RenPar'18 / SympA'2008 / CFSE'6
Fribourg, Suisse, du 11 au 13 février 2008

Programmation par squelettes algorithmiques pour le processeur CELL

Joel Falcou, Tarik Saidani, Lionel Lacassagne et Daniel Etiemble

Résumé Le processeur CELL est un exemple typique d'architecture multiprocesseurs hétérogènes sur puce, utilisant plusieurs niveaux de parallélisme pour obtenir des performances très élevées. Le défi pour les développeurs d'outils ou d'applications est alors de réduire l'écart entre les performances crêtes et les performances effectives. Sur des architectures plus classiques, des modèles comme les squelettes algorithmiques proposent une méthodologie simple et *scalable* pour la parallélisation d'applications complexes et ont donné naissance à des outils efficaces. Nous étudions dans cet article l'application du modèle des squelettes algorithmiques à la parallélisation d'applications de traitement d'images sur le processeur CELL. Nous proposons en outre une implantation efficace de ce modèle sous la forme d'un langage orienté domaine méta-programmé au sein de C++. Des résultats préliminaires sont donnés et démontrent la viabilité de l'approche.

1. Introduction

Les applications de traitement d'images sont généralement composées d'une suite d'opérateurs basiques de type point à point ou noyau de convolution dont la complexité tant en calcul qu'en accès mémoire empêche leur exécution temps-réel. Les différentes architectures à base de processeurs multi-cœurs ont permis de répondre à la demande toujours croissante de puissance que les systèmes classiques ne pouvaient satisfaire. Le processeur CELL[1] est un bon exemple d'une telle architecture permettant d'obtenir des performances très élevées en proposant plusieurs niveaux de parallélisme. Néanmoins, de nombreux écueils, comme la nécessité de programmer manuellement les transferts DMA[1] entre cœurs et avec la mémoire principale, rendent ardu le développement et la mise au point d'applications sur cette cible. Il est donc nécessaire de proposer des outils de déploiement qui simplifient ce processus.

De tels outils et méthodologies, appliqués à des architectures plus traditionnelles, ont fait l'objet d'une longue maturation et ont conduit à des résultats appréciables tant sur des architectures de type *clusters* que pour les machines SMP. Il n'existe par contre que très peu d'outils de ce type pour le CELL et la plupart d'entre eux ne considère qu'un modèle de déploiement de type SPMD[2] dans lequel peu d'opérateurs sont mis en œuvre, simplifiant d'autant les problèmes de gestion des synchronisations et des transferts DMA. On parle alors de **Stream processing**, et l'on retrouve dans cette catégorie des outils tels que Sequoia [2], Rapidmind [3], et UPC [4].

Dans cet article, nous étudions l'utilisation des squelettes algorithmiques [5] comme modèle de programmation pour le processeur CELL. Nos principales contributions sont :

- une formalisation des stratégies de parallélisation sur le CELL ;
- la définition d'un *Domain Specific Langage* implantant ces stratégies sous forme d'une bibliothèque C++et son adaptation aux spécificités du CELL ;
- une démonstration de l'efficacité de cette approche.

[1] Direct Memory Access
[2] Single Program Multiple Data

Le plan adopté est le suivant. La section 2 résume rapidement l'architecture du processeur CELL et nos travaux antérieurs sur la parallélisation d'applications. La section 3 présente le *Domain Specific Language* développé à partir de ces travaux et comment celui-ci est implanté directement au sein de C++. La section 4 rassemble les résultats préliminaires évaluant l'efficacité de notre approche. La section 5 conclue cet article et propose quelques développement futurs.

2. Le processeur CELL

Le processeur CELL (fig. 1) est une architecture multi-cœurs hétérogènes sur puce constituée d'une unité de calcul principale 64-bit (PPE), de huit unités spécialisées appelées *Synergistic Processing Elements* (SPE) [6], de divers interfaces d'entrées/sorties et d'un bus à grande bande passante – le *Element Interconnect Bus* (EIB) – qui permet la communication entre les différents composants du CELL. Sur la base d'une fréquence d'horloge de 3.2 GHz, le CELL fournit une puissance crête théorique de 204.8 GFlops/s en calcul flottant simple précision. Le EIB est quant à lui composé de quatre anneaux 128 bits, chacun pouvant gérer jusque à trois transferts concurrents. La bande-passante théorique de ce bus est de l'ordre de 200 Go/s pour les transferts internes (en effectuant huit transferts simultanés sans collision à 25Go/s).

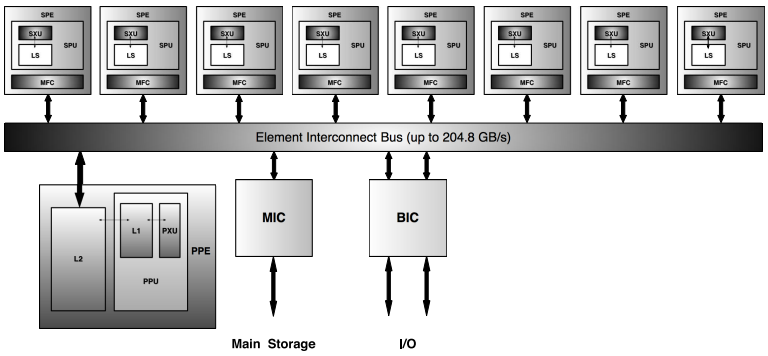

Fig. 1. Architecture du processeur CELL BE d'IBM

Le PPE est un PowerPC 64bits classique équipé de l'unité de calcul vectorielle multimédia Altivec. Cette unité s'occupe principalement de coordonner les SPEs et d'exécuter le système d'exploitation. Chaque SPE est composé d'un unité synergique (SPU) intégrant une unité vectorielle proche d'Altivec et d'un contrôleur mémoire – le *Memory Flow Controller* (MFC). Les SPEs sont équipés d'une mémoire local (ou *Local Store* de 256 Ko. Le MFC est constitué d'un contrôleur DMA 1D effectuant les transferts entre local store ou avec la mémoire principale.

2.1. Modèles de programmation pour le CELL

Dans [7], nous avons exploré plusieurs modèles de parallélisation et évalué l'influence des transferts DMA sur les performances de nos applications. Les résultats obtenus montrent que l'écriture d'un code performant passe par l'utilisation de modèle comme le modèle SPMD, le pipeline ou une combinaison de ceux-ci (fig. 2). Les mesures de performances effectuées sur une application simple mais représentative (un détecteur de points d'intérêt de Harris) ont montrés que l'optimisation du nombre et de la quantité des transferts entre SPEs et entre SPE et le PPE influent grandement sur celle-ci (fig. 3).

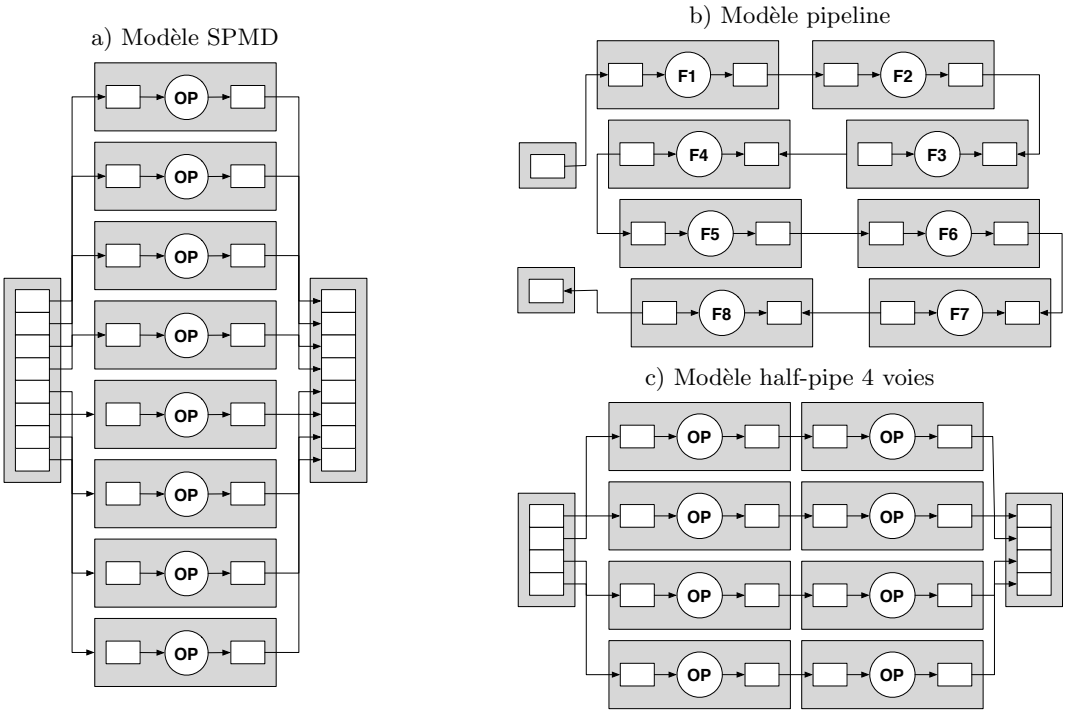

Fig. 2. Modèles de parallélisation envisageables sur le processeur CELL

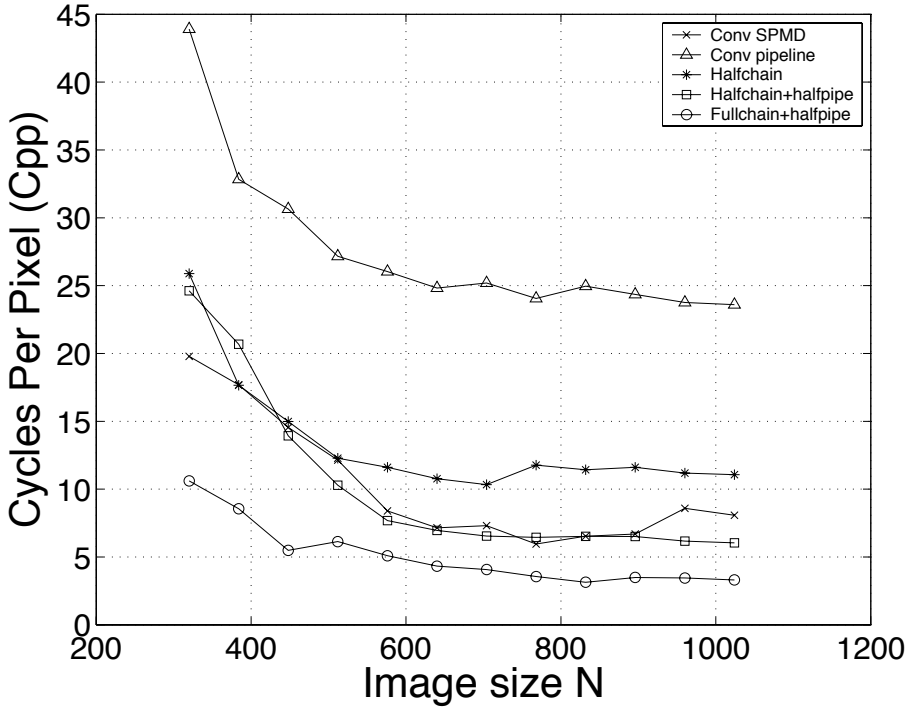

Fig. 3. Performances de 5 stratégies de parallélisation pour l'application Harris

2.2. Du *Stream Processing* aux Squelettes Algorithmiques

Actuellement, la programmation parallèle sur le processeur CELL est encore basée sur l'utilisation d'outils de bas niveau limités, difficiles à manier et sources d'erreurs. Les *Squelettes Algorithmiques*[5,8] se proposent de résoudre ce problème. Un squelette encapsule un schéma de parallélisation récurrent pour lequel une ou plusieurs implémentations efficaces sont connues. De tels squelettes se présentent classiquement comme des fonctions d'ordres supérieurs qui doivent être instanciées avec les fonctions spécifiques de l'application. Avec cette approche, le travail du programmeur se limite à choisir et à combiner un ensemble de squelettes pris dans une base prédéfinie, sans qu'il ait à se préoccuper des détails de la mise en œuvre du parallélisme sur l'architecture considérée.

L'ensemble des squelettes classiquement proposés dans la littérature nous apparait comme trop large pour être appliqués directement sur le CELL. Nous nous proposons donc de nous limiter aux schémas mis en avant précédemment et de proposer un modèle d'exécution inspiré du modèle *Stream Processing* [2] :

- les données et les résultats sont encapsulées par un type de données abstrait et stockées dans la mémoire principale ;
- les données sont découpées en bandes appelées *tuiles*. Ces tuiles constituent l'élément de base du flux de données transmis au SPEs par une série de transferts DMA asynchrones ;
- les opérateurs sont déployés sur les SPEs via une combinaison de squelettes algorithmiques. Ces opérateurs sont appliqués à une ou plusieurs tuiles afin de générer une ou plusieurs tuiles résultats ;
- les résultats sont transférés des SPEs vers la mémoire principale par transferts DMA asynchrones.

A partir de ce modèle, nous nous proposons de définir un langage permettant de définir de telles applications.

3. Un *Domain Specific Language* pour les squelettes algorithmiques

Le développement d'un outil efficace de programmation par squelettes algorithmiques est une tâche complexe. Plusieurs essais [8,9] montrent que le compromis entre expressivité et efficacité est un point crucial. Dans [10], nous décrivons QUAFF qui est une tentative pour résoudre cette tension en utilisant un système d'**évaluation partielle méta-programmée**. QUAFF est une bibliothèque C++ à base de méta-programmes *templates* qui évaluent **à la compilation** la structure d'une application parallèle décrite sous forme de squelettes. En pré-calculant le placement et les schémas de communications à la compilation, le temps d'exécution du code généré par QUAFF est très proche de celui d'un code C déployé manuellement. Au vu de ces résultats, nous avons décidé d'appliquer cette technique à la définition d'une bibliothèque de programmation parallèle par squelettes pour le processeur CELL. Grâce à la flexibilité de QUAFF, nous pouvons reformuler ces règles de production afin de prendre en compte les spécificités des modèles de parallélisation mis en avant. Cette bibliothèque – appelée SKELL BE [3] – est décrite ci-après.

[3] Skeletons for the CELL BE

3.1. Spécification du langage

Une application Skell BE se définit via le langage suivant :

$$
\begin{aligned}
\mathcal{A} &::= \mathsf{run}\ \Sigma \\
\Sigma &::= \Gamma \mid \mathsf{Farm}\ n\ \Gamma \\
\Gamma &::= \mathsf{Seq}\ f \mid \mathsf{Pipe}\ \Gamma_1 \ldots \Gamma_n \\
f &::= \textit{fonctions C++ définies par l'utilisateur} \\
n &::= \textit{entier} \geq 1
\end{aligned}
$$

Intuitivement, Seq transforme une fonction utilisateur en un élément utilisable au sein des autres squelettes ; Pipe encapsule un parallélisme de type flux : n étapes de calcul sont enchainées sur les données d'entrée, chaque étape pouvant s'exécuter en parallèle sur des données distinctes ; Farm encapsule un parallélisme de type données : une même opération est appliquée à l'ensemble des données d'entrée. Une application est finalement définie par l'appel de run sur un squelette Σ.

La structure parallèle de l'application est donc finalement représentée par un arbre dont les nœuds correspondent aux squelettes et les feuilles aux fonctions utilisateurs. Par exemple, considérons l'opération d' *ouverture* morphologique qui consiste à appliquer successivement à une image binaire ou en niveau de gris une opération d'érosion puis de dilatation. Cette application peut alors être parallélisée en utilisant un pipeline à deux étages répliqué quatre fois au sein d'une ferme. Son expression sous forme de squelette est alors :

$$\mathcal{A}_{morpho} \ = \ \mathsf{run}(\ \mathsf{Farm}\ 4\ (\ \mathsf{Pipe}(\ \mathsf{Seq}\ \textit{erode},\ \mathsf{Seq}\ \textit{dilate})\ )\ )$$

Cette définition produit alors le graphe suivant et le place sur les SPEs (fig. 4).

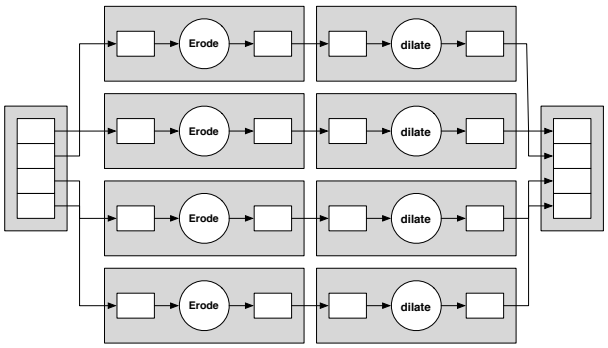

Fig. 4. Graphe de tâche de l'application de morpho-mathématique

3.2. De Quaff à Skell BE : Adaptation de la sémantique

Comme Quaff, Skell BE utilise un modèle à base de **réseaux de processus séquentiels communicants** dans lequel un programme parallèle est décrit comme un réseau de processus. La sémantique formelle décrite dans [10] est conçue pour être aisément modifiable et extensible. Ici, nous allons décrire quelles modifications ont été apportées pour tenir comptes des spécificités architecturales du CELL et des modèles de programmations qu'il supporte.

Support MPI pour le CELL
Plusieurs versions de MPI ont été proposé pour le CELL[11,12,13] mais toutes se fondent sur un portage complet des fonctionnalités du standard MPI. Notre approche a consisté à fournir un noyau plus compact ne supportant que les fonctions nécessaires à l'exécution des squelettes. En utilisant une méthode de transfert direct de SPE à SPE, nous maximisons l'utilisation de la bande passante mise à disposition. Les fonctionnalités supportés sont :

– gestion du rang d'un SPE et de communicateurs inter-SPEs ;
– envoi et réception synchrone de SPE à SPE via DMA ;
– envoi et réception synchrone de SPE à PPE via DMA ;
– barrière de synchronisation inter-SPEs ;

Une fois cette couche logicielle mise en place, l'adaptation de QUAFF se limite à des extensions des règles de sémantiques associées aux squelettes.

Modifications du squelette *Farm*
La principale différence entre QUAFF et SKELL BE réside dans le rôle particulier du squelette *Farm*. Dans SKELL BE , le processus maitre gérant la ferme est inexistant, les SPEs allant chercher par eux mêmes les éléments du flux dont ils ont besoin depuis la mémoire centrale. Pour ce faire, les groupes de SPEs prenant part à un même squelette *Farm* sont numérotés. La définition d'un processus est alors modifiée en y ajoutant deux valeurs : i, qui représente l'index du processus courant au sein d'un bloc SPMD et ω qui représente le nombre total de processus appartenant à ce bloc SPMD. Le reste du descripteur (identifiant de processus p, descripteur de code δ, listes des entrées/sorties I,O) reste identique.

$$\pi \;=\; \langle (p, i, \omega, \delta), I, O \rangle$$

La construction d'une instance de *Farm* passe alors par l'utilisation d'une règle (PAR) modifiée dans laquelle les indexes et tailles de chaque groupe SPMD sont mis à jour de manière adéquate.

$$\frac{\pi_i = \langle P_i, I_i, O_i \rangle \ (i = 1, 2) \qquad |P_1| = m \qquad |P_2| = n}{\begin{array}{c} \pi_1 \parallel \pi_2 = \langle P[(p_1^j, i_1, \omega_1, \delta_1) \leftarrow (p_1^j, i_1, \omega(P_1) + \omega(P_2), \delta)]_{j=1\ldots m} \\ [(p_2^j, i_2, \omega_2, \delta_2) \leftarrow (p_2^j, i_2 + \omega(P_1), \omega(P_1) + \omega(P_2), \delta)]_{j=1\ldots n}, \\ I_1 \cup I_2, O_1 \cup O_2 \rangle \end{array}} \quad \text{(PAR)}$$

Gestion du flux de données
Alors que QUAFF délègue la gestion de la source du flux de données au développeur, elle nécessite, dans le cas du CELL, l'utilisation de primitive de transfert DMA complexe. SKELL BE prend en charge cette gestion en fournissant un type abstrait (le type `tile`) qui encapsule ces fonctionnalités. L'impact principal de cette prise en charge par la règle de production $\widetilde{}$ qui insère dans la liste de code à exécuter par chaque processus les instructions de transferts entre le PPE et les SPEs. Cette règle s'exprime alors comme :

$$\frac{\pi = \langle P, I, O \rangle \qquad |I| = m \qquad |O| = n}{\begin{array}{c} \widetilde{\pi} = \langle P[(o^j, i, \omega, \delta) \leftarrow (o^j, i, \omega, \delta{+}{+}[\mathsf{Put}])]_{j=1\ldots n} \\ [(i^j, i, \omega, \delta) \leftarrow (i^j, i, \omega, [\mathsf{Get}]{+}{+}\delta)]_{j=1\ldots m}, I, O \rangle \end{array}} \quad \text{(LINK)}$$

3.3. Implantation de Skell BE en C++

Les règles de production introduite dans [10] et modifiée ci-dessus sont implantées via un mécanisme d'**évaluation partielle** [14,15,16]. Ce mécanisme se décompose en trois étapes : génération de l'arbre de syntaxe abstraite de l'application, transformation de l'AST en réseau de processus puis génération du code résiduel déduit des règles de production (fig. 5).

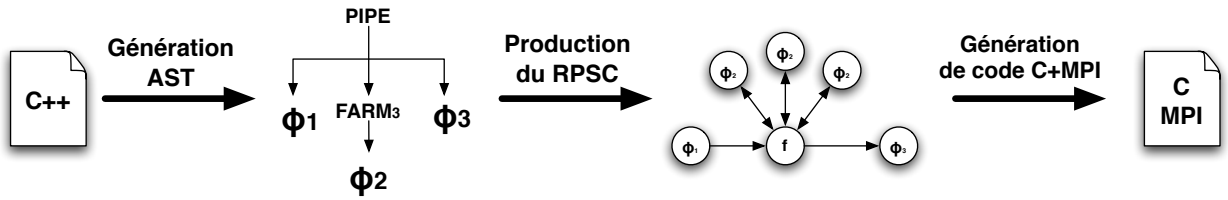

Fig. 5. Processus de génération de code par méta-compilation

L'idée repose sur le fait que chaque entité de ces règles est représenté au sein de Skell BE comme un *type* C++. Les règles en elles-même sont ensuite décrites comme des *méta-programmes* [17] manipulant et générant de tels types. Pour chaque squelette, un constructeur explicite est fourni par l'API de Skell BE. Cette fonction génère une valeur dont **le type** représente, au sein de l'arbre de syntaxe abstraite décrivant l'application, le squelette associé. Par exemple, la fonction `pipeline` se définit ainsi :

```
template<class U,class V> static inline Serial<U,V> pipeline(const U&,const V&)
{
  return Serial<U,V>();
}
```

Dans l'exemple de $\mathcal{A}_{morpho}$, l'appel à la fonction `run()` déclenche successivement l'instanciation des templates des types associés. Le code résiduel suivant est alors généré :

```
run( Farm< Serial< Seq<erode>, Seq<dilate> >,4>() );
```

Ce template contient désormais les informations sur la structure de l'arbre de syntaxe de l'application sous une forme utilisable par nos méta-fonctions. Le listing 1 présente les structures représentant un *graphe de processus*, un *processus* et un *descripteur de processus*. Ces structures, définies comme des types templates, encodent statiquement les caractéristiques de chacune de ces entités. L'ensemble des ces éléments est représentés soit par des constantes entières encapsulées dans un type, soit par une **liste de type** [18].

Listing 1. Structure de données statiques pour process_network, process et descriptor

```
template<class P,class I,class O> struct process_network
{
  typedef P   process;
  typedef I   inputs;
  typedef O   outputs;
};

template<class ID,class DESC, class IDX, class OMG, class IT, class OT> struct process
{
  typedef ID      pid;
  typedef DESC    descriptor;
  typedef IDX     index;
  typedef OMG     omega;
  typedef IT      input_type;
  typedef OT      output_type;
};

template<class IPID,class OPID,class CODE, class KIND> struct descriptor
{
  typedef IPID  i_pids;
  typedef OPID  o_pids;
  typedef CODE  instrs;
  typedef KIND  kind;
};
```

La fonction `run` doit alors convertir le type décrivant l'AST généré précédemment en un type représentant le graphe de processus associé. Pour ce faire, `run` extrait le type de son paramètre et le transmet à la méta-fonction `convert` qui est statiquement surchargée pour chaque type de squelette. Le listing 2 présente la surcharge de cette méta-fonction pour le squelette pipeline.

Listing 2. Surcharge template de `convert` pour `pipeline`

```
template<class S0,class S1,class ID> struct convert<Serial<S0,S1>,ID>
{
  typedef Serial<S1,mpl::void_>                       tail;
  typedef typename convert<S0,ID>::type               proc1;
  typedef typename convert<S0,ID>::new_id             next_id;
  typedef typename convert<tail,next_id>::new_id      new_id;
  typedef typename convert<tail,next_id>::type        proc2;
  typedef typename rule_serial<proc1,proc2>::type     type;
};
```

`convert` extrait les informations sur les squelettes contenus dans le pipeline depuis les types `S0` et `S1`, les convertis en graphe de processus, calcule un nouvel identifiant pour les prochains processus et applique la règle de production idoine (Serial), elle-même implantée sous forme de méta-fonction (listing 3) utilisant les méta-fonctions `transform` et `copy` fournis par BOOST : :MPL.

Listing 3. Version méta-programmée de la règle (SERIAL)

```
template<class P1, class P2> struct rule_serial
{
  typedef typename P1::process  proc1;
  typedef typename P2::process  proc2;

  typedef typename P1::inputs   i1;
  typedef typename P1::outputs  o1;
  typedef typename P2::inputs   i2;
  typedef typename P2::outputs  o2;

  typedef typename transform< proc1, apply_suffix<_1,o1,i2> >::type np1;
  typedef typename transform< proc2, apply_prefix<_1,i2,o1> >::type np2;

  typedef typename copy<np2, back_inserter< np1 > >::type  process;
  typedef typename P1::inputs                              inputs;
  typedef typename P2::outputs                             outputs;

  typedef process_network<process,inputs,outputs>          type;
};
```

3.4. Interface utilisateur

Contrairement à Quaff ou la définition de l'application était effectuée dans un seul source, Skell BE nécessite la définition de deux fichiers sources : un fichier pour la partie PPE et un pour la partie SPE. Si nous reprenons l'exemple de $\mathcal{A}_{morpho}$, les listings 5 et 4 présente le code effectivement écrit par l'utilisateur. Le code SPE (listing 4) est très concis : le fragment de code Skell BE est directement décris au sein de la construction `BEGIN_SKELL_KERNEL`.

Listing 4. Exemple d'application Skell BE – Source SPE

```
BEGIN_SKELL_KERNEL(morpho)
{
  run( farm<4>( pipeline( seq(erode), seq(dilate) ) ) );
}
END_SKELL_KERNEL()
```

Le code PPE (listing 5) se compose des fragments de codes séquentiels et d'un appel au kernel défini dans le source SPE via la fonction `RunKernel` dont les arguments contiennent le kernel à appeler, la donnée d'entrée, la donnée de sortie et la forme des bandes à transférer au SPEs.

Listing 5. Exemple d'application Skell BE – Source PPE

```
SKELL_REGISTER_KERNEL(morpho);
int main()
{
  skell::data<int> in(1,512,1,512),out(1,512,1,512);
  RunKernel(morpho,in,out,tiling(1,512,1,8));
}
```

4. Résultats

Nous nous intéressons ensuite à évaluer les performances de Skell BE. Pour ce faire plusieurs types de tests ont été mené sur une lame IBM QS20, les tests sur simulateurs étant peu fiables car sous-estimant les temps de communication, de synchronisation et de transferts. Pour tous ces tests, nous nous placerons dans une configuration maximisant la bande passante [7].

4.1. Surcoût par rapport à une implantation manuelle

Les premiers tests concernent le surcoût introduit par l'implantation de Skell BE par rapport à une implantation manuelle. Deux familles d'applications sont étudiées : la première utilise un squelette de type pipeline comportant un nombre croissant d'étapes, la seconde utilise le squelette farm avec un nombre de réplications croissants. Pour chaque instance de chaque famille, nous évaluons le temps d'exécution d'une telle application codée manuellement et celle de l'application Skell BE équivalente. Nous calculons alors le surcoût ω_{pipe} et ω_{farm} entre ces implantations. Le tableau 1 résume les résultats obtenus.

# SPU	1	2	3	4	5	6	7	8
ω_{pipe}	3.67%	3.51%	3.34%	3.09%	2.86%	2.65%	2.43%	2.32%
ω_{farm}	1.63%	1.51%	1.42%	1.37%	1.23%	1.18%	1.11%	1.05%

Tab. 1. Surcoût introduit par Skell BE pour *Pipeline* et de *Farm*

Dans tout les cas, le surcoût introduit par Skell BE est inférieur à 1.7% pour *Farm* et 3.7% pour *Pipeline*, et ce, quelque soit le nombre de SPEs utilisés. L'origine de ce surcoût est facilement expliquée par le code supplémentaire introduit par Skell BE pour gérer les communications et le démarrage des tâches.

4.2. Mesure d'accélération

Le deuxième tests consiste à évaluer comment l'accélération des applications Skell BE se comportent lorsque un nombre croissants de SPEs est utilisé. Pour ce faire, nous évaluons l'accélération fournie par un *pipeline* ou une *Farm* comportant N étages par rapport à une application similaire déployée sur un seul SPE. L'efficacité – c'est à dire le ratio entre cette accélération et le nombre de SPEs mis en œuvre – est alors évaluée. Le tableau 2 résume les résultats obtenus en donnant, pour chaque squelette, le nombre de cycles par point de calcul (*cpp*), l'accélération mesurée (γ) et l'efficacité (ϵ).

Nbr de SPE	CPP_{pipe}	γ_{pipe}	ϵ_{pipe}	CPP_{farm}	γ_{farm}	ϵ_{farm}
1	32.591	–	–	4.017	–	–
2	16.675	1.954	0.977	1.988	2.021	1.011
4	8.566	3.805	0.951	1.008	3.985	0.996
8	4.347	7.496	0.937	0.499	8.051	1.006

Tab. 2. Accélération et efficacité pour *Pipeline* et *farm*

Pour le squelette *Pipeline*, l'efficacité moyenne est de l'ordre de 95%. Pour le squelette *Farm*, elle est de l'ordre de 99%. Cet écart s'explique principalement par la quantité de communications qui diffère entre les deux schémas de parallélisation.

5. Conclusion

Nous avons mis en évidence des stratégies de parallélisation pour le déploiement automatique d'applications sur le processeur CELL. Nous avons montré comment ces stratégies peuvent être exprimées comme des constructions de haut-niveau appelées *squelettes*. Nous avons alors défini un *Domain Specific Language* pour construire des applications utilisant ces squelettes et implanté ce langage sous forme d'une bibliothèque C++ – Skell BE – qui utilise la méta-programmation *template* pour générer un code efficace. Nos résultats expérimentaux montrent que les performances des applications utilisant cette bibliothèque sont équivalentes à celles d'applications déployées manuellement.

Les perspectives de ces travaux incluent le développement d'applications de traitement d'images plus complexes, la formalisation de nouveaux squelettes pour le traitement d'images moins réguliers et dépendant des données et la simplification de la chaîne de développement en éliminant le besoin de travailler avec deux fichiers sources. Les évolutions à plus long terme concernent la fusion des interfaces et des sémantiques de Skell BE et Quaff afin de permettre le déploiement d'applications sur des *clusters* de lames CELL ou de *Playstation 3*.

Remerciements

Nous tenons à remercier Pascal Vezolle et Francois Thomas, de IBM France Deep Computing, pour leur aide et leurs précieux conseils.

Bibliographie

1. Pham, D., Aipperspach, T., Boerstler, D., Bolliger, M., Chaudhry, R., Cox, D., Harvey, P., Harvey, P., Hofstee, H., Johns, C., Kahle, J., Kameyama, A., Keaty, J., Masubuchi, Y., Pham, M., Pille, J., Posluszny, S., Riley, M., Stasiak, D., Suzuoki, M., Takahashi, O., Warnock, J., Weitzel, S., Wendel, D., Yazawa, K. : Overview of the architecture, circuit design, and physical implementation of a first-generation cell processor. IEEE Journal of Solid-State Circuits **41** (2006) 179–196
2. Fatahalian, K., Knight, T.J., Houston, M., Erez, M., Horn, D.R., Leem, L., Park, J.Y., Ren, M., Aiken, A., Dally, W.J., Hanrahan, P. : Sequoia : Programming the memory hierarchy. In : Proceedings of the 2006 ACM/IEEE Conference on Supercomputing. (2006)
3. McCool, M.D. : Data-parallel programming on the cell be and the gpu using the rapidmind development platform. In : GSPx Multicore Applications Conference. (2006)
4. Consortium, UPC. : UPC Language Specifications. Version 1.2 edn. Lawrence Berkeley National Lab Tech Report LBNL-59208 (2005)
5. Cole, M. : Algorithmic skeletons : structured management of parallel computation. MIT Press (1989)
6. IBM : Cell Broadband Engine Programming Handbook. Version 1.0 edn. IBM (2006)
7. Saidani, T., Lacassagne, L., Bouaziz, S., Khan, T.M. : Parallelization strategies for the points of interests algorithm on the cell processor. In : Parallel and Distributed Processing and Applications, 5th International Symposium, ISPA 2007, Niagara Falls, Canada, August 29-31, 2007, Proceedings. (2007) 104–112
8. Cole, M. : 13, Algorithmic skeletons. In : Research Directions in Parallel Functional Programming. Springer (1999)
9. Kuchen, H. : A skeleton library. In : Euro-Par '02 : Proceedings of the 8th International Euro-Par Conference on Parallel Processing, London, UK, Springer-Verlag (2002) 620–629
10. Falcou, J., Sérot, J. : Formal semantics applied to the implementation of a skeleton-based parallel programming library. In : Proceedings of the International Conference ParCo. (2007)
11. Ohara, M., Inoue, H., Sohda, Y., Komatsu, H., Nakatani, T. : Mpi microtask for programming the cell broadband enginetm processor. IBM Syst. J. **45** (2006) 85–102
12. Kumar, A., Jayam, N., Srinivasan, A., Senthilkumar, G., Baruah, P.K., Kapoor, S., Krishna, M., Sarma, R. : Feasibility study of mpi implementation on the heterogeneous multi-core cell be™ architecture. In : SPAA '07 : Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures, New York, NY, USA, ACM (2007) 55–56

13. Kumar, A., Senthilkumar, G., Krishna, M., Jayam, N., Baruah, P.K., Sharma, R., Srinivasan, A., Kapoor, S. : A buffered-mode mpi implementation for the cell betm processor. In Shi, Y., van Albada, G.D., Dongarra, J., Sloot, P.M.A., eds. : International Conference on Computational Science (1). Volume 4487 of Lecture Notes in Computer Science., Springer (2007) 603–610
14. Futamura, Y. : Partial evaluation of computation process - an approach to a compiler-compiler. Higher-Order and Symbolic Computation **12** (1999) 381–391
15. Veldhuizen, T.L. : C++ templates as partial evaluation. In Danvy, O., ed. : Proceedings of PEPM'99, The ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, San Antonio, University of Aarhus, Dept. of Computer Science (1999) 13–18
16. Herrmann, C.A., Langhammer, T. : Combining partial evaluation and staged interpretation in the implementation of domain-specific languages. Sci. Comput. Program. **62** (2006) 47–65
17. Veldhuizen, T. : Using C++ template metaprograms. C++ Report **7** (1995) 36–43 Reprinted in C++ Gems, ed. Stanley Lippman.
18. Abrahams, D., Gurtovoy, A. : C++ Template Metaprogramming : Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series). Addison-Wesley Professional (2004)