui indique l'intervalle entre les bornes *bmbl* et β. Enfin, la courbe grise correspond au temps d'exploration en fonction de la taille du faisceau.

En observant les courbes obtenues on s'apperçoit que l'écart entre la borne supérieure de la mémoire pour le programme original et pour le meilleur candidat par recherche exhaustive est négligeable : ces programmes n'affichent pratiquement aucune réduction mémoire. Cela est dû au fait que ces programmes utilisent peu de nœuds de routage et de manière très simple : par exemple le programme DCT est formé par un pipeline de filtres, les quatre nœuds de routage utilisés correspondent a des split et joins isolés résultant de la parallélisation de deux des étages.

Nos transformations agissent principalement sur les ensembles de nœuds de routage et optimisent les réorganisation de données. Comme les programmes CHANNEL, FM et DCT n'utilisent pas les nœuds de routage pour faire de la réorganisation de données, il y a très peu de transformations qui peuvent leur être appliqués. La taille de l'espace engendré est réduite et les candidats produits sont proches du programme original. Ceci est confirmé par la mesure du temps d'exploration qui est proche de zéro pour toutes les largeurs de faisceau.

Programmes produits par SLICES

Nous avons représenté en figure 4.8, les mesures pour les programmes produits par SLICES.

Multiplication matricielle Le programme MM-FINE original affiche une borne supérieure plus élevée que le programme MM-COARSE alors qu'elles ont la même borne inférieure. L'accumulation de la somme lors du calcul du produit scalaire est explicite dans MM-FINE. Ici, la borne supérieure correspond à l'ordonnancement qui garde toutes les multiplications en mémoire avant de faire leur somme ; la borne supérieure correspond à l'ordonnancement qui fait la somme au fur et à mesure.

L'exploration en faisceau est rapide et trouve le meilleur candidat dès que le faisceau atteint la taille 2. Les réductions sont importantes pour la borne supérieure β : -23.1% pour MM-COARSE et -26.9% pour MM-FINE. Elles le sont un peut moins pour la borne inférieure *bmlb* : -2.0% pour MM-COARSE et -4.9% pour MM-FINE.

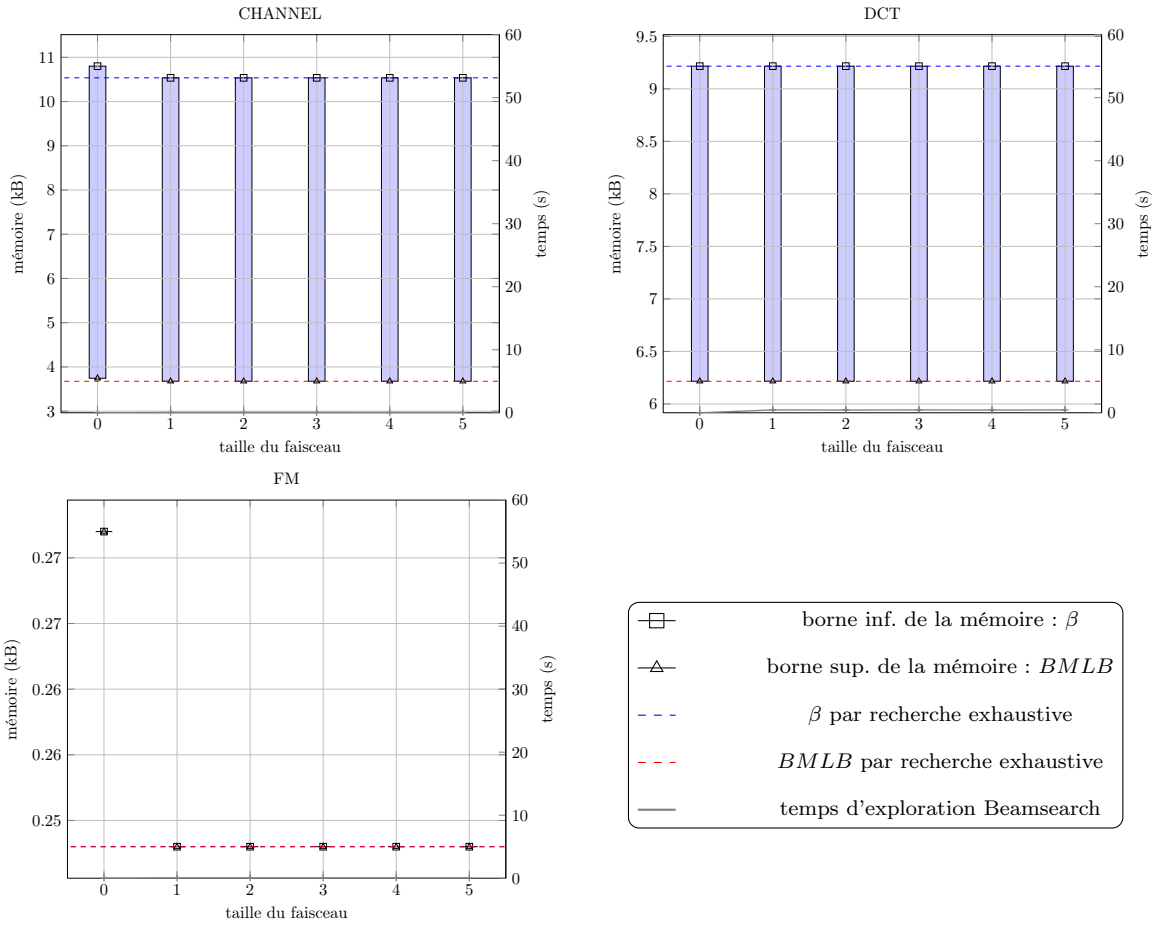


FIGURE 4.7 – **Réduction mémoire sur les programmes CHANNEL, FM et DCT. Ces programmes admettent peu de transformations. Le nombre de candidats engendrés est très faible et les réductions mémoire affichées sont négligeables.** Pour chaque taille de faisceau on a délimité l'espace entre la borne inférieure et la borne supérieure de la mémoire par une barre verticale ; les bornes mémoire se lisent sur l'échelle gauche. Lorsqu'elles sont connues, on a représenté les bornes inférieures et supérieures optimales obtenues par recherche exhaustive par les pointillées en rouge et en bleu. Enfin le temps d'exploration pour les différentes tailles de faisceau est donné par la ligne brisée grise et se lit sur l'échelle droite ; pour CHANNEL, FM et DCT le temps d'exploration est très court et se confond avec l'axe des abscisses.

Filtre de Gauss Dans le cas de GAUSS-10x10, l'heuristique converge rapidement (moins d'une minute) sur le candidat idéal avec une taille de faisceau de 3. Les réductions de mémoire sont importantes : -47.1% pour β et -49.4% pour *bmlb*. Dans cette configuration on peut garantir qu'indépendamment de l'ordonnancement choisi, l'application des transformations permettra de réduire la mémoire en monoprocesseur.

Dans le cas de GAUSS-100x100, le gain est plus faible : -30.8% pour β et -13.1% pour *bmlb*. Cela est probablement du au fait que les parties de GAUSS-100x100 qui n'admettent pas de transformations consomment plus de mémoire que celles de GAUSS-10x10. Le temps d'exploration est considérablement plus grand et la convergence plus lente. Cela peut paraître surprenant quand on pense que les graphes GAUSS-10x10 et GAUSS-100x100 sont identiques aux consommations/productions près sur les arcs. Nous avons déjà expliqué ce phénomène en § 3.7.5, certaines transformations vont produire un nombre d'arcs fonctions des productions/consommations dans L. Bien que nous évitions l'explosion combinatoire en éliminant les transformations qui créent trop d'arcs (ici nous avons fixé le nombre maximum d'arcs crées lors d'un déroulage, $MAXARCS$, à 20), l'augmentation de complexité est toujours présente. L'espace engendré pour GAUSS-100x100 est plus grand que l'espace engendré par GAUSS-10x10 ce qui explique la convergence plus lente et le temps d'exploration accru.

Filtre de Hough Dans le cas de HOUGH-10x10, l'heuristique converge avec une taille de faisceau de 1 vers la solution idéale. Le temps d'exploration est faible : de l'ordre d'une seconde. On obtient une réduction comparable sur les deux bornes de l'ordre de -25%.

Dans le cas de HOUGH-100x100, l'exploration exhaustive n'a pas terminé dans le temps imparti (une heure), nous ne connaissons donc pas le candidat idéal. Néanmoins il semblerait que l'heuristique de recherche converge avec une taille de faisceau de 3. Les réductions affichées sont comparables sur les deux bornes de l'ordre de -33%. Le temps d'exploration pour la méthode Beamsearch est raisonnable (en dessous de la minute).

Programmes de la suite StreamIt

Les résultats, que l'on commente ci-dessous, sont en figure 4.9.

Tri bitonique On remarque que pour le tri bitonique, les bornes inférieure et supérieure sont égales : sur cet exemple on sait donc comment construire un ordonnancement qui atteind la borne inférieure.

La recherche exhaustive sur BITONIC-32 dure plus d'une heure, on ne connait donc pas le meilleur candidat. La recherche par faisceau semblerait converger vers une solution qui réduit la mémoire de -24.3%. Néanmoins le temps d'exploration croît de manière exponentielle avec la taille du faisceau : pour les taille 4 et 5, l'exploration à été interrompue.

En réduisant la taille des entrées avec BITONIC-8, les temps d'explorations sont réduits drastiquement (moins de 10 secondes). La recherche exhaustive peut également être menée à bout et on voit que l'heuristique converge vers la meilleure solution (avec une réduction de mémoire de -29.6%).

DES Le comportement de DES est similaire à celui de BITONIC. Sur DES-16 l'exploration exhaustive est interrompue car trop longue. Par contre, DES-8 permet d'achever l'exploration exhaustive et montre que l'heuristique converge. Sur DES-8 et DES-16 la réduction sur les deux bornes est similaire, d'environ -30% pour β et -40% pour *bmlb*.

FFT Pour la transformée de Fourier, l'exploration converge rapidement vers le meilleur candidat (taille de faisceau de 4 pour un temps d'exploration inférieur à la minute). On affiche des réductions importantes : -55.6% pour β et -53.5% pour *bmlb*.

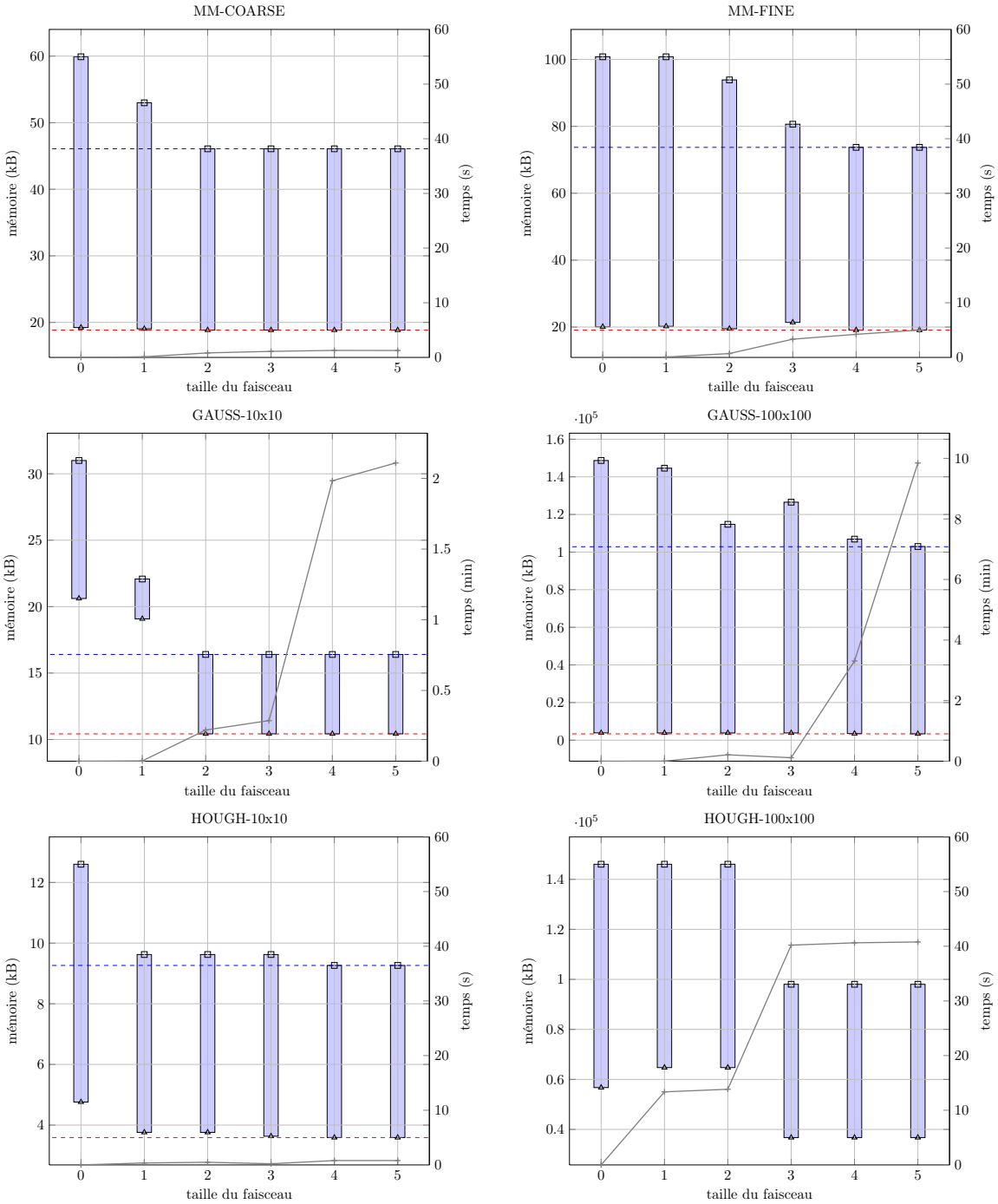


FIGURE 4.8 – **Réduction des bornes mémoire sur monocœur pour les programmes SLICES.** La legende est représentée sur la figure en regard. Pour chaque taille de faisceau on a délimité l'espace entre la borne inférieure et la borne supérieure de la mémoire par une barre verticale ; les bornes mémoire se lisent sur l'échelle gauche. Lorsqu'elles sont connues, on a représenté les bornes inférieures et supérieures optimales obtenues par recherche exhaustive par les pointillées en rouge et en bleu. Enfin le temps d'exploration pour les différentes tailles de faisceau est donné par la ligne brisée grise et se lit sur l'échelle droite.

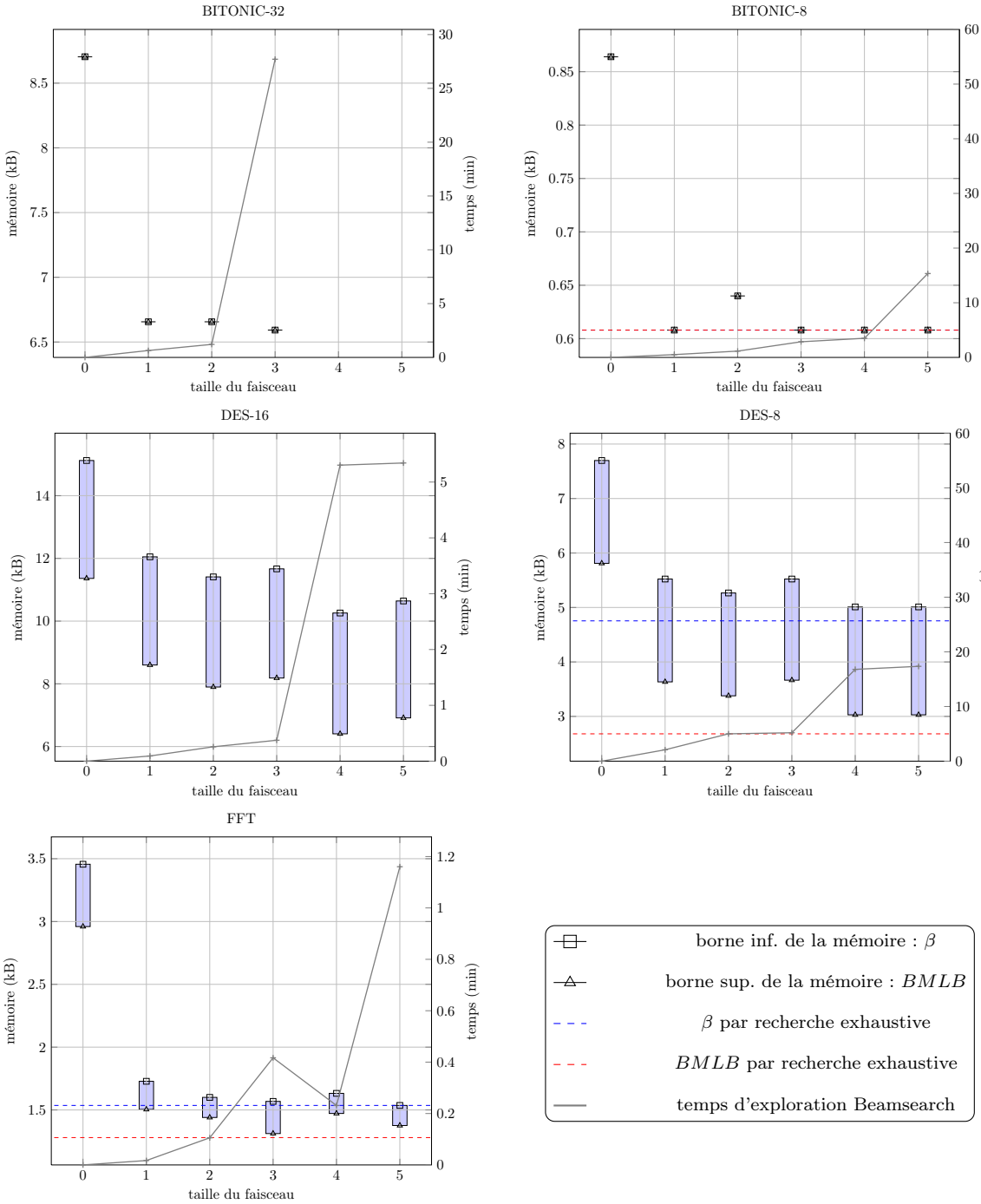


FIGURE 4.9 – **Réduction des bornes mémoire sur monocœur pour quelques programmes de la suite StreamIt.** Pour chaque taille de faisceau on a délimité l'espace entre la borne inférieure et la borne supérieure de la mémoire par une barre verticale ; les bornes mémoire se lisent sur l'échelle gauche. Lorsqu'elles sont connues, on a représenté les bornes inférieures et supérieures optimales obtenues par recherche exhaustive par les pointillées en rouge et en bleu. Enfin le temps d'exploration pour les différentes tailles de faisceau est donné par la ligne brisée grise et se lit sur l'échelle droite.

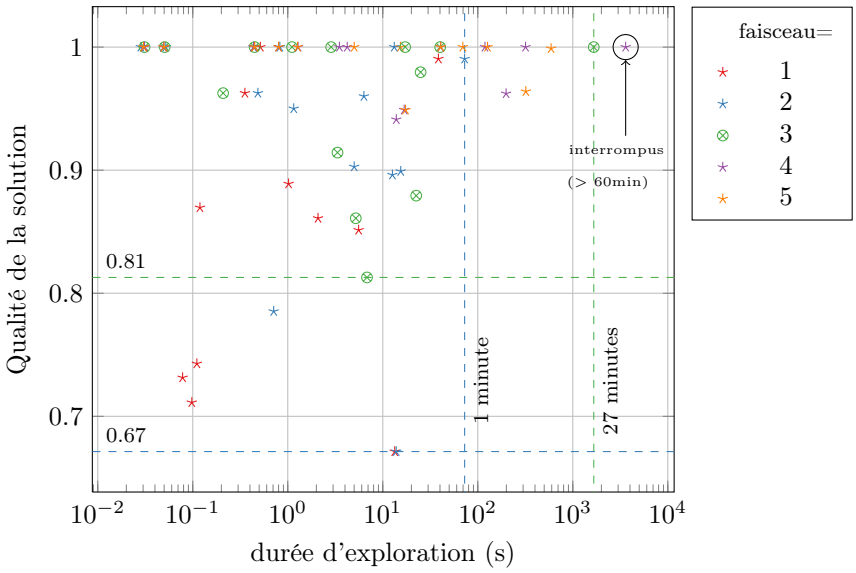


FIGURE 4.10 – Qualité de la solution et durée d'exploration pour différentes tailles de faisceau. La qualité de la solution est le rapport, $\frac{\beta(\text{meilleur})}{\beta(\text{faisceau})}$, entre la borne supérieure du meilleur candidat trouvé et la borne supérieure de la recherche par faisceau. Lorsque la recherche exhaustive ne termine pas, on calcule β(meilleur) sur le meilleur candidat trouvé par l'heuristique sur toutes les tailles de faisceau.

Évaluation de l'heuristique par faisceau

Quelle taille de faisceau choisir ? Il y a un compromis à faire dans le choix de la taille du faisceau : plus il est grand plus la solution va s'approcher de l'optimum, mais plus le temps d'exploration risque d'être grand. Nous avons représenté la qualité de la solution en fonction du temps d'exploration pour les différentes expériences réalisées sur la figure 4.10. Chaque classe de couleur correspond à toutes les expériences réalisées avec une même taille de faisceau. Remarquons que l'axe horizontal suit une échelle logarithmique.

Nous observons sur la figure que pour les classes de taille 4 et 5 l'exploration a été interrompue pour BITONIC-32. Ces classes sont donc trop coûteuses en temps. Considérons les points de la classe 3 en vert, la pire solution de cette classe n'est que 14% moins bonne que l'optimum et le temps d'exécution au pire est inférieur à 30 minutes. Pour les applications considérées, une taille de faisceau de 3 est un bon compromis entre qualité de la solution et temps d'exploration. Si 30 minutes nous parait trop long, on peut opter pour la classe 2 qui affiche pour des temps d'exploration inférieurs à la minute des et des solutions correctes.

Comparaison avec la recherche exhaustive Avec une taille de faisceau à 3, nous comparons le temps d'exécution de la recherche heuristique avec celui de la recherche exhaustive en table 4.2. Sur les programmes vraiment très petits (moins d'une seconde d'exploration), la recherche par faisceau affiche un léger surcoût lié à la gestion de la pile utilisée pour choisir les meilleurs candidats à chaque étape. Par contre sur les programmes dont l'espace d'exploration est vaste, la recherche par faisceau permet de gagner un temps considérable pour une réduction mémoire comparable (14% moins bonne que la recherche exhaustive dans le pire cas).

Programme	*Taille de faisceau 3 (s)*	*Exhaustif (s)*
MM-COARSE	1.11	1.1
MM-FINE	3.34	4.3
GAUSS-10x10	17.21	109.8
GAUSS-100x100	6.82	1131.6
BITONIC-32	1662.42	>3600
BITONIC-8	2.86	43.4
HOUGH-10x10	0.21	0.6
HOUGH-100x100	40.20	>3600
DES-16	22.49	>3600
DES-8	5.20	34.3
FFT	25.01	514.5
CHANNEL	0.05	0.1
DCT	0.45	0.4
FM	0.03	0.0

TABLE 4.2 – Temps d'exécution pour la recherche par faisceau et la recherche exhaustive.

4.3.4 Réduction de la mémoire en multiprocesseur

Nous venons de montrer que l'exploration des graphes par transformations permettait de réduire la mémoire en monoprocesseur significativement. Nous nous intéressons maintenant à la réduction mémoire pour une exécution parallèle sur multiprocesseur.

Nous considérons ici une architecture sans mémoire partagée : chaque processeur possède une zone de mémoire locale auquelle les autres processeurs ne peuvent accéder. Les communication entre processeurs se font par des transferts DMA entre leurs mémoires locales respectives. Nous supposerons que les tailles des mémoires locales sont homogènes de taille $memP$. Nous souhaitons ainsi réduire la taille mémoire nécessaire pour chaque partition.

Sur ce modèle d'exécution, la mémoire consommée par un programme sur multicœur dépend de l'ordonnancement (tout comme pour l'exécution sur un seul cœur) et du partitionnement. Dans [CLC+09] Choi propose une méthode pour adapter l'empreinte mémoire d'un ordonnancement SGMS aux contraintes mémoires de l'architecture. Cette méthode est basée sur un mapping qui prend en compte la mémoire disponible sur chaque processeur. Pour évaluer la réduction mémoire possible sur multicœur nous avons choisi d'utiliser la méthode de Choi. On veut également évaluer l'efficacité de notre méthode pour optimiser une méthode déjà existente de réduction de mémoire. Nous allons décrire le partitionnement proposé par Choi ci-dessous.

Chaque processeur possède une mémoire locale de taille $memP$. Nous souhaitons savoir qu'elle est la taille $memP$ minimale pour pouvoir executer un graphe G sur notre cible.

La méthode comporte trois étapes :

1. Ordonnancement du graphe par la méthode SGMS classique (décrite en § 4.1.4). On fait l'hypothèse pessimiste qu'il faut un transfert DMA entre chaque paire de nœuds.
2. On calcule une estimation pessimiste de la mémoire consommée par chaque arc e basée sur l'ordonnancement précédent.
3. On résout un programme linéaire pour essayer de trouver un partionnement sous la contrainte $memP$ en utilisant l'estimation pessimiste précédente.
4. On optimise l'assignation des étages dans l'ordonnancement en tenant compte du partitionnement réalisé.
5. On calcule la mémoire effectivement utilisée par chaque partition.

Mémoire d'un nœud Nous avons montré (§ 4.1.5) que dans l'ordonnancement SGMS la mémoire nécessaire pour assurer le transferts des données sur l'arc e s'écrit,

$$b(e) = (stage_m - stage_n + 1).\beta(e).coarse$$

.

Les nœuds producteurs et consommateurs sur un même cœur peuvent lire et écrire sur le même tampon. On choisit d'attribuer la mémoire de ce tampon au nœud producteur. Ainsi la mémoire des sorties d'un nœud m (à l'exception des nœuds `Duplicate`) s'écrit :

$$out\_bufs(m) = \sum_{\forall e \in \text{outputs}(m)} b(e)$$

Les nœuds `Duplicate` utilisent un seul tampon de production (§ 4.1.5) donc la formule devient,

$$out\_bufs(m) = \max_{\forall e \in \text{outputs}(i)} b(e)$$

Lorsqu'il y a un transfert DMA entre deux nœuds il faudra prévoir en plus un tampon distant de consommation pour pouvoir assurer le transfert, il faut donc considérer également pour tous les nœuds la mémoire nécessaire pour les entrées DMA,

$$in\_bufs(m) = \sum_{\forall e \in \text{dma\_inputs}(m)} (stage_e - stage_m + 1).\beta(e)$$

La mémoire totale nécessaire pour chaque nœud m s'écrit donc $b(m) = in\_bufs(m) + out\_bufs(m)$.

Partitionnement Pour partitionner le graphe, cette méthode résout un problème de programmation linéaire en nombre entiers. On note $b(n)$ la mémoire consommée par le nœud n.

On introduit un ensemble de variables binaires a_{nP}, telles que $a_{nP} = 1$ lorsque le nœud n appartient à la partition P. Chaque nœud doit être assigné sur exactement une partition,

$$\sum_{P \in \mathcal{P}} a_{nP} = 1, \quad \forall n \in G \tag{4.2}$$

Nous souhaitons équilibrer la charge de travail sur l'ensemble des partitions,

Minimiser II tel que,

$$\sum_{n \in G} w(n).a_{nP} \leq II, \quad \forall P \in \mathcal{P} \tag{4.3}$$

Enfin, la mémoire des nœuds d'une partition ne peut pas dépasser la mémoire disponible sur un cœur,

$$\sum_{n \in G} b(n).a_{nP} \leq memP, \quad \forall P \in \mathcal{P} \tag{4.4}$$

Si les $b(n)$ sont des constantes, ce problème est linéaire et sa résolution est possible avec un solveur ILP (dans notre implémentation glpsol-4.43 est utilisé). Cependant, les $b(n)$ dépendent de l'ordonnancement choisi et du partitionnement ; on ne peut donc pas les considérer comme des constantes.

La solution proposée par Choi consiste à faire une première estimation des $b(n)$ pessimiste en considérant que tous les nœuds sont sur des partitions différentes. Et d'utiliser cette estimation pour résoudre le problème linéaire ci-dessus. Si le problème admet une solution alors on peut essayer de réduire encore plus la mémoire consommée en recalculant l'assignation des étages. Pour cela on refait le scheduling SGMS en prenant en compte le partitionnement effectif, ce qui permet notamment d'éliminer un certain nombre de tampons entrants inutiles. Nous ne présenterons pas cette optimisation ici, le lecteur intéressé pourra consulter l'algorithme détaillé dans [CLC+09].

Calcul de la mémoire minimale par la méthode de Choi La méthode précédente permet de savoir s'il est possible d'exécuter notre programme avec une mémoire de taille $memP$ par processeur. Pour trouver le $memP$ minimal, Choi propose d'appliquer sa méthode avec des tailles $memP$ de plus en plus petites jusqu'à ce que le problème linéaire n'admette pas de solution. On notera la borne minimale obtenue, $minP$.

4.3.5 Choix de ϕ et exploration en multiprocesseur

Pour réduire la mémoire $minP$, il faut réduire les $b(n)$. Rappelons que les $b(n)$ sont le maximum ou le cumul des $(stage_m - stage_n + 1).\beta(e)$. Lors de l'exploration du graphe G nous ne conaissons pas encore les étages des nœuds, nous ne pouvons donc pas travailler sur la partie $(stage_m - stage_n + 1)$. [2] Nous allons donc essayer de réduire les $b(e)$. Pour diminuer $minP$ il faut réussir à partitionner efficacement les $b(n)$ parmi les cœurs. En transformant les nœuds les plus gourmands en mémoire en un ensemble de nœuds plus petits nous pouvons espérer obtenir un meilleur partitionnement où la mémoire consommée est équitablement répartie sur l'ensemble des cœurs.

Guidés par les considérations précédentes nous avons choisi

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

Autrement dit, la fonction économique cherche à réduire le maximum de β sur le graphe et si plusieurs candidats sont à égalité elle essaye de réduire le cumul des β sur le graphe. Intuitivement cette fonction économique cherche à diviser les arcs les plus gourmands en mémoire en plusieurs arcs plus petits. Lorsque ceux-ci ne peuvent plus être réduits, la fonction économique cherche à réduire la mémoire totale du graphe.

4.3.6 Résultats multiprocesseur

Nous avons mesuré la réduction de $minP$ avec l'heuristique de recherche par faisceau et la recherche exhaustive sur une architecture avec **4 cœurs**. L'ensemble des résultats sont repris sur les figures 4.11 et 4.12. On n'a pas figuré DCT, FM et CHANNEL pour lesquels les réductions obtenues sont négligeables (comme pour le cas monoprocesseurs ces programmes admettent très peu de transformations).

Les réductions obtenues sont importantes et les temps d'explorations sont comparables à ceux obtenus sur monocœur. On peut noter tout de même que l'utilisation de la nouvelle fonction économique aurait tendance à accélerer l'exploration de BITONIC-32.

Convergence vers l'optimal Dans le cas monocœur la convergence vers l'optimal est très rapide sur tous les programmes testés. Pour les résultats multicœur, certains programmes (BITONIC-8 et GAUSS-100x100) ne convergent pas de manière aussi concluante vers l'optimum exhaustif. Nous pensons que ceci est dû à ce que la métrique utilisée n'est pas directement $minP$, ce qui peut créer des résultats intermédiaires dans l'exploration qui bien que minimisant ϕ ne sont pas les meilleurs pour $minP$. Néanmoins pour la plus part des cas, ϕ s'avère une bonne métrique pour diminuer $minP$.

Variation de la réduction selon le nombre de cœurs

En augmentant le nombre de cœurs, le nombre de partitions augmente. On peut donc supposer que la taille de chaque partition va être plus petite et donc que $minP$ va être plus petit. Ce raisonnement est vrai lorsque les nœuds ont tous une consommation mémoire homogène. Par contre

2. Cependant l'optimisation des étages de Choi réduira par la suite la différence entre les étages $(stage_m - stage_n + 1)$.

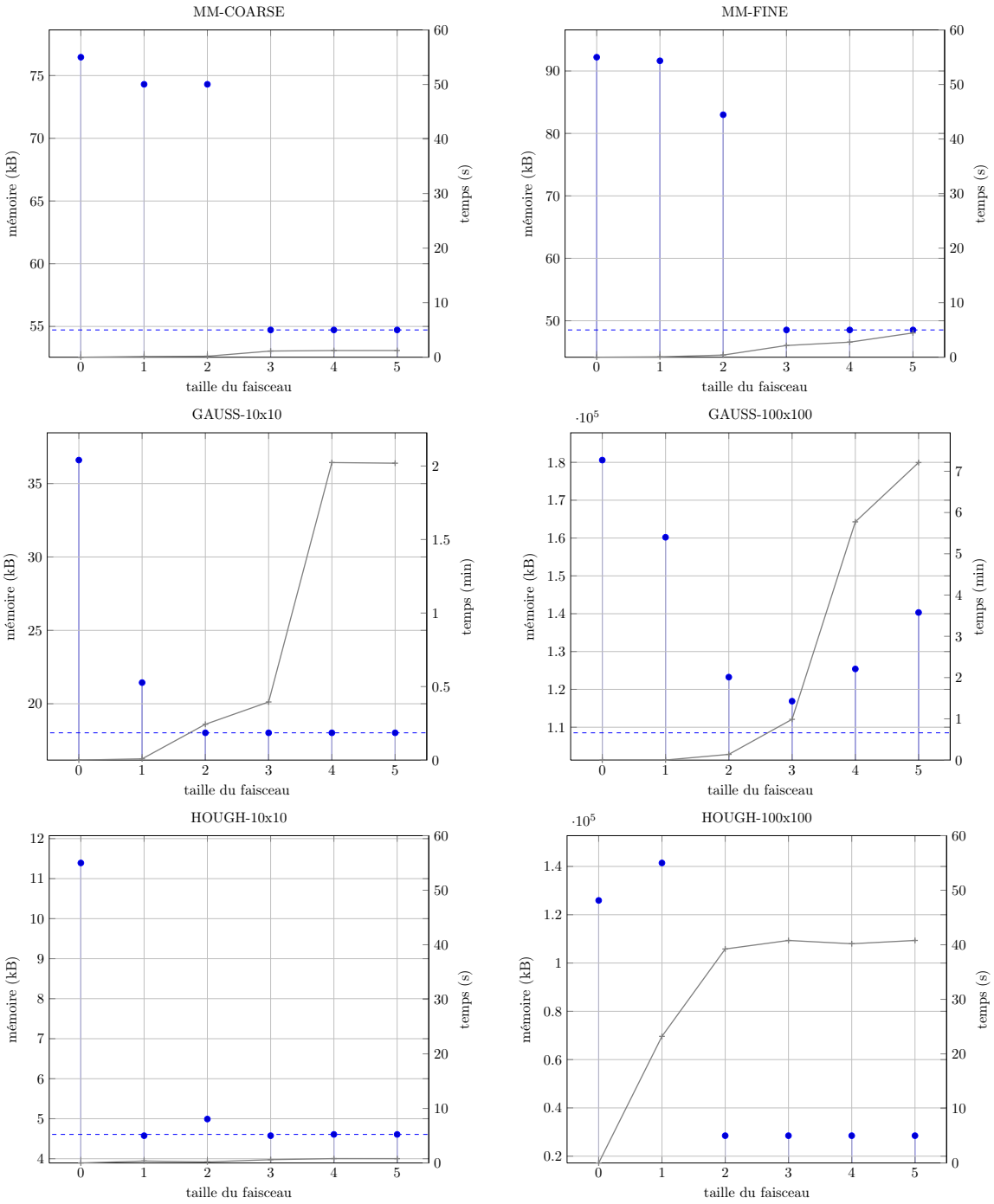


FIGURE 4.11 – **Réduction de $minP$ (calculé avec la méthode de Choi) sur 4 cœurs pour les programmes SLICES.** La légende se trouve sur la figure en regard. Pour chaque taille de faisceau on a représenté la taille $minP$ obtenue par les barres verticales. Lorsque le $minP$ optimum obtenu par recherche exhaustive est connu, il est indiqué par les pointillées. Enfin le temps d'exploration pour les différentes tailles de faisceau est donné par la ligne brisée grise et se lit sur l'échelle droite.

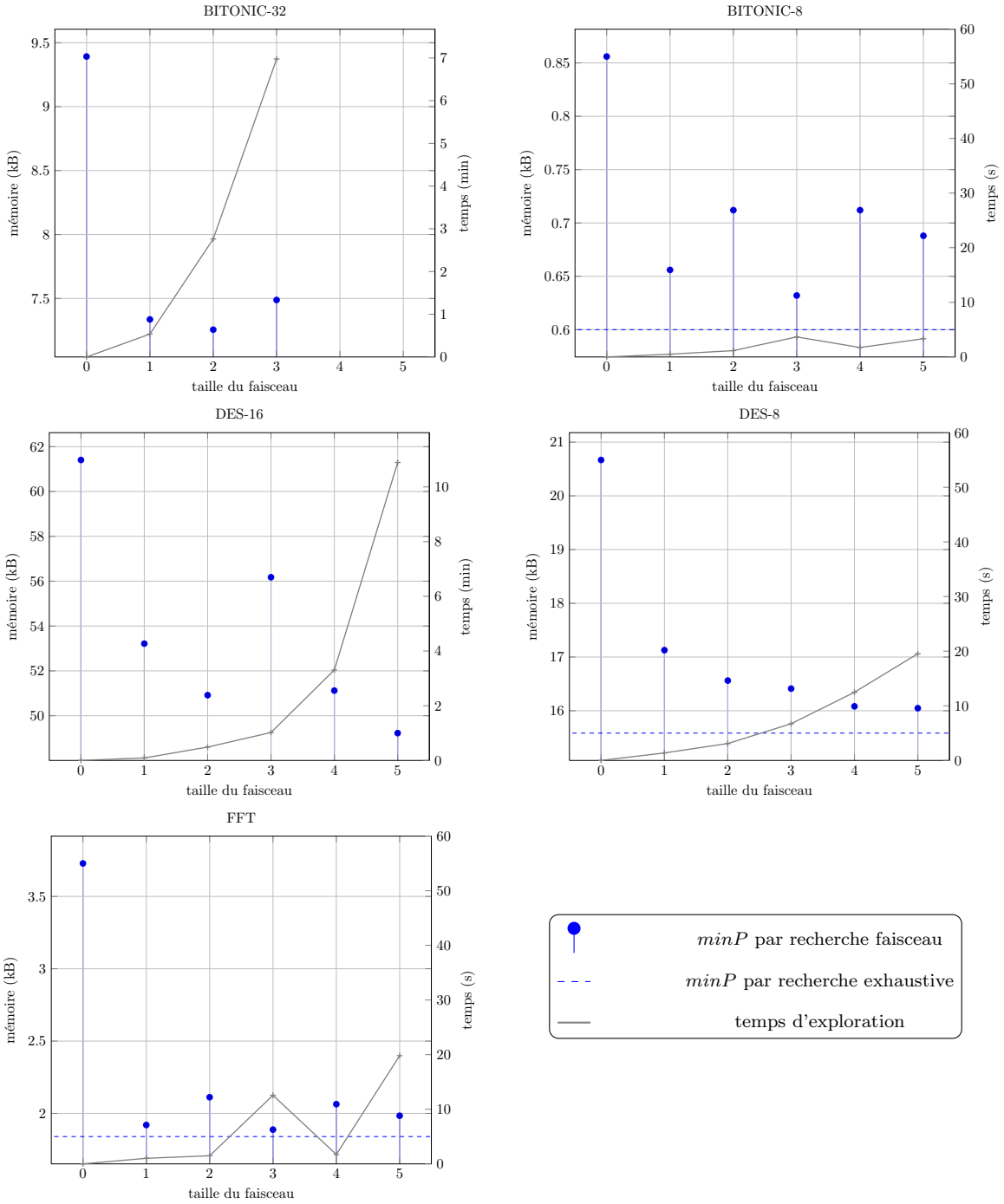


FIGURE 4.12 – **Réduction de $minP$ (calculé avec la méthode de Choi) sur 4 cœurs pour les programmes de la suite StreamIt.** Pour chaque taille de faisceau on a représenté la taille $minP$ obtenue par les barres verticales. Lorsque le $minP$ optimum obtenu par recherche exhaustive est connu, il est indiqué par les pointillées. Enfin le temps d'exploration pour les différentes tailles de faisceau est donné par la ligne brisée grise et se lit sur l'échelle droite.

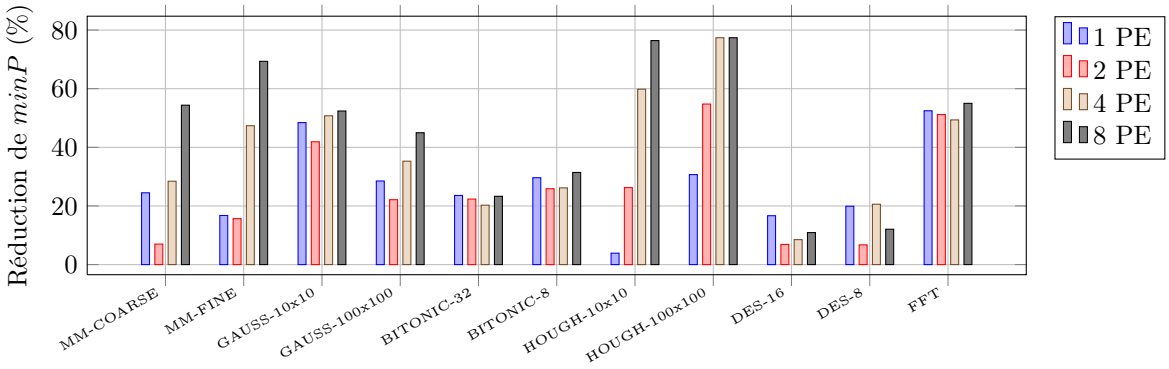


FIGURE 4.13 – Réduction de la mémoire en multiprocessoeur en fonction du nombre de cœurs utilisés (pourcentage $\frac{minP(\text{faisceau3}) - minP(\text{original})}{minP(\text{original})}$)

lorsque le programme original compte un nœud très gourmand en mémoire, l'ajout de nouvelles partitions ne permet pas de réduire la taille mémoire ; en effet la taille mémoire des partitions ne pourra jamais être plus petit que les besoins du nœud gourmand.

En utilisant les transformations du chapitre 3 il est parfois possible de casser un nœud en plusieurs nœuds plus petits, de manière à mieux répartir la consommation mémoire sur les différentes partitions. Pour étudier ce phénomène nous avons mesuré la réduction mémoire en fonction du nombre de cœurs utilisés (cf. figure 4.13). On peut séparer les programmes en deux classes :

- **Plateau**, les graphes de cette catégorie Bitonic, FFT, DES, affichent une réduction mémoire qui ne varie pas avec le nombre de cœurs.
- **Croissants**, pour les graphes de cette catégorie MM-Fine, MM-Coarse, Gauss, Hough, la réduction mémoire augmente avec le nombre de cœurs.

Les graphes *Croissants* cassent les nœuds les plus gourmands en mémoire en plus petits nœuds. Le partitionnement de Choi peut donc mieux répartir le coût mémoire sur les différentes partitions. Pour ces programmes là, plus on augmente le nombre de cœurs plus la réduction mémoire est importante. Bien entendu il y a une limite : lorsque nous avons ajouté suffisamment de partitions pour répartir les nœuds cassés, la réduction mémoire ne diminuera plus.

Les transformations des graphes *Plateau* sont plus uniformes : certaines parties du graphe sont simplifiées ce qui permet de réduire la mémoire totale du graphe, mais on ne casse pas les nœuds les plus gourmands ; et donc la réduction n'augmente pas avec l'ajout de cœurs.

4.3.7 À quoi sont dues les réduction mémoire ?

Les réductions mémoire précédentes en monocœur et multicœur sont obtenues grâce à différents facteurs :

Simplifications Les simplifications enlevent des nœuds ou des arcs inutiles, elles permettent donc d'économiser de la mémoire. Ces transformations réduisent la consommation mémoire en monocœur et multicœur.

Séparation des nœuds en plus petits éléments Ce premier phénomène à été mis en évidence en § 4.3.6 : certaines transformations (et en particulier les transformations qui suppriment des

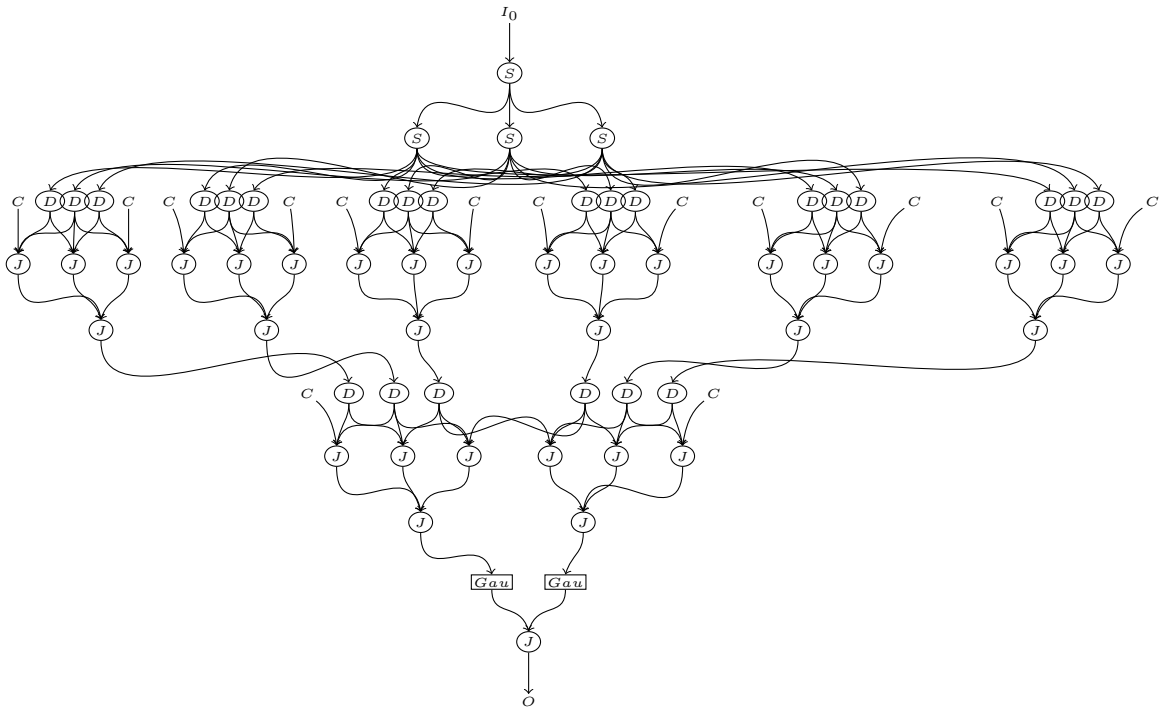


FIGURE 4.14 – Meilleur candidat trouvé pour réduire la mémoire de GAUSS-10x10 en multiprocesseur avec la recherche par faisceau 3. Les transformations proposées ont cassé l'extraction des blocs en des parties plus petites. (Le graphe original se trouve figure 2.11 en page 33.)

points de synchronisation) permettent de réecrire un nœud gourmand en mémoire en utilisant un ensemble de plus petits nœuds. Ce type de réecriture permet une répartition plus uniforme des besoins en mémoire de l'application, elle sont donc particulièrement utile dans le cas de la réduction mémoire en multiprocesseur. Nous avons représenté la séparation des nœuds en plus petits éléments sur GAUSS en figure 4.14.

Séparation des cycles La méthode d'ordonnancement que nous utilisons fusionne les cycles (§ 4.1.4). Les cycles ne peuvent donc pas être répartis sur différents processeurs. Pourtant les transformations proposées permettent dans les deux exemples MM-FINE et HOUGH de séparer les cycles en plusieurs cycles plus petits. Ceci est particulièrement intéressant puisque on peut ainsi réduire la granularité de chaque cycle après fusion. Ce procédé est utile dans le cas multiprocesseur. Nous avons représenté la séparation des cycles dans l'exemple HOUGH sur la figure 4.15.

4.3.8 Discussion

Notre méthode affiche des réductions de l'empreinte mémoire significatives sur monocœur et multicœur. L'utilisation de l'heuristique Beamsearch se révèle efficace, puisqu'elle accélère considérablement le temps d'exploration et dégrade peu la solution.

Et la fusion ? La diminution de mémoire sur monocœur vient principalement d'une simplification du graphe. Par exemple un nœud `Join` à une seule branche va être éliminé car il n'a aucun effet sur l'ordre des données. On peut obtenir des résultats similaires en fusionnant tous les nœud

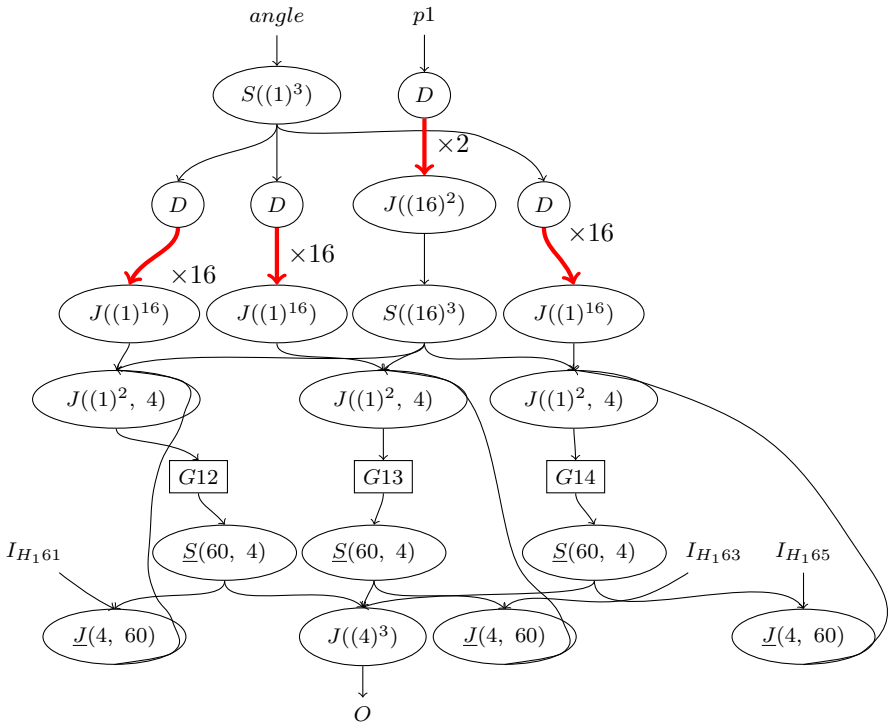


FIGURE 4.15 – Meilleur candidat trouvé pour réduire la mémoire de HOUGH avec la recherche par faisceau 3. Le cycle dans le graphe a été separé en trois cycles plus petits qui travaillent chacun avec une valeur d'angle différente. Ceci correspond à la parallélisation classique du filtre de Hough qui découpe l'espace transformé selon les angles. (Le graphe original se trouve figure 2.12 en page 35.)

du graphe. En effet en fusionnant les nœuds du graphe, on va exprimer les communications entre nœuds en utilisant des variables au sein de la même fonction. Les optimisations du compilateur C utilisé (`gcc-4.3`) réalisent des optimisations similaires à celles obtenues par exploration. Par exemple le code généré pour un nœud `Join` à une seule branche va être optimisé par *gcc* qui va éliminer la copie entre l'entrée et la sortie. La fusion n'est pas toujours envisageables car elle entraine une augmentation, parfois très importante, de la taille du code générée. Néanmoins en fusionnant des groupes d'une dizaine de nœuds à chaque fois, on peut vraissemblablement obtenir des réductions comparables.

Au contraire, en multiprocesseur l'intérêt de notre méthode ne fait pas de doute. La diminution de mémoire obtenue sur multicœur provient principalement de la réecriture de certains sous-graphes en plus petits éléments (ce qui permet une meilleure répartition des coûts mémoire). On ne pense pas que ces réductions puissent être obtenues avec la fusion, puisque celle ci regroupe les nœuds au lieu de les séparer.

Faut-il vraiment deux métriques pour le cas mono et multicœur ? En réalité, les deux métriques arrivent sur l'ensemble des programmes à des graphes identiques (à la symétrie près) pour la *recherche exhaustive*. Ceci est du au fait que pour réduire la somme des β il faut souvent réduire le maximum et *vice versa*. Néanmoins, il est important de considérer deux métriques lorsque l'on utilise l'heuristique. Dans le cas multiprocesseur par exemple, si on n'utilisait pas la métrique $max_G(\beta(G))$, la solution convergera moins rapidement vers le candidat dont les gros consommateurs de mémoire ont été cassés en nœuds plus petits et qui donc minimisera $minP$.

D'autres heuristiques ? Nous avons réalisé des expériences comparables avec une heuristique par partitionnement : le graphe est découpé, on explore chaque partition de manière isolée pour trouver la meilleur sous-graphe pour cette partition, et on recole les meilleurs sous-graphes. On ne sait appliquer cette heuristique qu'aux fonctions ϕ qui vérifient $\phi(G_1 \cup G_2) = f(\phi(G_1), \phi(G_2))$, où f est une fonction monotone ; ce qui est le cas des deux fonction économiques présentées dans cette section. Les résultats obtenus par cette autre heuristique, présentés dans [dOCLB10b], sont comparables, mais l'heuristique est moins efficace et plus contraignante. En § 4.4, nous proposons une heuristique de recherche itérative, qui pourrait éventuellement s'adapter à la réduction mémoire.

4.4 Optimisation des communications

Dans la section précédente nous avons vu qu'avec la métrique ϕ_{mem} il était possible de réduire considérablement la mémoire des programmes SJD. Nous allons maintenant nous intéresser à une autre application de la méthode d'exploration de graphes : la réduction du coût des communications.

Les programmes dirigés par les données, sont parfois composés de filtres dont le travail est léger en comparaison avec les volumes de données copiés d'un nœud à l'autre. C'est par exemple le cas de FFT ou de BITONIC. Dans ces programmes, le facteur limitant la performance est le coût des communications. Il est donc critique de réduire leur coût si on veut obtenir des bonnes performances.

Le calcul du coût des communications d'un programme SJD a été développé en § 4.1.3. Les communications au sein d'une même partition ne sont pas prises en compte puisqu'elles se font soit à travers les registres (si les nœuds sont de petite taille et fusionnés), soit à travers le cache ou une mémoire locale rapide. Au contraire, les communications entre deux partitions sont coûteuses et modélisées par $c_e = c_0 + \frac{s(e).\beta(e)}{bw}$. Le coût total des communications entre partitions s'écrit $C(\mathcal{P})$ et correspond à la somme des coûts sur les arcs entre les différentes partitions de $\mathcal{P}$.

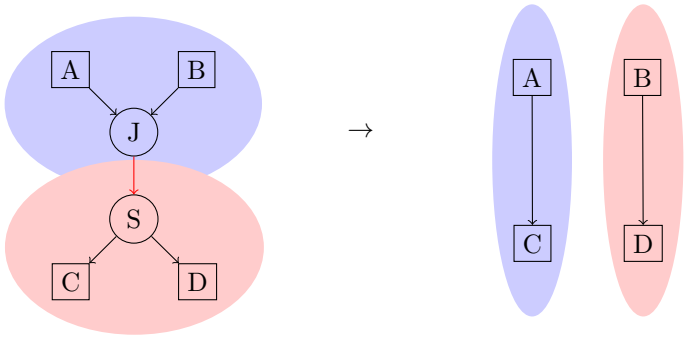
Une version préliminaire des résultats de cette section a été présentée dans [dOCLB10c].

4.4.1 Choix de ϕ et exploration

Pour calculer le coût en communications il nous faut connaître le partitionnement : on ne considère que les coûts de communications entre les cœurs. Le problème c'est que la phase d'exploration est effectuée avant le partitionnement (cf. figure 4.1) ; on ne peut donc pas, à priori, utiliser l'information de partitionnement dans le calcul de ϕ. Pour lever ce problème nous proposons plusieurs approches.

Choix d'un ϕ indirect Pour avoir un partitionnement qui diminue les coûts de communication, il faut mettre dans la même partition les nœuds qui s'échangent beaucoup de données entre eux, et au contraire séparer les nœuds qui communiquent peu entre eux. On souhaiterait donc, avec les transformations proposées, faire apparaître des grappes de nœuds indépendantes entre elles.

Une stratégie pour favoriser cette configuration, est l'élimination des goulets d'étranglement dans le graphe. En effet la présence de goulets d'étranglements empêche un partitionnement en parties indépendantes, comme sur l'exemple ci-dessous :



Sur cet exemple on a éliminié la jonction $J - S$ ce qui permet au partionneur de trouver deux sous-graphes indépendants. Bien entendu, la jonction $J - S$ ne peut être éliminée que dans les configurations où il n'y a pas de dépendences entre AC et BD.

Guidés par cette intuition, on va emettre l'hypothèse que les nœuds `Join` avec beaucoup de canaux entrants sont à éliminer si on veut réduire le coût de communications. On va donc essayer de réduire leur poids en choissisant :

$$\phi_0(G) = \text{ moyenne }_{N \in G \text{ tq } |in(N)| \geq 2}(\sum_{e \in in(N)} c_e),$$

c'est-à-dire on va considérer la moyenne du coût des nœuds à plusieurs branches d'entrée.

Bien entendu, cette métrique est basée sur une stratégie *ad hoc*, elle ne garantit donc pas de réduire le coût des communications et peu parfois l'augmenter. On verra dans la suite que pour la plus part des programmes, elle donne tout de même des résultats corrects.

Prise en compte du partitionnement durant l'exploration Une deuxième solution consisterait à partitionner chacun des candidats explorés, ce qui nous permettrait de mesurer directement le coût en communications du graphe. Cette solution, bien qu'idéale, est bien trop coûteuse à mettre en place en pratique. En effet on multiplierait la complexité de l'exploration par la complexité du partitionnement.

Exploration itérative La troisième solution proposée est un compromis pour pouvoir utiliser les informations de partitionnement dans ϕ tout en évitant de partitionner chaque graphe exploré. L'exploration itérative consiste à alterner une phase de partitionnement et une phase d'exploration.

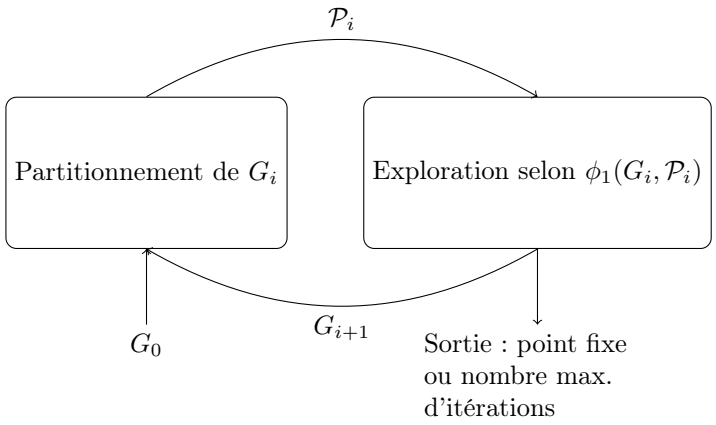


FIGURE 4.16 – L'exploration itérative permet de prendre en compte le partitionnement pour le calcul de ϕ.

On va commencer par calculer un partitionnement $\mathcal{P}_0$ sur le graphe original, puis on va explorer l'espace des transformations avec une recherche par faisceau en utilisant $\phi_1 = C(\mathcal{P}_0)$, soit le coût des communications selon le partitionnement original ; on va produire un graphe transformé G'. Les transformations trouvées par la phase d'exploration vont peut-être aider le partitionneur à trouver une meilleure configuration, on va donc partitionner le nouveau graphe G' en produisant $\mathcal{P}_1$. On va répéter ce processus jusqu'à ce que la solution converge vers un point fixe, ou si la solution ne converge pas assez vite (on a prouvé qu'il n'existe pas de dérivations infinies par nos transformations) s'arrêter au boût d'un nombre d'itérations suffisament grand. L'algorithme est schématisé en figure 4.16.

4.4.2 Mesures de la réduction du coût des communications

Protocole expérimental

Pour réaliser les expériences qui suivent nous avons considéré trois méthodes :

- Recherche par faisceau, taille de faisceau 3, avec ϕ_0 indirect présenté dans la section précédente ;
- Recherche exhaustive avec ϕ_0 indirect présenté dans la section précédente ;
- Recherche itérative avec $\phi_1 = C(\mathcal{P})$ direct, pour la phase d'exploration on fait une recherche par faisceau de taille 1 ; on réalise au maximum 20 itérations.

On va s'intéresser à la réduction des communication avant et après transformations. Le graphe original et le graphe transformé vont être partitionnés à l'aide de METIS comme décrit en § 4.1.3, les partitions respectivement obtenues seront notées $\mathcal{P}$ et $\mathcal{P}'$. On veut donc mesurer,

$$R = \frac{C(\mathcal{P}') - C(\mathcal{P})}{C(\mathcal{P})} = \frac{\sum_{e' \in inter(\mathcal{P}')} c_0 + \beta(e').s(e')/bw}{\sum_{e \in inter(\mathcal{P})} c_0 + \beta(e).s(e)/bw} - 1.$$

Nos mesures vont donc être affectés par le facteur $bw.c_0$ qui dépend de la latence et de la bande passante de l'architecture considérée. On considère deux architecture théoriques à **4 cœurs** de calcul : la première architecture modélise une communication où la latence est négligeable (par exemple la communication entre CPU à travers un cache partagé) ; la deuxième architecture modélise une communication où la latence est forte (par exemple des transferts DMA entre CPU et GPU).

Architectures à faible latence

Pour modéliser les coûts de communication sur des architectures à faible latence, on va considérer sur le modèle de Hockney que $c_0 = 0$, le choix de bw n'est pas important puisqu'il disparait en faisant le rapport R.

Pour un avoir un apperçu des trois méthodes nous avons représenté le rapport R en fonction de la durée d'exploration sur la figure 4.17. Au niveau du temps d'exploration, les méthodes par faisceau et itératives affichent des temps similaires à ceux obtenus en § 4.4. La méthode itérative se trouve à mi-chemin entre ces deux méthodes : légèrement plus coûteuse que la recherche exhaustive, elle reste toujours en dessous de la barre des 30 minutes.

En ce qui concerne la réduction des coûts de communications, la méthode itérative est sans conteste la meilleure. Les résultats en détail, programme par programme, sont repris sur la figure 4.18. On constate qu'à l'exception de BITONIC-8 (où la recherche exhaustive est légèrement supérieure), la recherche itérative est égale ou meilleure aux deux autres méthodes sur tous les programmes. Ce n'est pas un résultat étonnant, la recherche itérative optimise directement le coût en communication alors que la recherche exhaustive et par faisceau optimisent ϕ_0, la taille moyenne des nœuds goulets. La stratégie indirecte ϕ_0 marche plutôt bien sur certains programmes, mais est moins précise que ϕ_1.

Sur GAUSS-100x100, la stratégie d'élimination des goulets se révèle néfaste : elle augmente en fait le coût mémoire du programme. L'optimisation de ϕ_0 a éclaté le graphe en des morceaux trop petits, au prix d'un nombre de nœuds accru ; en rendant la tâche du partitionneur plus ardue.

La réduction du coût total des communications est importante car elle va libérer de la bande passante sur le bus de communication. Un deuxième paramètre intéressant est le coût de communication maximal par processeur (c'est-à-dire le temps passé au maximum par chaque processeur pour copier les données sur un ordonancement stationnaire). Pour cela nous mesurons $C_{max} = max(\{\sum_{e \in inter(P,Q)} c_e : \forall P, Q \in \mathcal{P}\})$. La réduction de C_{max} mesurée est représentée en figure 4.19, nous observons que les résultats sont quasiment identiques à ceux de la courbe précédente : à l'exception de BITONIC-32 qui n'affiche aucume réduction de C_{max}.

Sur les architectures à faible latence, les trois méthodes affichent des réductions significatives. Néanmoins la méthode itérative se révèle plus efficace que les deux autres.

Architecture à forte latence

Pour modéliser les coûts de communication sur des architectures à forte latence, on va prendre comme référence la communication GPU/CPU en reprenant les valeurs mesurées dans [VD08] : $c_0 = 11\mu s$ et $bw = 3.3Gb/s$. Sur une telle architecture, si à chaque execution d'un nœud seul quelques élements sont envoyés la performance est désastreuse car les coût sont dominés par la latence. Il faut donc augmenter la charge utile de chaque envoi de données en gonflant le grain des tâches comme expliqué en § 4.1.5. Pour les expériences ci-dessous on a choisi un facteur de $COARSE = 1024$.

On a réalisé les mêmes expériences que pour le cas des architectures à faible latences, en figures 4.20, 4.21 et 4.22. Pour la plus part des programmes, les résultats sont similaires. On remarque néanmoins que pour BITONIC-8, BITONIC-32, DES-16 et MM-COARSE, les réductions obtenues sont faibles voir négatives (augmentation des coûts en mémoire) pour la recherche exhaustive et par faisceau alors que la recherche itérative affiche de bons résultats. Nous attribuons ce mauvais comportement des exploration basés sur ϕ_0 à l'utilisation d'une métrique indirecte qui ne nous garantit aucunement une réduction des communications. En particulier, les architectures à forte latence posent un problème intéressant : l'augmentation du nombre d'arêtes entre partitions conduit à une augmentation du coût de communications, en effet pour chaque arête on doit « payer » le coût de latence c_0 pour établir la communication. La métrique ϕ_0 ne prend pas du tout en compte ce phénomène contrairement à la métrique ϕ_1 utilisée par la recherche itérative.

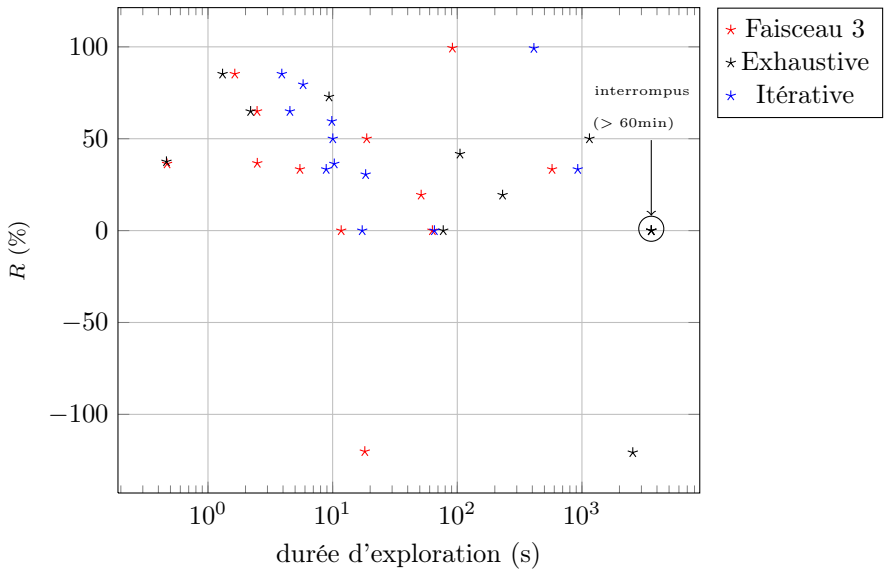


FIGURE 4.17 – Réduction du coût de communication global en fonction de la durée d'exploration pour l'ensemble des programmes sur une architecture faible latence disposant de 4 cœurs. On compare les méthodes d'exploration : par faisceau, exhaustive et itérative.

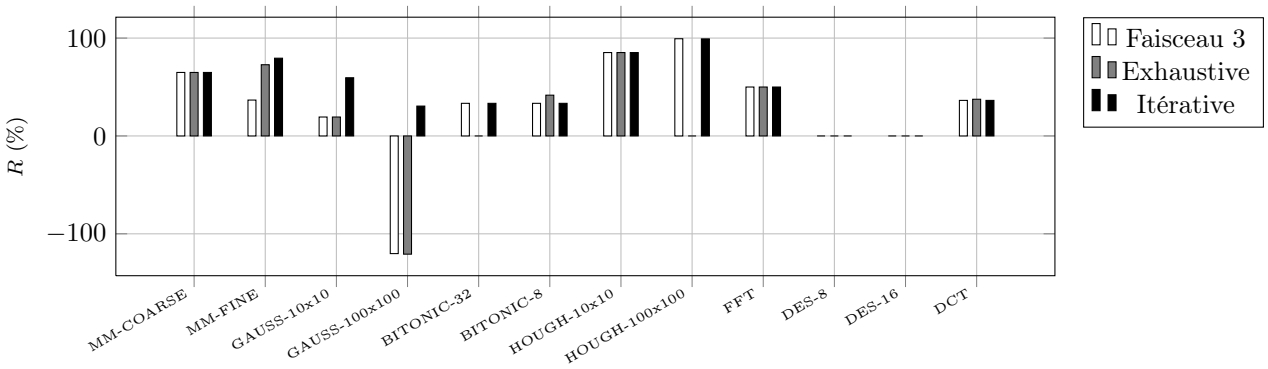


FIGURE 4.18 – Réduction du coût de communication global sur une architecture à faible latence disposant de 4 cœurs pour les différents programmes.

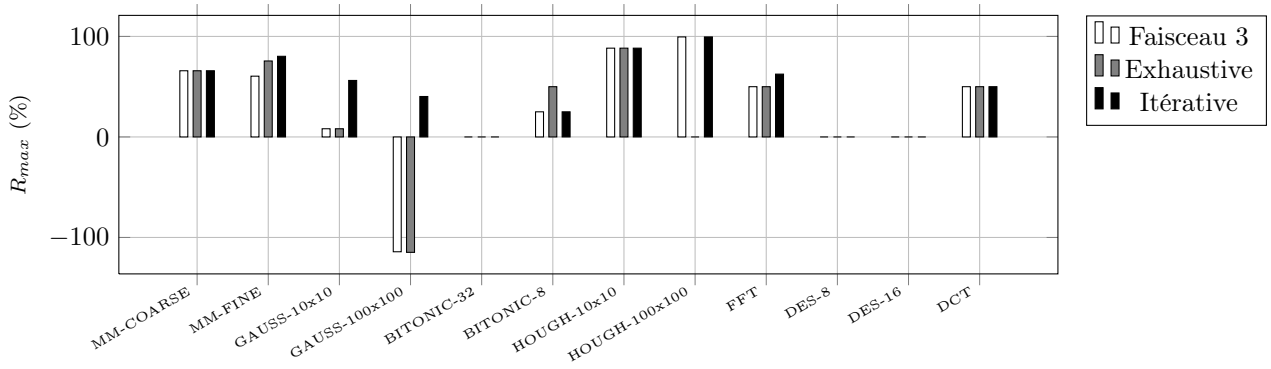


FIGURE 4.19 – Réduction du coût de communication maximum entre deux partitions sur une architecture à faible latence disposant de 4 cœurs pour les différents programmes.

L'utilisation de la méthode itérative sur des architectures à forte latence est donc vivement recommandée.

4.4.3 Impact sur le temps d'exécution

Dans la section précédente nous avons montré que les transformations que nous proposons permettent de réduire considérablement le coût des communications. Le transfert des données est souvent le goulet d'étranglement de performance pour les programmes dirigés par les données. La réduction du coût de communication devrait donc avoir un impact positif sur la vitesse d'exécution des programmes. Dans cette section nous allons vérifier cette hypothèse en comparant la vitesse d'exécution des programmes avant et après transformations.

Architecture Ne disposant pas d'un processeur multicœur embarqué nous avons décidé de mesurer les performances des programmes sur un multicœur classique. Il s'agit d'un quadricœur Nehalem (Xeon W3520 cadencé à 2.67GHz) disposant de 256KB de cache L2 pour chaque processeur et d'un cache L3 partagé de 8MB. On se place donc dans le cadre d'une architecture à faible latence.

Protocole expérimental

Nous avons sélectionné parmis les benchmarks, les programmes provenant de StreamIt (pour avoir une référence) dont on réduisait significativement le coût en communications sur une architecture à faible latence : MM-COARSE, BITONIC-32, DCT, FFT.

Nous avons considéré deux versions de chaque programme :

- version *originale*, le graphe original sans transformations.
- version *itérative*, le meilleur graphe obtenu par la recherche itérative décrite dans la section précédente.

Nous avons compilé ces deux versions avec notre backend SJD en utilisant le partitionneur METIS. Les facteurs de *coarse* utilisés pour le gonflement des tâches, obtenus par profilage manuel (§ 4.1.5) sont :

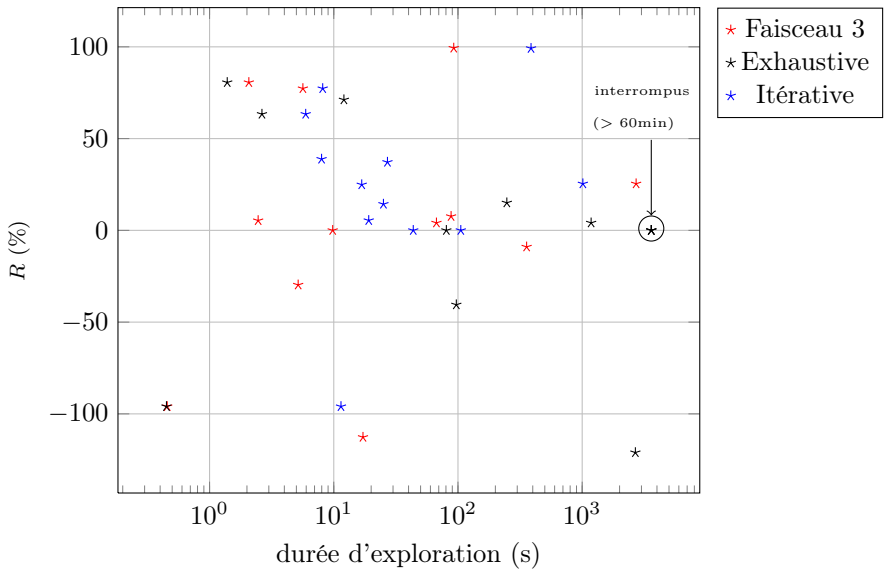


FIGURE 4.20 – Réduction du coût de communication global en fonction de la durée d'exploration pour l'ensemble des programmes sur une architecture forte latence disposant de 4 cœurs. On compare les méthodes d'exploration : par faisceau, exhaustive et itérative.

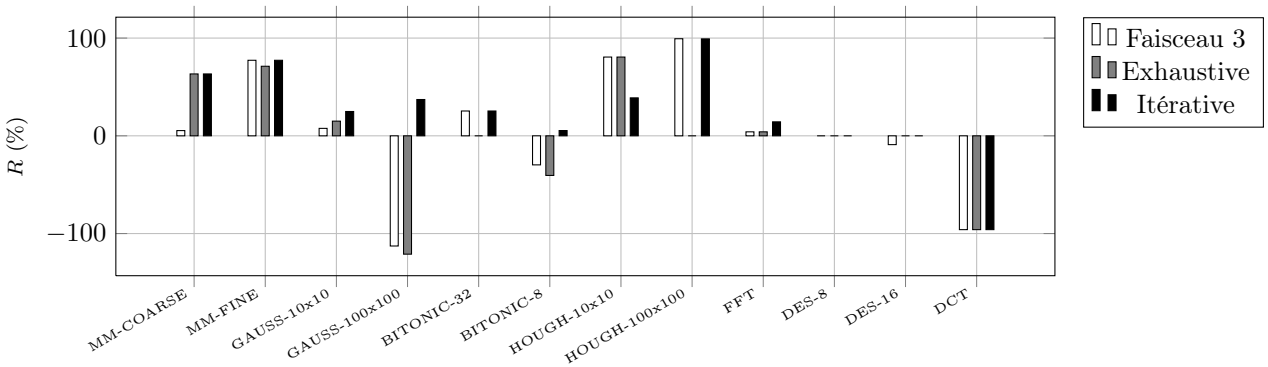


FIGURE 4.21 – Réduction du coût de communication global sur une architecture à forte latence disposant de 4 cœurs pour les différents programmes.

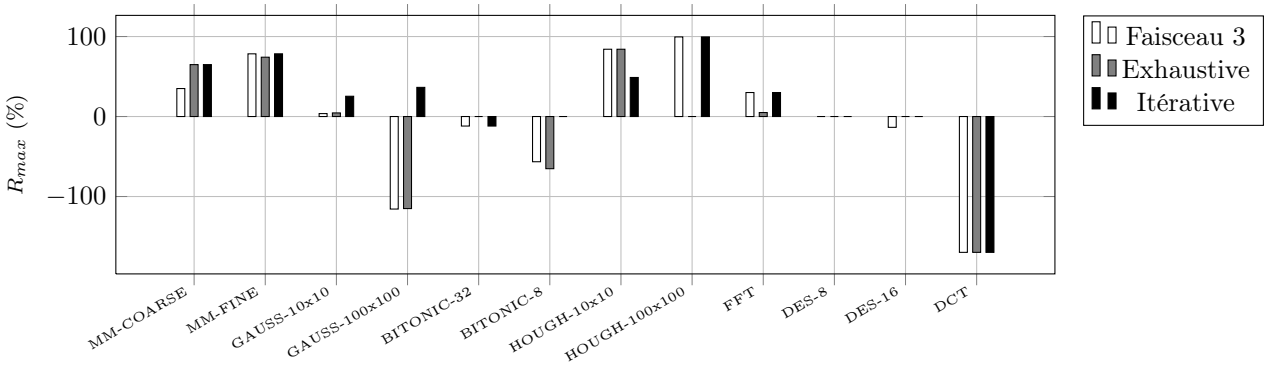


FIGURE 4.22 – Réduction du coût de communication maximum entre deux partitions sur une architecture à forte latence disposant de 4 cœurs pour les différents programmes.

Programme	*COARSE*
MM-COARSE	256
BITONIC-32	4096
FFT	4096
DCT	1024

La fusion des tâches à été désactivée pour le programme DCT, puisque celui-ci utilise un constructeur particulier de StreamIt (`peek`) que nous savons compiler ; mais que nous ne traitons pas encore dans l'algorithme de fusion.

Le code C généré par notre backend à été compilé avec `gcc-4.3 -O3` pour BITONIC-32, FFT et DCT. Pour MM-COARSE, le code généré après fusion des tâches est très long (environ 15000 lignes de code par partition) et `gcc` a du mal à le compiler (la compilation n'avait pas fini après avoir tourné pendant plusieurs heures). C'est pourquoi pour MM-COARSE, nous utilisons le compilateur LLVM avec la commande `clang-1.1 -O2` pour compiler le programme (le temps de compilation étant long mais raisonnable : de l'ordre d'une dizaine de minutes). Bien entendu, dans l'idéal, il faudrait réimplémenter notre algorithme de fusion des tâches de manière à générer un code plus compact (§ 4.1.6).

Le temps d'exécution de chaque programme à été mesuré sur un grand nombre de cycles (1000 pour MM-COARSE, 4000 pour BITONIC-32, 16000 pour FFT et 500 pour DCT). Le temps d'exécution à été mesuré à 5 reprises et nous avons gardé la valeur médiane des 5 exécutions.

Enfin, nous avons mesuré dans les mêmes conditions le temps d'exécution de la version StreamIt du programme sur 1 seul cœur, compilé avec la commande `strc -O3`. Ce temps nous sert de référence : quand on parlera de speedup, on fera référence au rapport entre le temps d'exécution obtenu sur 4 cœurs par notre backend et le temps d'exécution obtenu sur 1 cœur avec StreamIt.

Résultats

Les résultats obtenus sont resumés sur la figure 4.23. Sans transformations notre backend obtient des speedup de ×2.6 pour BITONIC-32, ×2.2 pour FFT et ×1.6 pour DCT. MM-COARSE n'obtient aucun speedup ; la duplication des lignes et des colonnes se fait sur une premier cœur, l'envoi des données aux autres cœurs constitue le goulet d'étranglement pour la performance. Sur une architecture à quatre cœurs on pourrait prétendre à un meilleur speedup.

Après transformations, les speedup obtenus sont bien meilleurs : ×3.5 pour BITONIC-32, ×3.4 pour FFT et ×2.1 pour DCT. Pour MM-COARSE, les étapes de transposition et duplication ont

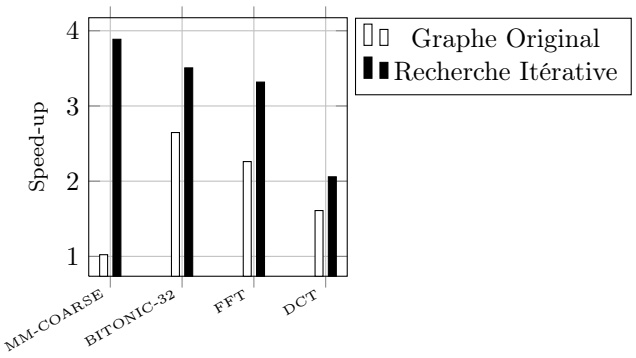


FIGURE 4.23 – Speedup du temps d'execution avec et sans exploration itérative sur 4 cœurs (compilée avec notre backend) par rapport à la version de référence séquentielle (compilée avec StreamIt -O3).

été cassés en plusieurs blocs, ce qui permet à chaque processeur de faire les duplications qui lui sont nécessaires localement, MM-COARSE affiche ainsi, après exploration, un speedup de $\times 3.8$. On peut observer le graphe avant et après transformations avec le partitionnement obtenu sur la figure 4.24.

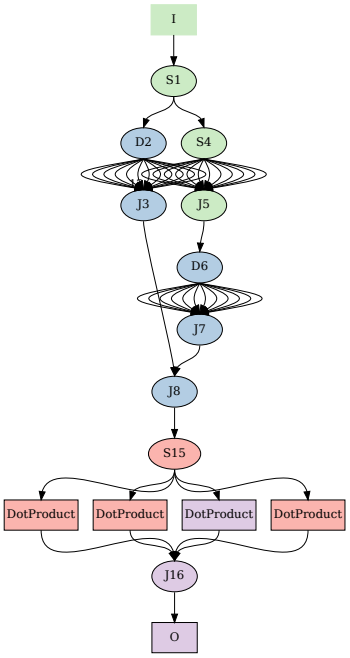
Nous allons regarder de plus près ce qui se passe sur l'exemple de la FFT. En figure 4.25(a) on a représenté le partitionnement obtenu pour la version originale de la FFT, sur la droite on a représenté les communications entre partitions. En figure 4.25(b) on a représenté le partitionnement obtenu par la recherche itérative, comme on peut le voir les communications entre partitions (maximum et cumulées) sont moindres ; ce qui explique le speedup mesuré.

4.5 Travaux connexes

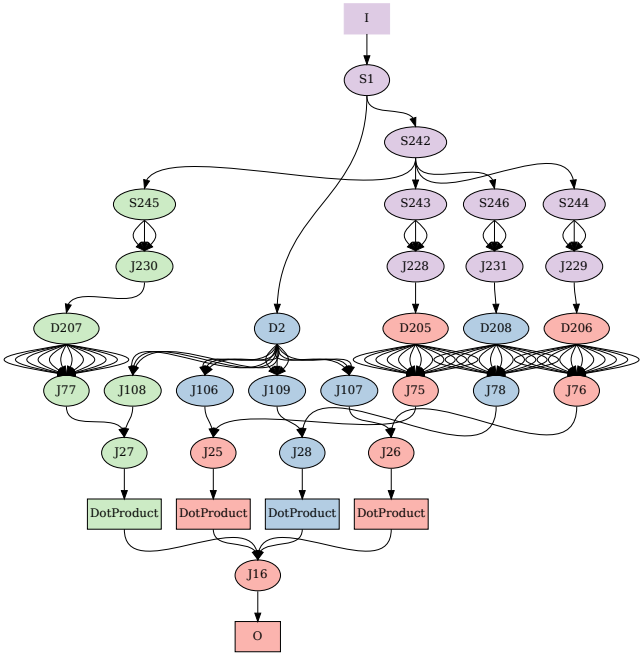
Fusion des tâches et facteur coarse La fusion des tâches permet d'augmenter la granularité des tâches ; puisque la taille du code est plus importante, l'optimiseur a plus de latitude pour trouver des optimisation intéressantes, voir des optimisation inter-nœuds ; enfin dans certains cas, plusieurs opérations successives peuvent être factorisées de manière à réduire les communications entre nœuds d'un même pipeline. De nombreux auteurs [STRA05][CRA09][LDWL06] insistent sur l'importance de la fusion pour maximiser les performances des programmes flots de données. Sermulins[STRA05] propose notamment une méthode de fusion qui tient compte de la taille du cache de la cible, il étudie également l'importance du facteur coarse pour augmenter la taille des échantillons pour les filtres à grain fin.

SGMS La méthode *Stream Graph Modulo Scheduling* SGMS a été proposée dans [KM08] pour ordonnancer des programmes StreamIt sur le processeur Cell BE. La méthode a été étendue par [CLC+09] qui l'a adaptée aux problématiques de l'embarqué, en proposant une extension qui réduit l'empreinte mémoire et le nombre de cœurs utilisés. SGMS a aussi été utilisée pour executer des programmes sur GPU[UGT09a] et GPU+CPU[UGT09b]. SGMS est une méthode qui à l'avantage d'être simple et performante à la fois. Son principal défaut est la non prise en compte des arcs de feedback qui impose une étape préalable de fusion de cycles. Il existe des méthodes d'ordonnancement plus fines [TLA02] qui prennent en compte les cycles.

Dans notre backend, pour conserver les différentes copies des données sur le pipeline logiciel de SGMS nous allouons autant de cases que la différence des étages entre consommateur et producteur plus un, dans [KM08] et [CLC+09] on montre que ce nombre de cases est suffisant pour assurer

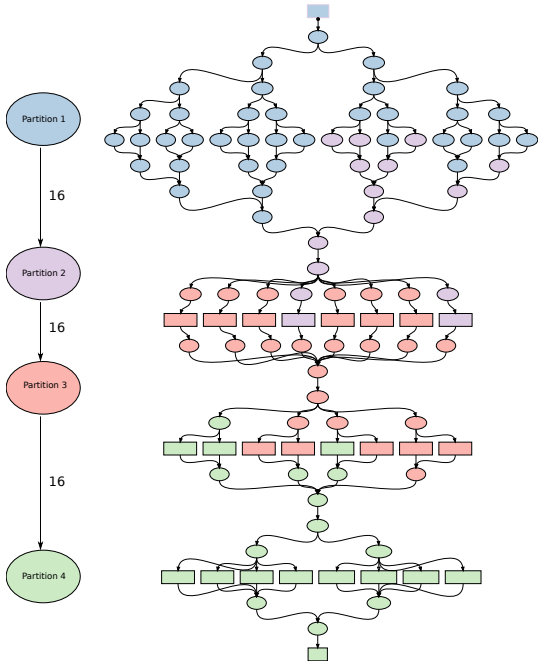


(a) Partitionnement de la MM-COARSE avant partitionnement sur 4 cœurs

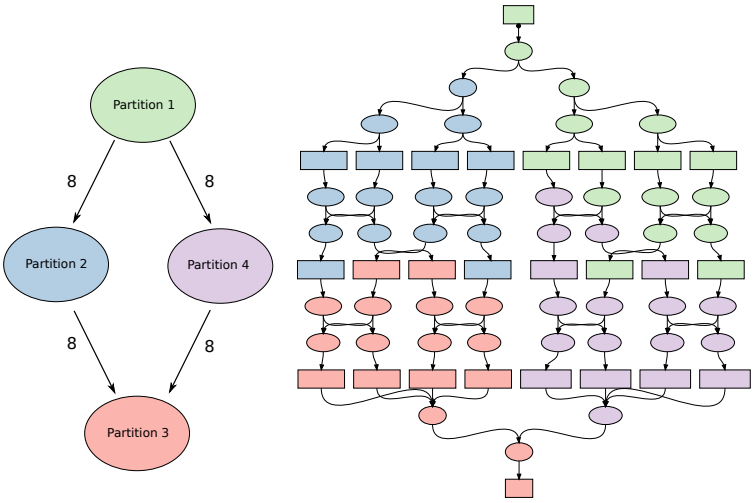


(b) Partitionnement de MM-COARSE après transformations sur 4 cœurs

FIGURE 4.24 – Exploration itérative : amélioration du coût des communications sur le calcul de MM-COARSE. Les duplication sont faites localement en réduisant significativement le nombre d'élements échangés entre les différentes partitions.



(a) Partitionnement de la FFT avant partitionnement sur 4 cœurs



(b) Partitionnement de la FFT après transformations sur 4 cœurs

FIGURE 4.25 – Exploration itérative : amélioration du coût des communications sur le calcul de la FFT. La dérivation choisie à répércutée les étages de réorganisation de données de la FFT entre les étages, en faisant apparaître la structure en papillon sous-jacente.

la coherence des données. Néanmoins des travaux récents montrent qu'il est parfois possible de réduire encore plus ce nombre [CRA10].

Exploration itérative La recherche itérative que l'on réalise peut-être rapprochée de l'algorithme de partitionnement proposé par Carpenter[CRA09]. Dans cette méthode, le compilateur va itérer entre une phase de partitionnement et une phase de transformations (déplacement d'un nœud d'une partition à l'autre, fission ou fusion de nœuds). Le partitionnement de la méthode de Carpenter garantit que les partitions générés seront connexes et convexes (ce qui selon l'auteur permet une fusion des tâches plus efficace et des pipelines plus courts). Par contre l'ensemble de transformations que nous proposons est bien plus riche et permet des transformations profondes des motifs de communication lors de l'exploration.

Optimisation de la mémoire La minimisation de la mémoire dans les graphes de flots de données est un sujet qui a été abondamment étudié. Certaines approches comme celle de Choi[CLC+09], réduisent la mémoire en optimisant le partitionnement. De nombreuses approches cherchent à trouver des ordonnancements[WBS07][SGB08] qui réduisent la mémoire consommée sans pour autant dégrader le débit. Enfin, certaines approches diminuent la mémoire en réutilisant le même tampon pour plusieur nœuds (*buffer sharing*)[MB04][FHHG10]. Nous pensons que l'approche par transformations que nous proposons est complémentaire de ces approches, puisqu'elle transforme directement la manière dont est implémenté le programme. On a d'ailleurs montré que notre approche pouvait être associée avantageusement avec le partitionnement de Choi(§ 4.3.4). Il serait intéressant d'étudier l'association (et les éventuelles interférences) de notre méthode avec les méthodes de partage des tampons.

Optimisation des communications La minimisation des communications entre les différents cœurs est également un problème important. Il a été étudié dans le modèle polyédrique dans [LCL99] où les auteurs proposent un algorithme qui sépare le programme en les composantes fortement connexes du graphe de dépendances et réduit la communication entre chaque partition. Udupa propose dans [UGT09b] une méthode pour cacher le coût des communications entre CPU et GPU pour des programmes StreamIt par la résolution d'un problème linéaire en nombre entiers.

4.6 Conclusion

Nous avons présenté l'architecture du backend SJD que nous avons implémenté. Ce backend transforme un programme SJD en un programme C executable sur un cible multiprocesseur. Puis nous avons étudié deux applications de la méthode d'exploration par transformations présentée dans le chapitre 3.

Tout d'abord, nous nous sommes intéressés au problème de la réduction de l'empreinte mémoire sur un ou plusieurs cœurs. Pour cela nous avons proposé deux métriques ϕ adaptées à ce problème et nous avons mesuré la mémoire consommée avant et après exploration. En monocœur nous obtenons une réduction moyenne de la borne supérieure de la mémoire de -26% sur l'ensemble des benchmarks. En multicœur, nous obtenons pour l'ensemble des benchmarks une réduction moyenne de mémoire de -34% avec le mapping de Choi sur 4 cœurs.

Pour démontrer la versatilité de l'exploration, nous avons étudié une deuxième application : la réduction du coût des communications entre différents cœurs sur des architectures à faible et forte latence. La mesure du volume de communications entre partitions est étroitement lié au partitionnement qui dans notre chaîne de compilation est effectué en aval de l'exploration. Pour lever ce problème nous avons proposé deux solutions : 1) l'utilisation d'une métrique indirecte qui mesure le poids moyen des nœuds de synchronisation et 2) l'utilisation d'une méthode itérative réalisant de multiples aller-retours entre l'exploration et le partitionnement. Toutes architectures

et benchmarks confondus, l'utilisation d'une métrique indirecte réduit en moyenne le coût de communication de 16% contre 35% pour la méthode itérative.

Enfin, nous avons voulu vérifier si cette diminution du coût des communications se traduisait en une augmentation des performances à l'exécution. Pour cela nous avons compilé à l'aide du backend développé, 4 des benchmarks pour lesquels l'exploration itérative réduisait de manière significative le coût en communications. La comparaison des temps d'exécution des versions optimisées par rapport aux versions non transformées a mis en évidence des accelérations significatives sur les 4 benchmarks.

Chapitre 5

Conclusion

Sommaire

5.1 Bilan

Dans le deuxième chapitre de cette thèse nous nous sommes intéressés à l'expression des réorganisations de données dans les langages à flots de données. Nous avons proposé et étudié deux langages. Le premier langage, SJD, est très expressif (il permet d'écrire tout les arrangements avec répétitions des données) mais peu ergonomique : les programmes SJD sont verbeux et peu naturels, surtout pour l'expression de réorganisation de données en plusieurs dimensions. Le deuxième langage, SLICES, est moins expressif mais offre une syntaxe très haut-niveau : le concepteur déclare explicitement les données qu'il souhaite extraire. Nous montrons comment tout programme SLICES peut-être compilé vers un programme SJD équivalent en maîtrisant la complexité du graphe produit. Ces deux langages complémentaires gagnent à être combinés : les réorganisations régulières de données peuvent être écrites succintement en SLICES et SJD peut être utilisé ponctuellement pour l'expression des réorganisations irrégulières.

Dans le troisième chapitre de cette thèse nous nous sommes intéressés aux transformations de programmes SJD. Nous avons défini un modèle formel qui permet de déterminer si une transformation est correcte. Nous nous sommes apuyés sur ce formalisme pour proposer un ensemble de transformations correctes que nous classons en deux grandes catégories : les transformations simplificatrices qui suppriment des arcs ou des nœuds et les transformations restructurantes qui altèrent l'expression du routage des données dans le graphe. Nous montrons que si l'on prive cet ensemble de transformations de la transformation générale de déroulage, l'ensemble des variantes générées est fini. Il est possible de récupérer certaines transformations de déroulage en les combinant avec des transformations simplificatrices tout en conservant la finitude de l'espace engendré. Enfin, deux méthodes pour explorer cet espace sont proposées.

Dans le quatrième et dernier chapitre, nous décrivons la chaîne de compilation de graphes

SJD que nous avons implémentée et qui est construite autour de l'ordonnancement SGMS[KM08]. Nous montrons que l'exploration des transformations de graphes SJD permettent de réduire l'empreinte mémoire de différents programmes pour une execution séquentielle et parallèle. Puis nous appliquons la méthode à un problème différent : la réduction du coût des communications. Nous montrons que les transformations de graphes permettent de réduire significativement les coût de communication multiprocesseurs et que cette réduction a un impact important sur le temps d'exécution.

5.2 Réponses aux problématiques

Combiner l'expressivité d'un langage de haut-niveau avec des optimisations efficaces

Dans le chapitre 2 nous avons montré qu'il est possible de compiler le langage SLICES vers le langage SJD. Nous avons ainsi séparé les problématiques de l'expression et de l'optimisation. L'expression peut être faite dans un langage haut-niveau comme SLICES (proche d'ArrayOL) et l'optimisation peut être faite dans un langage bas-niveau comme SJD (proche de StreamIt). SJD est donc utilisé comme un langage intermédiaire dans la compilation. Cette approche permet de découpler les constructions haut-niveau introduites qui peuvent être très riches (blocs, grilles, itérateurs) et les régles d'optimisation qui sont définies sur un langage minimaliste comme SJD. Pour que cette approche soit efficace il faut s'assurer que les graphes SJD générés à partir de SLICES soient susceptibles d'être optimisés.

Nous avons montré que la complexité de nœuds et d'arêtes des graphes SJD générés à partir de SLICES ne dépendent que de la taille des motifs extraits et du nombre d'itérations sur les nids de boucles imbriqués. Ceci nous garantit dans la plus part des cas que les graphes générés sont de taille raisonable et peuvent être optimisés par la méthode d'exploration.

Dans cette thèse nous avons fait le choix de nous concentrer sur l'optimisation des graphes SJD. Néanmoins il est pertinent de se demander si l'on ne gagnerait pas à réaliser certaines optimisations directement sur le modèle SLICES. On pourrait par exemple calculer l'intersection des blocs extraits par les différents itérateurs d'un programme, de manière à identifier et factoriser les réorganisations similaires.

Etudier les transformations sur les réorganisations de données pour des graphes arbitraires avec des boucles

Pour optimiser les graphes SJD nous avons décidé d'étendre les transformations utilisées par StreamIt aux graphes arbitraires cycliques. Dans le chapitre 3, nous avons proposé un ensemble de transformations sur les graphes SJD arbitraires. La principale difficulté a été de définir formellement une transformation correcte en présence de boucles de retour sur le graphe. Le lemme de localité nous a permis de vérifier facilement la correction de chaque transformation proposée.

L'ensemble de transformations proposé est assez divers : on retrouve ainsi des transformations de suppression, de compaction, de factorisation, de regroupement et de déroulage.

Proposer une méthode d'exploration de l'espace d'implémentation guidée par les contraintes architecturales et par les critères d'optimisation

Dans cette thèse nous avons proposé deux méthodes heuristiques d'exploration de graphes : une première méthode basée sur la recherche par faisceau et une deuxième méthode de recherche itérative. Pour pouvoir évaluer ces méthodes nous avons également étudié une méthode de recherche exhaustive. La recherche exhaustive était envisageable puisque l'on a prouvé que l'espace d'exploration engendré par nos transformations est fini.

Dans le chapitre4 nous avons évalué ces différentes méthodes d'exploration sur deux problèmes importants dans la compilation : la réduction de la mémoire consommée par un programme et la

réduction du coût des communications. Le premier problème traduit une contrainte architecturale : le programme ne peut dépasser la mémoire disponible sur l'architecture. Le deuxième problème est du domaine de l'optimisation : en réduisant les coût de communication il est possible d'accélérer les programmes pour lesquels la communication est le goulet d'étranglement.

5.3 Perspectives

Combiner différents objectifs dans l'exploration

Dans l'évaluation expérimentale de la méthode d'exploration nous n'avons considéré qu'une seule fonction objectif à la fois : réduire la mémoire ou réduire le coût des communications. En réalité dans un processus de compilation industriel ; les concepteurs sont confrontés à un ensemble d'objectifs inter-dépendants : réduire le coût des communications ; augmenter le parallélisme et garantir que la mémoire consommée est inférieure à la mémoire disponible sur l'architecture cible.

Il serait intéressant d'évaluer la méthode d'exploration dans un contexte d'exploration multi-objectifs. Une première possibilité consiste à considérer une fonction objectif ϕ qui combine linéairement, avec des coefficients de pondération, les différents objectifs individuels. Ce n'est pas forcément la meilleure approche : en effet il est inutile de réduire l'empreinte mémoire d'un programme, une fois que l'on est en dessous du seuil de mémoire disponible dans l'architecture.

À notre avis une approche intéressante pourrait être construite à partir de l'approche itérative. On pourrait séparer les critères à optimiser en : (1) contraintes, par exemple, l'empreinte mémoire maximum par processeur ou le nombre de cœurs disponibles, et (2) en objectifs, par exemple réduire les communications, augmenter le parallélisme déployé, réduire la consommation énergétique. Les objectifs seraient optimisés au fur et à mesure des itérations partitionnement/transformations et les programmes qui ne satisfont pas les contraintes seraient rejettés. Bien entendu, ce n'est pas ausssi simple, puisque l'ordonnancement, le partitionnement et les transformations qui améliorent un des objectifs peuvent en empirer un autre.

Étendre les transformations à un modèles SJD avec contrôle

Dans cette thèse nous avons fait le choix de nous restreindre à un modèle CSDF sans contrôle (SJD). Néanmoins il existe des modèles de flots qui prennent en compte le contrôle, par exemple Buck[Buc93] propose un modèle où des joiners (appelés *select*) et splitters (appelés *switch*) sont commandés par un flux de prédicats qui déterminent si c'est la valeur gauche ou la valeur droite qui sera produite en sortie.

Il serait intéressant de considérer quelles transformations sont possibles dans un modélè DDF. Par exemple, une transformation qui nous semble intéressante est schématisé en figure 5.1. Le graphe initial utilise les nœuds prédicatés de Buck ; si le flux de prédicats P_1 contient une valeur vraie alors la branche gauche sera executée, dans le cas contraire ce sera la branche droite. On pourrait décider de déplacer les nœuds *switch* le long des branches prédicatées comme sur l'exemple. En fait cela revient à rendre l'execution des nœuds A_1 et B_1 spéculative ; ce qui permet de faire les calculs correspondants avant de connaître la valeur du prédicat. Une fois la valeur du prédicat connue, les nœuds *switch* intermédiaires selectionneront la production choisie (par exemple celle de la branche A) et élimineront la production non choisie (par exemple en redirigeant la sortie de la branche B vers le puit T). Cette transformation fait apparaître du parallélisme spéculatif ; elle permet de commencer à exécuter les branches A et B avant de savoir laquelle sera choisie.

Cette transformation n'est qu'un exemple ; nous pensons qu'il peut-être très intéressant de s'intéresser aux transformations de graphes DDF. Bien entendu, il faudrait étendre la définition de transformation correcte pour prendre en compte l'exécution prédicatée des DDF.

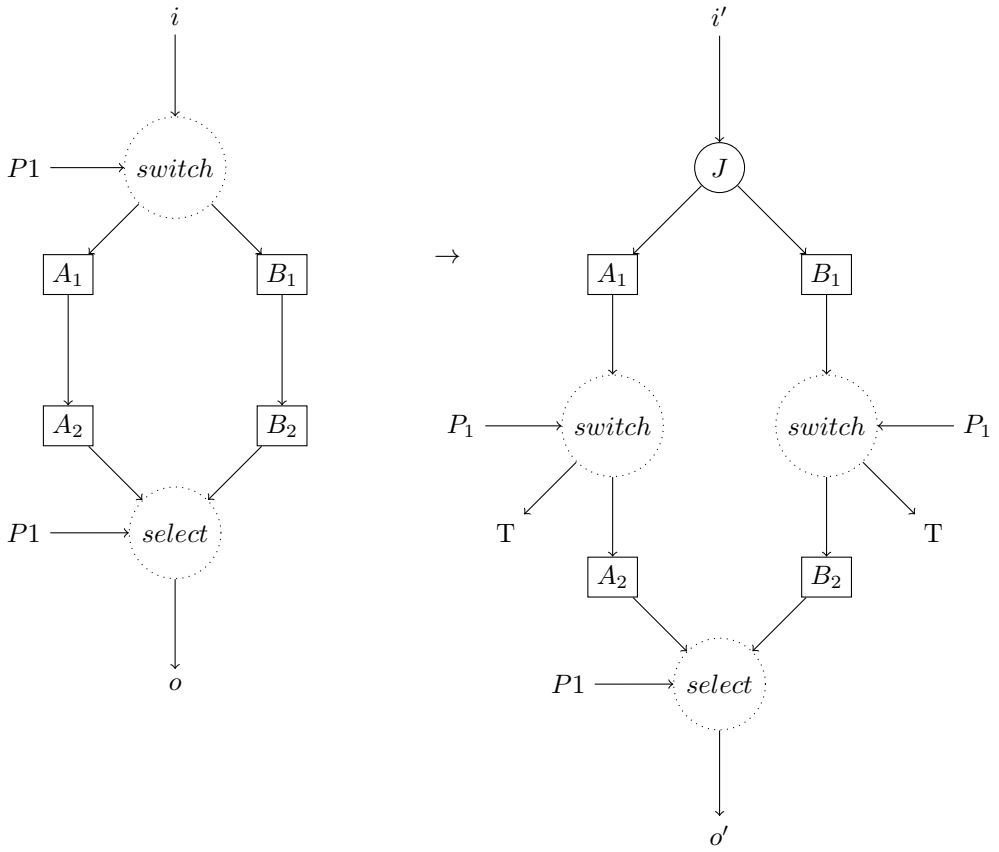


FIGURE 5.1 – Exemple de transformation possible sur le modèle DDF : on fait apparaître du parallélisme spéculatif.

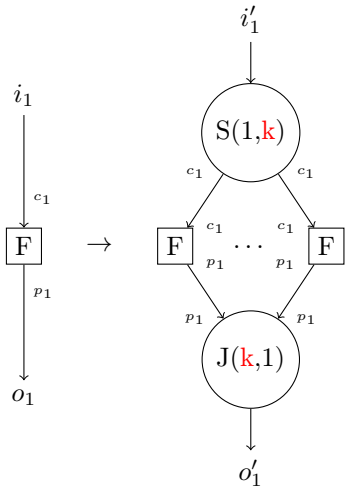


FIGURE 5.2 – Exemple de transformation paramétrique.

Considérer des transformations sur des groupes de nœuds

Les transformations présentées dans le chapitre chapitre 3 considèrent toujours un nœud et ses enfants ou un nœud et ses parents. On pourrait imaginer des transformations plus complexes prennant en compte un ensemble de nœuds plus important. Nous pensons que cela permettrait de faire des transformations plus importantes qui réduiraient la complexité de l'exploration. Par exemple supposons l'existence d'un pipeline formé de deux sous-graphes des nœuds `Split` et `Join` ; si le premier groupe de nœuds travaille avec une granularité multiple des consommations du deuxième groupe. Alors il est possible de permuter les deux groupes sur le pipeline. Même si on peut arriver au même résultat par des transformations pas à pas sur des paires de nœuds, cela est plus long et complique l'exploration. Nous pensons qu'il peut être intéressant de définir des transformations sur des groupes de nœuds plus importants de manière à explorer le graphe à différents niveau de granularité : par exemple, en essayant d'abord les transformations importantes, et ensuite raffiner le candidat choisi par des petites transformations.

Transformations paramétriques

Un des inconvénients dans le système de transformations proposé est l'obligation de choisir des paramètres pour certaines transformations (*SplitF*, *ReorderS*, etc.). En effet chaque paramètre exploré va créer une nouvelle branche dans le graphe d'exploration. On pourrait imaginer un système d'exploration qui travaille sur des graphes paramétriques. Par exemple *SplitF* remplacerait un nœud `Filter` par k copies avec k étant un paramètre libre comme sur la figure 5.2. Pour que cela fonctionne il faudrait redéfinir l'ensemble des transformations de manière à ce qu'elles puissent s'appliquer sur des graphes dont certaines caractéristiques dépendent d'un paramètre libre k.

Puisque certaines transformations imposent des conditions sur la divisibilité du nombre d'arêtes ; au fur et à mesure des transformations le paramètre k serait contraint. Il faudrait bien entendu propager ces contraintes sur l'arbre d'exploration. Une fois un graphe candidat choisi, les libertés restantes sur les paramètres du graphe seraient fixés lors du partitionnement ou de l'ordonnancement de manière optimale.

Le principal avantage de cette proposition est de réduire la complexité de l'exploration ; au lieu de devoir tester plusieurs paramètres pour chaque transformation paramétrée ; on fait une seule fois la transformation et les paramètres sont contraints au fur et à mesure de l'exploration.

Bien entendu cette proposition complique considérablement l'implémentation de l'exploration qui doit tenir compte de paramètres variables à différents endroits de la chaîne de traîtement.

SLICES : déclaration des libertés par le concepteur

Le modèle SLICES considere que les patrons extraits doivent être produits dans un ordre précis : ceci est d'ailleurs souvent le cas, par exemple pour le calcul du filtre gaussien, les noyaux 3×3 doivent être extraits dans l'ordre de l'image produite. Néanmoins dans certains cas l'ordre de production peut être altéré sans changer la sémantique, par exemple lorsque l'on calcule une réduction sur un ensemble de données avec une opération associative et/ou commutative [Pot98]. Il serait intéressant de permettre au concepteur d'exprimer les libertés d'ordre sur les éléments des flux. Cela permettrait par exemple pour les flux dont l'ordre n'est pas important de relacher l'ordonnancement. Cela nécessiterait peut-être d'introduire de nouveaux nœuds SJD : par exemple, un nœud `Join` qui n'impose pas d'ordre sur les arrivées de données sur les canaux de réception.

Bibliographie

[ABC+06] K. Asanovic, R. Bodik, B.C. Catanzaro, J.J. Gebis, P. Husbands, K. Keutzer, D.A. Patterson, W.L. Plishker, J. Shalf, S.W. Williams, et al. The landscape of parallel computing research : A view from berkeley. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2006-183*, pages 2006–183, 2006.

[ABD05] A. Amar, P. Boulet, and P. Dumont. Projection of the Array-OL specification language onto the Kahn process network computation model. In *Parallel Architectures, Algorithms and Networks, 2005. ISPAN 2005. Proceedings. 8th International Symposium on*, page 6, 2005.

[AEH+99] Marc Andries, Gregor Engels, Annegret Habel, Berthold Hoffmann, Hans-Jörg Kreowski, Sabine Kuske, Detlef Plump, Andy Schürr, and Gabriele Taentzer. Graph Transformation for Specification and Programming. *Sci. Comput. Program.*, 34(1) :1–54, 1999.

[AGK+05] S. Amarasinghe, M. Gordon, M. Karczmarek, J. Lin, D. Maze, R.M. Rabbah, and W. Thies. Language and compiler design for streaming applications. *International Journal of Parallel Programming*, 33(2) :261–278, 2005.

[AW77] E. A. Ashcroft and W. W. Wadge. Lucid, a nonprocedural language with iteration. *Commun. ACM*, 20(7) :519–526, 1977.

[BCE+03] A. Benveniste, P. Caspi, S.A. Edwards, N. Halbwachs, P. Le Guernic, and R. De Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1) :64–83, 2003.

[BELP95] G. Bilsen, M. Engels, R. Lauwereins, and JA Peperstraete. Cyclo-static data flow. In *Acoustics, Speech, and Signal Processing, 1995. ICASSP-95., 1995 International Conference on*, volume 5, 1995.

[BHLM94] J. Buck, S. Ha, E.A. Lee, and D.G. Messerschmitt. Ptolemy : A framework for simulating and prototyping heterogeneous systems. *International Journal of Computer Simulation*, 4(2) :155–182, 1994.

[BML97] S.S. Bhattacharyya, P.K. Murthy, and E.A. Lee. APGAN and RPMC : Complementary heuristics for translating DSP block diagrams into efficient software implementations. *Design Automation for Embedded Systems*, 2(1) :33–60, 1997.

[Bou07] P. Boulet. Array-OL revisited, multidimensional intensive signal processing specification. *Research Report RR-6113, INRIA*, 2007.

[BS08] D. Black-Schaffer. *Block Parallel Programming for Real-Time Applications on Multi-core Processors*. PhD thesis, Stanford University, 2008.

[Buc93] J.T. Buck. *Scheduling dynamic dataflow graphs with bounded memory using the token flow model*. PhD thesis, University of California at Berkeley, 1993.

[CGH+08] A. Charfi, A. Gamatié, A. Honoré, J.L. Dekeyser, and M. Abid. Validation de modèles dans un cadre d'IDM dédié à la conception de systèmes sur puce. *4èmes Jounées sur l'Ingénierie Dirigée par les Modèles (IDM 08), Mulhouse, France*, 2008.

[CLC+09] Yoonseo Choi, Yuan Lin, Nathan Chong, Scott Mahlke, and 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.

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In *STOC '71 : Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158, New York, NY, USA, 1971. ACM.

[CPRB92] A. Chandrakasan, M. Potkonjak, J. Rabaey, and R. Brodersen. "An Approach For Power Minimization Using Transformations". In *VLSI Signal Processing*, pages 41–50, 1992.

[CRA09] Paul M. Carpenter, Alex Ramirez, and Eduard Ayguade. Mapping stream programs onto heterogeneous multiprocessor systems. In *CASES '09 : Proceedings of the 2009 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 57–66, New York, NY, USA, 2009. ACM.

[CRA10] P. Carpenter, A. Ramirez, and E. Ayguade. Buffer sizing for self-timed stream programs on heterogeneous distributed memory multiprocessors. *High Performance Embedded Architectures and Compilers*, pages 96–110, 2010.

[CT65] J.W. Cooley and J.W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of computation*, 19(90) :297–301, 1965.

[DBDM02] F. Devin, P. Boulet, J.L. Dekeyser, and P. Marquet. GASPARD : a visual parallel programming environment. In *Proceedings of International Conference on Parallel Computing in Electrical Engineering, 2002. PARELEC'02.*, pages 145–150, 2002.

[DCS04] Steven J. Deitz, Bradford L. Chamberlain, and Lawrence Snyder. Abstractions for dynamic data distribution. In *Proc. of the Workshop on High-Level Parallel Programming Models and Supportive Environments*, 2004.

[DH72] R.O. Duda and P.E. Hart. Use of the Hough transformation to detect lines and curves in pictures. *Communications of the ACM*, 15(1) :11–15, 1972.

[DM79] Nachum Dershowitz and Zohar Manna. Proving Termination with Multiset Orderings. *Comm. ACM*, 22(8), 1979.

[dOCLB09] P. de Oliveira Castro, S. Louise, and D. Barthou. Design-Space Exploration of Stream Programs through Semantic-Preserving Transformations. Technical report, hal-00447376, 2009.

[dOCLB10a] P. de Oliveira Castro, S. Louise, and D. Barthou. A Multidimensional Array Slicing DSL for Stream Programming. In *2010 International Conference on Complex, Intelligent and Software Intensive Systems*, pages 913–918. IEEE, 2010.

[dOCLB10b] P. de Oliveira Castro, S. Louise, and D. Barthou. Reducing memory requirements of stream programs by graph transformations. In *High Performance Computing and Simulation (HPCS), 2010 International Conference on*, pages 171–180. IEEE, 2010.

[dOCLB10c] Pablo de Oliveira Castro, Stéphane Louise, and Denis Barthou. Automatic mapping of stream programs on multicore architectures. Présentation au 15th Workshop on Compilers for Parallel Computing, Vienna, 2010.

[Dum05] P. Dumont. *Spécification Multidimensionnelle pour le traitement du signal systématique*. PhD thesis, Université des sciences et technologies de Lille, 2005.

[ET06] S.A. Edwards and O. Tardieu. SHIM : A deterministic model for heterogeneous embedded systems. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 14(8) :854–867, 2006.

[EVT08] S.A. Edwards, N. Vasudevan, and O. Tardieu. Programming shared memory multiprocessors with deterministic message-passing concurrency : Compiling SHIM to Pthreads. In *Design, Automation and Test in Europe, 2008. DATE'08*, pages 1498–1503. IEEE, 2008.

[Fea92a] P. Feautrier. Some efficient solutions to the affine scheduling problem. Part I : One-dimensional time. *International journal of parallel programming*, 21(5) :313–347, 1992.

[Fea92b] P. Feautrier. Some efficient solutions to the affine scheduling problem. Part II : Multidimensional time. *International journal of parallel programming*, 21(5) :389–420, 1992.

[FHHG10] M.H. Foroozannejad, M. Hashemi, T.L. Hodges, and S. Ghiasi. Look into details : the benefits of fine-grain streaming buffer analysis. *ACM SIGPLAN Notices*, 45(4) :27–36, 2010.

[GBL+08] Thierry Goubier, Frédéric Blanc, Stéphane Louise, Renaud Sirdey, and Vincent David. Définition du Langage de Programmation ΣC, RT LIST DTSI/SARC/08-466/TG. Technical report, CEA Saclay, France, 2008.

[Gro] StreamIt Group. Streamit benchmarks. `http://groups.csail.mit.edu/cag/streamit/shtml/benchmarks.shtml`.

[GTK+02] Michael I. Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S. Meli, Andrew A. Lamb, Chris Leger, Jeremy Wong, Henry Hoffmann, David Maze, and 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.

[Hal98] N. Halbwachs. Synchronous programming of reactive systems. In *Computer Aided Verification*, pages 1–16. Springer, 1998.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8) :666–677, 1978.

[Hoc94] Roger W. Hockney. The communication challenge for mpp : Intel paragon and meiko cs-2. *Parallel Comput.*, 20(3) :389–398, 1994.

[JS05] A. Jantsch and I. Sander. Models of computation and languages for embedded system design. *IEE Proceedings-Computers and Digital Techniques*, 152(2) :114–129, 2005.

[Kah74] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information processing*, pages 471–475, Stockholm, Sweden, Aug 1974. North Holland, Amsterdam.

[KDK+01] B. Khailany, W.J. Dally, U.J. Kapasi, P. Mattson, J. Namkoong, J.D. Owens, B. Towles, A. Chang, and S. Rixner. Imagine : Media processing with streams. *IEEE micro*, 21(2) :35–46, 2001.

[KK98] George Karypis and Vipin Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM J. Sci. Comput.*, 20(1) :359–392, 1998.

[KM66] R.M. Karp and R.E. Miller. Properties of a model for parallel computations : Determinancy, termination, queueing. *SIAM Journal on Applied Mathematics*, pages 1390–1411, 1966.

[KM08] Manjunath Kudlur and Scott Mahlke. Orchestrating the Execution of Stream Programs on Multicore Platforms. In *Proc. of the SIGPLAN conf. on Programming Language Design and Implementation*, pages 114–124. ACM, 2008.

[LCL99] A.W. Lim, G.I. Cheong, and M.S. Lam. An affine partitioning algorithm to maximize parallelism and minimize communication. In *Proceedings of the 13th international conference on Supercomputing*, pages 228–237. ACM, 1999.

[LDWL06] Shih-wei Liao, Zhaohui Du, Gansha Wu, and 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.

[Lee06] EA Lee. The problem with threads. *Computer*, 39(5) :33–42, 2006.

[LL97] A.W. Lim and M.S. Lam. Maximizing parallelism and minimizing synchronization with affine transforms. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, page 214. ACM, 1997.

[LM87] EA Lee and DG Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9) :1235–1245, 1987.

[Low76] Bruce T. Lowerre. *The Harpy Speech Recognition System.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1976.

[LP95] EA Lee and TM Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5) :773–801, 1995.

[MAB+10] H. Munk, E. Ayguadé, C. Bastoul, P. Carpenter, Z. Chamski, A. Cohen, M. Cornero, P. Dumont, M. Duranton, M. Fellahi, et al. ACOTES Project : Advanced Compiler Technologies for Embedded Streaming. *International Journal of Parallel Programming*, pages 1–54, 2010.

[Mat96] The MathWorks, Inc. *MATLAB, Language Reference Manual v5*, 1996.

[MB04] P.K. Murthy and S.S. Bhattacharyya. Buffer merging—a powerful technique for reducing memory requirements of synchronous dataflow specifications. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 9(2) :212–237, 2004.

[MGAK03] W.R. Mark, R.S. Glanville, K. Akeley, and M.J. Kilgard. Cg : A system for programming graphics hardware in a C-like language. In *ACM SIGGRAPH 2003 Papers*, page 907. ACM, 2003.

[Mil80] R. Milner. A calculus of communicating systems. *Lecture Notes in Computer Science*, 92, 1980.

[Moo75] G.E. Moore. Progress in digital integrated electronics. In *Electron Devices Meeting, 1975 International*, volume 21, pages 11 – 13, 1975.

[MPW92] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, i. *Information and Computation*, 100(1) :1–40, 1992.

[MSA+85] J. McGraw, S. Skwdzielewsli, S. Allan, D. Grit, R. Oldehoeft, J. Glauert, I. Dobes, and P. Hohensee. SISAL : streams and iteration in a single assignment language-version 1.2. *Language reference manual, Lawrence Livermore National Laboratory*, 1985.

[OH05] K. Olukotun and L. Hammond. The future of microprocessors. *Queue*, 3(7) :29, 2005.

[Por08] T.E. Portegys. General Graph Identification With Hashing. *Normal, IL, Illinois State University*, 2008.

[Pot98] W.M. Pottenger. The role of associativity and commutativity in the detection and transformation of loop-level parallelism. In *Proceedings of the 12th international conference on Supercomputing*, page 195. ACM, 1998.

[PPL95] T. M. Parks, J. L. Pino, and Edward A. Lee. “A Comparison of Synchronous and Cycle-Static Dataflow". In *Asilomar Conf. on Signals, Systems and Computers*, 1995.

[PS92] P. Panangaden and V. Shanbhogue. The expressive power of indeterminate dataflow primitives. *Information and Computation*, 98(1) :99–131, 1992.

[QR00] F. Quilleré and S. Rajopadhye. Optimizing memory usage in the polyhedral model. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(5) :773–815, 2000.

[Rau94] B.R. Rau. Iterative modulo scheduling : An algorithm for software pipelining loops. In *Proceedings of the 27th annual international symposium on Microarchitecture*, pages 63–74. ACM, 1994.

[RC77] R.C. Read and D.G. Corneil. The graph isomorphism disease. *Journal of Graph Theory*, 1(4) :339–363, 1977.

[RCHP91] J.M. Rabaey, C. Chu, P. Hoang, and M. Potkonjak. Fast prototyping of datapath-intensive architectures. *Design Test of Computers, IEEE*, 8(2) :40 –51, jun. 1991.

[Sch03] Sven-Bodo Scholz. Single Assignment C : Efficient Support for High-Level Array Operations in a Functional Setting. In *J. Funct. Program.*, 2003.

[Sco72] D. Scott. Continuous lattices. *Toposes, algebraic geometry and logic*, pages 97–136, 1972.

[SGB08] Sander Stuijk, Marc Geilen, and Twan Basten. “Throughput-Buffering Trade-Off Exploration for Cyclo-Static and Synchronous Dataflow Graphs". *IEEE Trans. Comput.*, 57(10) :1331–1345, 2008.

[Sou01] J. Soula. *Principe de Compilation d'un Langage de Traitement de Signal*. PhD thesis, Université des sciences et technologies de Lille, 2001.

[STRA05] J. Sermulins, W. Thies, R. Rabbah, and S. Amarasinghe. Cache aware optimization of stream programs. *ACM SIGPLAN Notices*, 40(7) :126, 2005.

[Thi09] W. Thies. *Language and Compiler Support for Stream Programs.* PhD thesis, Massachussetts Institute of Technology, 2009.

[TLA02] W. Thies, J. Lin, and S. Amarasinghe. Phased Computation Graphs in the Polyhedral Model. Technical report, Massachussetts Institute of Technology, 2002.

[UGT09a] A. Udupa, R. Govindarajan, and M.J. Thazhuthaveetil. Software pipelined execution of stream programs on GPUs. In *Proceedings of the 2009 International Symposium on Code Generation and Optimization*, pages 200–209. IEEE Computer Society, 2009.

[UGT09b] Abhishek Udupa, R. Govindarajan, and Matthew J. Thazhuthaveetil. Synergistic execution of stream programs on multicores with accelerators. In *Proc. of the 2009 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 99–108. ACM, 2009.

[VBI08] Ajay K. Verma, Philip Brisk, and Paolo Ienne. "Data-Flow Transformations to Maximize the Use of Carry-Save Representation in Arithmetic Circuits". *IEEE Trans. on CAD of Integrated Circuits and Systems*, 27(10) :1761–1774, 2008.

[VD08] V. Volkov and J.W. Demmel. Benchmarking GPUs to tune dense linear algebra. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–11. IEEE Press, 2008.

[vR95] Guido van Rossum. *Python Reference Manual.* CWI Report, 1995.

[WBS07] Maarten Wiggers, Marco Bekooij, and Gerard Smit. "Efficient Computation of Buffer Capacities for Cyclo-Static Dataflow Graphs". In *Proc. Int. Conf. on Design Automation*, pages 658–663. ACM, 2007.