



---

**RAPPORT**  
de stage de master 2  
**Mention Informatique**

**DE L'UNIVERSITÉ PIERRE ET MARIE CURIE**

Spécialité: Systèmes Et Applications Répartis

par

**Racha Ahmad**

---

**Titre**

**Exécution parallèle sur cartes graphiques d'un  
système de gestion de flux de travail de  
calculs numériques**

---

Responsables de stage :

**M. Emmanuel Chailloux - LIP6, UPMC**

**M. Mathias Bourgoin - LIFO, Université d'Orléans**

**Année universitaire 2015-2016**



# Table des matières

<b>Introduction</b>	<b>3</b>
<b>1 État de l'art</b>	<b>5</b>
1.1 Contexte général . . . . .	5
1.2 L'approche algébrique . . . . .	6
1.3 Le parallélisme . . . . .	6
1.4 La programmation GPGPU . . . . .	8
1.5 Motivation et objectifs . . . . .	8
1.5.1 Motivation pratique . . . . .	8
1.5.2 Objectifs . . . . .	9
<b>2 Analyse des outils</b>	<b>11</b>
2.1 Chiron, analyse et exploitation . . . . .	11
2.1.1 Le modèle de l'exécution dans Chiron . . . . .	11
2.1.2 Les flux de travail réalisés afin d'être gérés par Chiron . . . . .	15
2.2 SPOC, analyse et prise en main . . . . .	19
2.2.1 Fonctionnement de SPOC . . . . .	20
2.2.2 Langage SAREK . . . . .	20
2.2.3 Programmes réalisés en OCaml . . . . .	20
2.2.4 Implémentation des kernels en SPOC . . . . .	22
2.3 Combiner SPOC et Chiron . . . . .	23
<b>3 Étude expérimentale - Combinaison Chiron/SPOC</b>	<b>29</b>
3.1 Exemple arithmétique . . . . .	29
3.2 Exemple de traitement d'image . . . . .	31
3.2.1 Détection des contours : <i>Canny filter</i> . . . . .	31
3.2.2 Les étapes de filtrage . . . . .	31
3.2.3 <i>Canny filter</i> via Chiron et SPOC . . . . .	33

3.2.4	<i>Canny filter</i> via une implémentation CPU . . . . .	36
<b>4</b>	<b>Conclusion</b>	<b>39</b>
<b>Annexes</b>		<b>40</b>
<b>A</b>	<b>Logiciels</b>	<b>41</b>
A.1	Chiron . . . . .	41
A.2	SPOC . . . . .	41
<b>B</b>	<b>Fichier de configuration de flux de travail XML</b>	<b>43</b>
<b>C</b>	<b>Résultats : <i>Canny filter</i></b>	<b>45</b>
<b>D</b>	<b>Les kernels de calcul : <i>Canny filter</i></b>	<b>49</b>
D.1	Gaussian . . . . .	49
D.2	Sobel . . . . .	51
D.3	Non max suppression . . . . .	54
D.4	Hysterises . . . . .	57
<b>E</b>	<b>L'implémentation de Gray et Hysterises en SAREK : <i>Canny filter</i></b>	<b>59</b>
E.1	Gray . . . . .	59
E.2	Hysterises . . . . .	62
<b>F</b>	<b>Exemple d'un fichier xml de configuration Chiron</b>	<b>67</b>
	<b>Bibliographie</b>	<b>71</b>

**Introduction** Ce rapport décrit mon stage effectué au sein de l'équipe APR (Algorithmes, Programmes et Résolution) au laboratoire LIP6 à l'Université Pierre-et-Marie-Curie sous la direction de M. Emmanuel Chailloux et M. Mathias Bourgoin. Le stage de fin d'étude s'inscrit dans le cadre du Master d'informatique spécialité Systèmes et Applications Répartis (SAR). Il s'est déroulé du 2 février au 31 Juillet 2016.

L'intitulé de mon stage est « Exécution parallèle sur cartes graphiques d'un système de gestion de flux de travail de calculs numériques ». Ce stage s'est déroulé dans le cadre du projet *Parallel Scientific Workflow Management Systems on GPU* de Sorbonne universités en collaboration avec les universités brésiliennes UFRJ<sup>1</sup> et UFF<sup>2</sup>.

Le but du stage est de réaliser une combinaison entre un système de gestion de flux de travail (le flux de travail est la représentation d'une suite de tâches ou opérations à effectuer) et la programmation GPGPU (l'abréviation de general-purpose computing on graphics processing units). Plus précisément, entre le système de gestion de flux de travail : *Chiron* ([15] et Annexe A.1), et l'outil de la programmation GPGPU : *SPOC*, ([1, 2] et Annexe A.2). De ce fait, il faut étudier et analyser la faisabilité de combiner Chiron et SPOC, concrétiser des études expérimentales et évaluer la performance de cette combinaison. Afin de remplir les objectifs de ce stage j'ai procédé en plusieurs étapes. (i) D'abord il a fallu installer et préparer l'environnement de travail à savoir le système d'exploitation et un environnement pour la programmation GPGPU adapté avec les pilotes associés et bien sûr les deux outils de base dans notre stage Chiron et SPOC (ii) En parallèle j'ai consacré le temps suffisant pour faire l'état de l'art de sujet. (iii) L'étape suivante était la prise en main de Chiron et SPOC. Pour cela j'ai effectué plusieurs flux de travail à gérer par Chiron (séparation de SPOC) et j'ai analysé le modèle de l'exécution dans Chiron. Pour SPOC j'ai appris le langage de programmation OCaml et fait en sorte d'avoir une certaine maîtrise pour achever des implémentations propres et optimisées. (iv) Une fois pratiqué et maitrisé Chiron et SPOC séparément je suis passée à l'étape essentielle de mon stage à savoir combiner SPOC et Chiron. J'ai concrétisé des études expérimentales pour illustrer cette combinaison et j'ai focalisé les comparaisons sur un flux de travail de traitement d'images qui s'effectue en plusieurs étapes : "The canny filter". Les comparaisons sont faites avec une implémentation CPU et différents paradigmes de programmation GPGPU. A la fin de mon stage j'ai analysé plus profondément les codes sources de Chiron afin d'ajouter un environnement GPGPU spécifique pour Chiron pour les traitements des flux de travail qui prend en considération le comportement de la combinaison faite et que j'estime améliorer (limiter les transferts, envoyer plusieurs flux de travail en même temps selon la disponibilité de GPU, ...etc). Mais, cette partie n'est pas finalisée.

Le rapport de stage est articulé de la manière suivante. Dans le Chapitre 1 je décris le contexte général, l'état de l'art et les motivations et objectifs. En Chapitre 2 j'analyse et je teste séparément le système de gestion de flux de travail, Chiron, et l'outil de la programmation

---

1. UFRJ - l'Université Fédérale de Rio de Janeiro

2. UFF - l'Université Fédérale Fluminense.

GPGPU, SPOC, ensuite réalise la combinaison Chiron/SPOC. Des études expérimentales en Chapitre 3 sont effectué avec comparaison de performance. Finalement en Chapitre 4 je conclue le travail.

# Chapitre 1

## État de l'art

### 1.1 Contexte général

Le progrès massif dans les différents domaines scientifiques requiert des analyses plus complexes et raffinées sur les expériences scientifiques. Ces expériences à grande échelle sont généralement composées de plusieurs modèles de calcul mathématiques et informatiques. Ils peuvent impliquer l'exécution des simulations sur plusieurs pas de temps, le *fine-tuning* sur un grand ensemble de paramètres. Afin d'assurer une bonne gestion de ces expériences il faut garantir leurs efficacité, cohérence, et reproductibilité [4]. C'est une tâche très importante mais aussi compliquée compte tenu de la nécessité d'indiquer la provenance des données. Dans le but d'accorder une approche systématique pour modéliser et exécuter de telles expériences, de nombreux domaines scientifiques (comme les services d'informations, et l'analyse de signal) utilisent les systèmes de gestion de flux de travail scientifique (scientific workflow management systems SWfMS) [4]. Les SWfMS sont des outils pour modéliser et orchestrer l'exécution des flux de travail scientifique. Ils permettent aux utilisateurs de spécifier un flux de travail composé par des activités (programmes informatiques) et par le flux de données entre ces activités. Cela permet d'enchaîner et de contrôler des différentes tâches pour réaliser un traitement complexe [14]. Les SWfMS ont démontré leurs capacités dans différents domaines scientifiques comme l'astronomie et la biologie [16], [12], orchestrant la parallélisation des activités de flux de travail [3].

L'un des principaux avantages des systèmes de gestion du flux de travail provient de l'enregistrement de la provenance des données (l'origine et l'historique d'un jeu de données). Les deux caractéristiques importantes de la provenance d'un produit de données sont d'une part le produit ancêtre de données à partir duquel ce produit de données a évolué, et d'autre part le processus de transformation de ce produit ancêtre de données, éventuellement par le biais du flux de travail, qui a aidé à obtenir ce produit de données. Les informations de provenance peuvent être collectées par un modèle orienté processus, où les processus sont les

principales entités pour lesquelles la provenance est recueillie, et la provenance des données est déterminée par l'inspection d'entrée et de sortie des produits de données de ces processus. Si les informations de provenance sont disponibles, ils donnent la possibilité aux utilisateurs d'analyser les résultats de flux de travail, et de décider la dépendance et l'ordonnancement entre les activités de ce flux. Des requêtes soumises sur les données de provenance peuvent alimenter automatiquement les ensembles de données d'entrée pour un flux de travail en temps d'exécution. L'historique de dérivation des ensembles de données décrit comment le flux de travail a été spécifié et exécuté. Il peut être utilisé pour répliquer les données sur un autre site, ou le mettre à jour si un ensemble de données est périmé en raison de modifications apportées à ses ancêtres.

## **1.2 L'approche algébrique pour l'exécution parallèle de flux de travail**

Considérant un flux des tâches avec une dépendance de données entre ces tâches, le modèle d'exécution d'un flux de travail est étroitement couplé à ces spécification [8]. Ce couplage réduit la possibilité d'améliorer les stratégies d'exécution par le système de gestion et par conséquence les améliorations doivent être codées manuellement. Pour répondre à la question de l'optimisation de l'exécution parallèle d'un flux de travail une approche a été proposée dans [14]. Elle consiste à abstraire la gestion d'un flux de travail scientifique sous une approche algébrique, inspirée par l'algèbre relationnelle pour les bases de données. Cela fournit un modèle uniforme de données qui exprime toutes les données d'une expérience par des relations, chaque combinaison de valeurs des paramètres compose un n-uplet, et les activités de gestion de flux de travail consomment et produisent des n-uplets. Cette abstraction a amélioré la conception et la réutilisation du flux de travail, lorsque les activités sont représentées comme des opérateurs algébriques. Les moteurs de gestion de flux de travail peuvent en outre jouer avec des expressions algébriques équivalentes. Ainsi, les SWfMS identifient la façon dont les données sont structurées et ce qu'il faut attendre de chaque activité, en termes de production et de consommation des données. De cette façon, il est possible d'effectuer des optimisations algébriques et d'adopter des stratégies de distribution intelligentes.

## **1.3 Le parallélisme dans les systèmes de gestion du flux de travail scientifique**

L'ampleur des données produites dans les nombreux domaines scientifiques ne cesse pas d'augmenter, et le progrès de capacité de stockage, de bande passante du réseau, et de puissance de traitement est souvent dépassé. Outre les progrès algorithmiques, pour faire face

à l'augmentation des volumes de données on utilise le parallélisme. Cela se reflète dans le développement de l'exécution parallèle des threads sur des puces simples. Ainsi que le développement des infrastructures qui combinent plusieurs machines à des clusters, des grids et des clouds. Les environnements d'exécution de flux de travail parallèle sont généralement conçus afin de fonctionner avec ces configurations matérielles (et systèmes) particulières.

Une grille informatique (en anglais, grid) est une infrastructure virtuelle constituée d'un ensemble de ressources informatiques potentiellement partagées, distribuées, hétérogènes, dé-localisées et autonomes. Cette infrastructure est qualifiée de virtuelle car les relations entre les entités qui la composent n'existent pas sur le plan matériel mais d'un point de vue logique. L'idée d'un grid est de relier les ressources informatiques afin de résoudre des problèmes de calcul exigeants sans avoir besoin d'un super-ordinateur. Plusieurs SWfMS, tels que Pegasus [7] ou Condor Dagman [6], ont été développés pour utiliser les grids pour l'exécution parallèle des flux de travail de calcul intensif. L'informatique en nuage (en anglais, cloud computing) est l'exploitation de la puissance de calcul ou de stockage de serveurs informatiques distants par l'intermédiaire d'un réseau. Ces ressources de calcul et de stockage sont louables sur demande et sur Internet. Le rendement réel des machines virtuelles louées varie considérablement en fonction de la configuration du matériel et de l'utilisation des ressources sous-jacentes partagées par d'autres utilisateurs. Dans les environnements de grids et de clouds, le transfert important sur des zones étendues des données et les retards dans l'instanciation de grandes quantités des tâches, conduit à une dégradation de performance. Un cluster de calcul est un ensemble d'ordinateurs étroitement connectés et fonctionnans comme un système unique. Un nombre toujours croissant de cœurs par processeur et processeurs par cluster ont conduit à un fort potentiel de parallélisation. Puisque les ressources de calcul en clusters sont étroitement couplées, la localité de données est moins problématique par rapport à des environnements distribués, tels que les grids et les clouds. Les clusters fournissent un environnement plus homogène en termes de performances du processeur et de la latence / bande passante entre les nœuds de calcul. Cependant, l'évolutivité des clusters est limitée et les coûts initiaux d'investissement sont assez élevés, ce qui est problématique. De toute évidence, l'exécution des flux de travail scientifique présente des défis différents en fonction de l'infrastructure informatique sous-jacente [3]. Les spécifications pour l'exécution en parallèle d'un flux de travail scientifique sont généralement définies avec un langage de script. Ce langage est traité par un moteur de flux de travail qui génère un plan pour l'exécution parallèle. Ce plan est une mise en correspondance entre l'architecture parallèle des matériels et la planification des tâches. Développeurs de flux de travail doivent décider l'ordre, les dépendances et les stratégies de parallélisation. Ces décisions resserrent les possibilités de parallélisation, et peuvent donner à manquer de grandes opportunités d'optimisation. Le but de ce stage est d'expérimenter l'accélération GPU comme une nouvelle approche de parallélisme dans le domaine de gestion de flux de travail scientifique. Actuellement, le seul SWfMS qui est en mesure de tirer parti des accélérateurs tels que GPGPU est Swift[9]. Swift est un modèle de programmation implicitement

parallèle et déterministe qui applique des applications externes sur des collections de fichiers, en utilisant un style fonctionnel qui simplifie l'exécution parallèle et distribué. Néanmoins, la définition de flux de travail dans Swift est fortement couplée à la procédure d'exécution du flux de travail. Et la provenance de données dans Swift est basée sur l'extraction des données à partir des fichiers de logs, de ce fait, le scientifique peut explorer la base de données de provenance uniquement lorsque l'exécution est terminée.

## 1.4 La programmation GPGPU

Les cartes graphiques (GPU) sont des dispositifs performants et spécialisés dotés de nombreuses unités de calcul, dédiés à l'affichage et au traitement 3D. La technologie GPGPU est l'abréviation de *general-purpose computing on graphics processing units*, c'est-à-dire un calcul générique sur un processeur graphique. Cette technologie exploite la puissance de calcul des GPUs pour le traitement massivement parallèle, elle permet d'accélérer les portions de code les plus lourdes en ressources de calcul, en les parallélisant sur de nombreuses unités de calcul, le reste de l'application restant affecté au CPU. La programmation GPGPU offre une accélération très élevée pour un large éventail d'applications scientifiques et commerciales, nettement supérieure à celle offertes par une architecture basée uniquement sur des CPUs [13]. Les GPUs offrent une possibilité d'améliorer les exécutions parallèles de flux de travail scientifiques, souvent exprimé avec un grand nombre de tâches de calcul. Cependant, plusieurs problèmes et questions importantes ont encore besoin d'être abordées. Par exemple, la programmation des cartes graphiques demande un couplage fort entre les unités de calculs parallèles du GPU avec le CPU pour obtenir de bonnes performances, en particulier sur le transfert de données.

## 1.5 Motivation et objectifs

### 1.5.1 Motivation pratique

Dans ce stage on étudie une combinaison entre un système de gestion de flux de travail et la programmation GPGPU. Comme mentionné précédemment la programmation GPGPU présente une solution très intéressante dans la parallélisation des milliers d'activités indépendantes, comme dans les opérations de MapReduce. Cependant, il y a plusieurs défis ouverts dans l'exploration des différents modèles de parallélisme pendant l'exécution du flux de travail. Parmi les défis à relever, on s'intéresse ici au problème de l'optimisation de la planification d'exécutions parallèles, laquelle doit tirer profit des architectures hétérogènes (multi-coeurs, accélérateurs). Par exemple la programmation des cartes graphiques demande un couplage fort entre les unités de calculs parallèles du GPU avec le CPU pour obtenir de bonnes performances, en particulier sur le transfert de données. Les systèmes hétérogènes nécessitent des conceptions complexes combinant différents paradigmes de programmation pour gérer chaque

matériel de manière spécifique. On cherche alors à abaisser cette complexité dans les flux de travail pour calculs numériques parallèles en utilisant des abstractions de flux de données et des constructions de haut niveau de programmation parallèle pour représenter la spécification du flux de travail et permettre d'optimiser le plan d'exécution parallèle.

Dans ce stage le choix des outils qui feront l'objet cette combinaison, qu'on envisage de réaliser, est imposé. Comme système de gestion de flux de travail on s'appuie particulièrement sur Chiron. Chiron (voir Annexe A.1) a été créé à l'institut COPPE<sup>1</sup> de l'Université Fédérale de Rio de Janeiro (UFRJ) avec l'équipe de recherche de Marta Matosso au laboratoire NACAD. Pour la programmation GPGPU, on utilise SPOC (voir Annexe A.2) qui consiste à une abstraction haut niveau de la programmation GPGPU. Spoc est un outil issu du travail de l'équipe APR du laboratoire LIP6 à l'Université Pierre et Marie Curie.

### 1.5.2 Objectifs

Dans ce projet nous visons à aborder l'efficacité dans l'exécution parallèle d'un flux de travail de calcul numérique. En laissant plus d'espace au moteur de gestion de flux de travail pour prendre les décisions sur les choix des plans d'exécutions. Ce travail consiste à combiner les abstractions de programmation de différents niveaux, à savoir au niveau du langage de spécification de flux de travail (groupe brésilien) et au niveau de la programmation parallèle (groupe français). La sémantique des opérations algébriques fournit l'exécution en parallèle en mappant depuis la langue de description de flux de travail vers des expressions algébriques équivalentes, nous prévoyons que l'expression peut encore être mise en correspondance avec des abstractions de SPOC. De cette façon, le mapping vers SPOC profitera de la sémantique des opérateurs algébriques. L'objectif est donc de combiner la sémantique des opérations algébriques de dataflow, comme celles proposées dans le système Chiron, avec la puissance de construction de composition de squelettes SPOC pour obtenir la génération dynamique de l'ordonnancement des exécutions parallèles. Cette combinaison, en plus d'être complémentaire, est originale dans le cadre de flux de travail pour calculs numériques. Ensemble, ils ont le potentiel d'isoler le matériel et la programmation de bas niveau de la spécification de haut niveau du flux de travail. Les abstractions prévoient également l'application des règles d'optimisation génériques.

Il existe plusieurs encapsulations de Chiron (voir Annexe A.1) dans des différents domaines (Clusters, Cloud-based..). Nous visons à développer une nouvelle encapsulation de Chiron qui supporte l'accélération GPU pour l'exécution parallèle de workflow. Le traitement parallèle dans Chiron est obtenu dans un style MapReduce (Hadoop). La flexibilité amenée par ce concept algébrique nous permet d'utiliser un mode d'activation d'activité qui nous permet d'ordonnancer dynamiquement le flux de travail et de l'optimiser. Dans ce mode les activités des flux de travail sont présentées par des activations (des programmes informatiques externes)

---

1. COPPE – Instituto Alberto Luiz Coimbra de Pós-Graduação e Pesquisa de Engenharia.

[15]. On cherche alors à déployer ces activations en parallèle sur le GPU, et à étudier l'algorithme de l'optimisation et de la planification d'exécution parallèle de ces activations, à cause de la grande taille des données analysées, et l'intensité du calcul.

# Chapitre 2

## Analyse des outils

Dans ce chapitre nous étudions séparément notre système de gestion de flux de travail Chiron, ainsi que l'outil de programmation GPGPU, SPOC. L'étude consiste à explorer les deux outils et réaliser des tests séparés pour se familiariser avec les deux outils et bien les exploiter avant de les combiner. La combinaison sera testé dans le chapitre suivant (Chapitre 3).

### 2.1 Chiron, analyse et exploitation

Chiron [15] est un système de gestion de flux de travail de calculs numériques (scientifique workflow management system), il exécute ces simulations comme une chaîne d'activités (programmes) et un flux de données (dataflow) sur ces activités. Ce système fournit la gestion des simulations scientifiques, leur exécution parallèle tout en enregistrant la provenance des données. Chiron implémente l'approche algébrique dans un style MapReduce. L'utilisation de MapReduce comme approche de programmation permet aux scientifiques de programmer d'une façon plus simple la procédure du calcul en cachant le parallélisme, qui peut être complexe à gérer [15].

Chiron est écrit en Java, le fichier exécutable de Chiron est un fichier jar généré après avoir compilé notre projet. Avant de réaliser plusieurs flux de travail à gérer par Chiron nous présentons le modèle de l'exécution dans Chiron.

#### 2.1.1 Le modèle de l'exécution dans Chiron

Afin d'effectuer une exécution parallèle sur tous les nœuds de calcul de l'environnement, Chiron utilise *Message passing interface* (MPI). Cela nécessite l'installation de la machine virtuelle Java et le MPJ *MPI-LIKE Message passing interface for java*. Chiron intègre également le système de base de données relationnelles PostgreSQL pour gérer l'exécution parallèle.

La figure 2.1 donne une vue de haut niveau du flux de données dans un flux de travail

de Chiron, en utilisant la perspective des processus de MPI. Cette figure illustre les nœuds de calcul de Chiron (chacun avec un rang spécifique) en cours d'exécution dans un environnement HPC en utilisant un espace de disque partagé pour le stockage. Cela signifie que tous les noeuds de calcul ont une vue locale des données de l'expérience et ce n'est pas nécessaire d'effectuer des transferts pendant l'exécution. Chaque nœud de calcul exécute une instance de Chiron. Chaque nœud également rassemble les données de provenance (heure de début, heure de fin, l'état de l'exécution, les journaux, les erreurs) de l'exécution parallèle pour chaque activation.

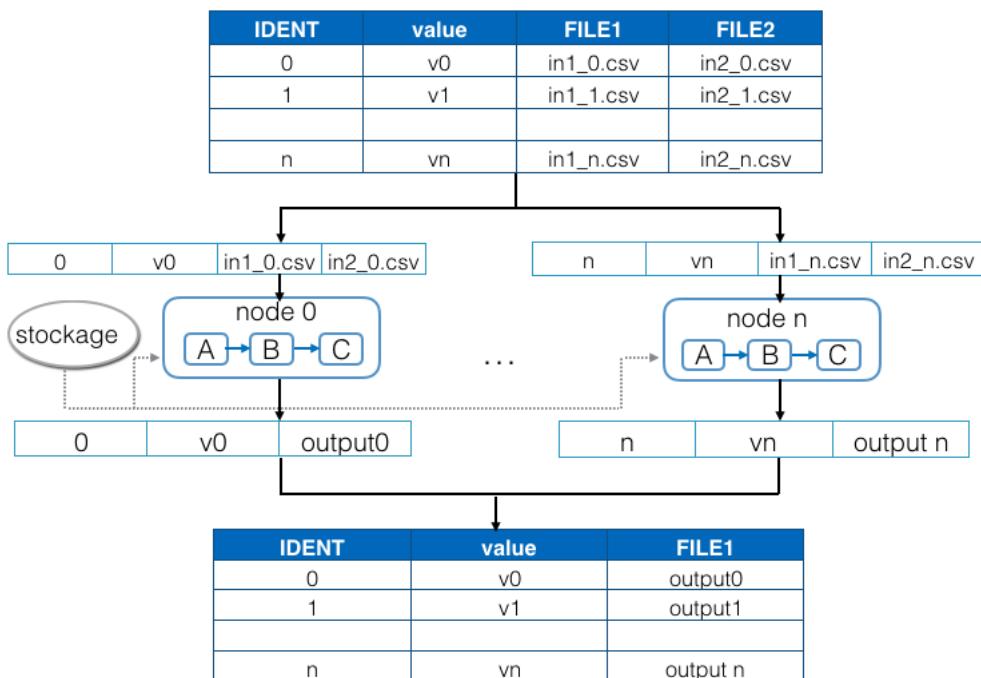


FIGURE 2.1 – Le flux de données dans Chiron

L'architecture de Chiron est illustrée dans le schéma 2.2. Chiron adopte une technique bien connue appelée le parallélisme imbriqué. Elle permet une combinaison de distribution entre les cœurs des noeuds en utilisant MPI et *multi-threading*. En supposant k nœuds et n cœurs par noeud. Chaque nœud représente une instance de Chiron. Et chaque instance a un *thread* pour ordonner les activations disponibles appelé *processor thread*. Dans le nœud 0, il y a un *thread* appelé *workflow processor*, qui orchestre l'exécution du flux de travail et décide ce qui est prêt à être consommé. Ensuite, il distribue les activations prêtes aux *activation schedulers*. *processor thread* est également responsable de lire et écrire des données de provenance dans la base de données lors de l'exécution du flux de travail. Le *activation scheduler* utilise MPI pour communiquer avec le *thread workflow processor*. Cette communication est utilisée pour signaler les activations achevées et l'obtention des nouvelles activations à être consommées. A chaque fois qu'une activation donnée est en mode bloquant, le processeur de flux de travail *workflow processor* répond à une demande avec

un message d'attente au lieu de répondre que l'activation est prête à être consommée. Plus tard, quand *activation schedulers* est en mesure de demander de nouvelles activations à nouveau. Et lorsqu'il n'y a pas plus d'une activation à être consommée, le processeur de flux de travail répond avec un message de terminaison pour terminer l'exécution. Dans ce cas, lorsque le *activation scheduler* reçoit ce type de message, il termine l'exécution de son instance lorsque tous ses processeurs d'activation terminent leurs activations. Toutes les communications entre les instances sont effectuées en utilisant les messages de *polling*, via MPI.

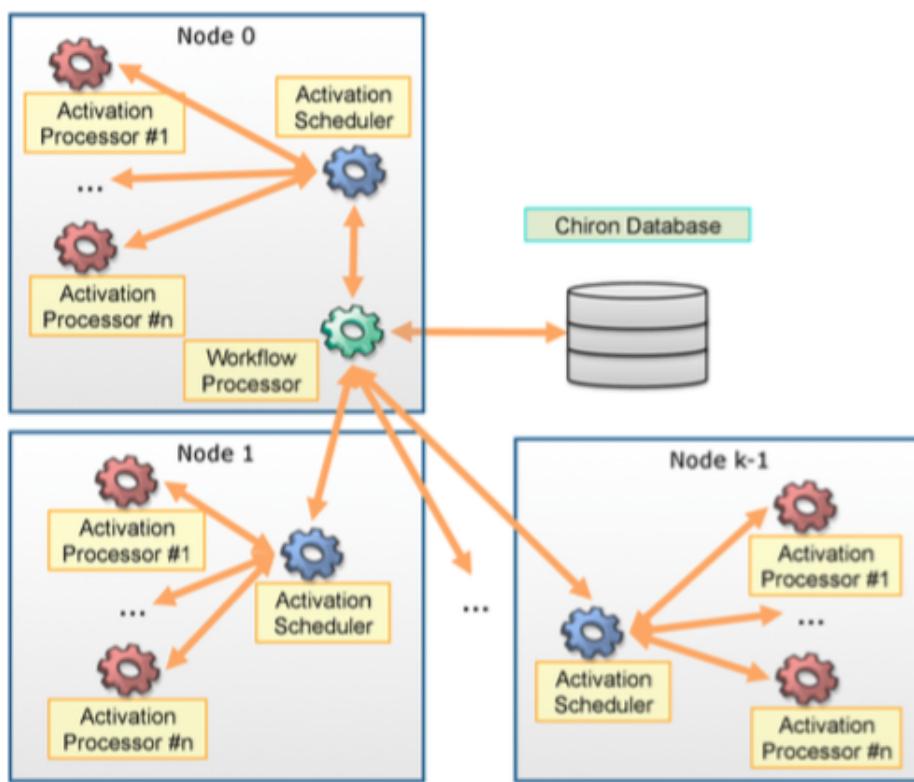


FIGURE 2.2 – l'architecture de Chiron, source de l'image [15]

Chiron utilise l'approche algébrique pour gérer l'exécution parallèle des activations. Toutes les données de flux de travail sont représentées comme des relations d'entrée et des relations de sortie. Chaque relation contient un ensemble d'attributs, et toutes les activités sont régies par l'un des opérateurs algébriques. Les activités peuvent produire plusieurs activations. Les activités peuvent avoir des fichiers de modèle qui sont utilisés pour construire et exécuter les activations. Les activations sont associées à un ensemble de n-uplets et peuvent produire et consommer des fichiers lors de l'exécution.

Les spécifications d'un flux de travail sont écrites dans un fichier de format XML. Le fichier XML (voir Figure 2.3) contient les méta-données (telles que le type de l'opérateur algébrique pour l'activité) et les descriptions concernant le flux de travail, ses activités, leurs schémas, et

```

<Chiron>
  <environment verbose="true"/>
  <constraint workflow_execetag="workflow-query-1" cores="1"/>
  <workspace workflow_dir="/home/racha/Documents/stage/Spoc_Workflows/Files_Workflow"/>
  <database name="chiron" username="chiron" password="chiron" port="5433" server="localhost"/>
  <conceptualWorkflow tag="workflow-query" description="">

    <Activity tag="Ex3Act1" description="" type="Filter" activation=".experiment.cmd;./extractor.cmd"
      template="%=WFDIR%/template_act1">
      <relation reltype="Input" name="IAct1"/>
      <relation reltype="Output" name="OAct1" />
      <field name="ID" type="float" input="IAct1" output="OAct1" decimalplaces="0"/>
      <field name="FILE1" type="file" input="IAct1" output="OAct1">
        <field name="V" type="float" output="OAct1" decimalplaces="0"/>
      </field>
      <field name="FILE2" type="file" input="IAct1" output="OAct1">
        <field name="K" type="float" output="OAct1" decimalplaces="0"/>
      </field>
      <field name="FILE3" type="file" output="OAct1"/>
    </Activity>

  </conceptualWorkflow>

  <executionWorkflow tag="workflow-query" execmodel="DYN_FAF" expdir="%=WFDIR%/exp">
    <relation name="IAct1" filename="input.dataset"/>
  </executionWorkflow>
</Chiron>

```

FIGURE 2.3 – Le fichier de configuration du workflow

la commande d’activation pour chaque activité. Fondamentalement, le XML représente toutes les informations qui sont nécessaires pour exécuter le flux de travail. Les relations entre les activités sont stockées dans des fichiers des données tabulaires et elles sont référencés dans le fichier XML. Dans la figure 2.3 la balise **ConceptualWorkflow** définit le comportement de chaque composant dans un flux de travail. La balise **database** transmet à la base de données les informations de flux de travail. La description des différents éléments XML avec les valeurs de réglage est présentée en détails dans l’Annexe B.

La balise **ExecutionWorkflow** est chargée d’exécuter une instance de flux de travail conceptuel. De telle façon, tous les paramètres nécessaires sont spécifiés dans ce fichier xml.

Lorsque Chiron commence, il analyse le fichier des configurations XML, et convertit la définition de flux de travail XML aux expressions algébriques. Pendant ce processus, Chiron stocke les informations de la provenance dans la base de données. Avec cette représentation algébrique du flux de travail, Chiron effectue l’identification et l’optimisation des fragments du flux de travail (groupe d’expressions algébriques), et envoie les stratégies pour chaque fragment. Après avoir évalué les coûts et choisi le plan de l’exécution Chiron commence l’exécution de flux de travail en se basant sur cette représentation algébrique optimisée. Pendant l’exécution, la provenance rétrospective est également recueilli et stocké dans la base de données. La gestion de provenance dans Chiron utilise une base de données relationnelle. Ce mécanisme

permet la gestion et le suivi des activations dans l'exécution. Ainsi, les scientifiques peuvent exécuter des requêtes qui évaluent les données de provenance lors de l'exécution du flux de travail, et il est possible de savoir quelles activités ont été exécutées et celles qui sont en cours d'exécution. Il est également possible d'identifier les erreurs. Toutes les méta-données du flux de travail, ses activités, activations, les données primitives, et les références de fichiers sont stockés dans la base de données Chiron. Les fichiers eux-mêmes sont maintenus dans la zone de stockage de l'application. Le stockage de ces informations établit une base de données de provenance prospective et rétrospective pour l'environnement HPC.

### 2.1.2 Les flux de travail réalisés afin d'être gérés par Chiron

On présente dans la suite deux flux de travail réalisés sur Chiron dont les activations sont écrites comme des scripts en langage shell.

#### 2.1.2.1 SplitMap workflow

C'est un flux de travail composé de deux activités act1 et act2. La première activité est dirigée par l'opérateur Split-Map, et la deuxième activité est dirigée par l'opérateur Map. Ce flux de travail analyse un fichier de données tabulaires (n-uplets). L'opérateur Split-Map est caractérisé par la production d'un set de n-uplets dans la relation de sortie pour chaque n-uplet consommé en entrée, (voir Figure 2.4).

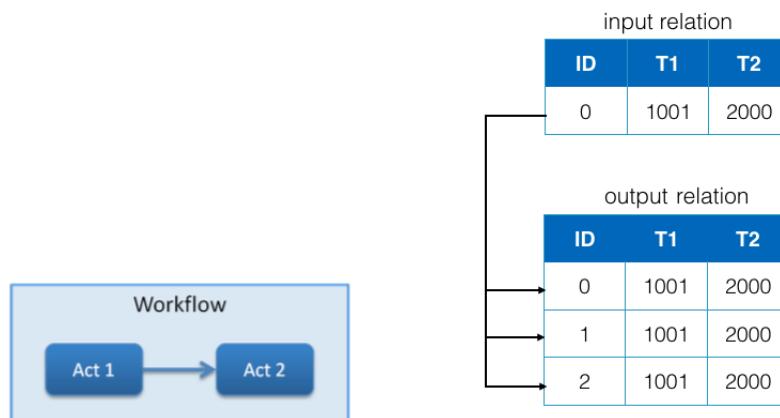


FIGURE 2.4 – structure d'un flux de travail (à gauche ), Split-Map (à droite)[15]

Le code suivant présente l'activation de la première activité act1. Ce code produit pour chaque n-uplet consommé en entrée trois n-uplets en sortie et met à jour le champ ID.

---

<sup>1</sup> echo "IDENT;ID;T1;T2" >> ERelation.txt  
<sup>2</sup> echo "%=IDENT%;(%=ID%\*3));%="T1%;%="T2%" >> ERelation.txt

---

```

3 I=((%ID%*3))
4 echo "%IDENT%;((I+1));%T1%;%T2%" >> ERelation.txt
5 echo "%IDENT%;((I+2));%T1%;%T2%" >> ERelation.txt

```

---

Figure 2.5 présente les tables des deux relations d'entrée et de sortie pour la première activité act1 comme elles sont enregistrées dans la base de données de Chiron. On remarque que plusieurs tâches s'exécutent pour chaque activité, chaque tâche analyse une ligne de donnée du fichier d'entrée. Remarquons également dans la figure 2.5 dans la table de sortie de l'activité act1, les deux colonnes *previousactid* et *nexttactid* montrent que c'est une table intermédiaire qui présente en même temps la sortie de l'activité 822 (act1) et l'entrée de l'activité 823 (act2). Cela reflète le flux des données entre les activités de flux de travail.

	<b>ewkfid</b> <b>integer</b>	<b>key [PK] integer</b>	<b>taskid integer</b>	<b>id double precision</b>	<b>t1 double precision</b>	<b>t2 double precision</b>
<b>1</b>	976	1	1970	0	1000	1001
<b>2</b>	976	2	1971	1	2000	2001
<b>3</b>	976	3	1972	2	3000	3001
<b>4</b>	976	4	1973	3	2000	2002
<b>5</b>	976	5	1974	4	3000	3002
<b>6</b>	976	6	1975	5	1000	1002
*						

	<b>ewkfid</b> <b>integer</b>	<b>key [PK] integer</b>	<b>previousactid integer</b>	<b>nextactid integer</b>	<b>previoustaskid integer</b>	<b>nexttaskid integer</b>	<b>id double precision</b>	<b>t1 double precision</b>	<b>t2 double precision</b>
<b>1</b>	976	1	822	823	1978	1976	0	1000	1001
<b>2</b>	976	2	822	823	1978	1977	1	1000	1001
<b>3</b>	976	3	822	823	1978	1978	2	1000	1001
<b>4</b>	976	4	822	823	1971	1979	3	2000	2001
<b>5</b>	976	5	822	823	1971	1980	4	2000	2001
<b>6</b>	976	6	822	823	1971	1981	5	2000	2001
<b>7</b>	976	7	822	823	1972	1982	6	3000	3001
<b>8</b>	976	8	822	823	1972	1983	7	3000	3001
<b>9</b>	976	9	822	823	1972	1984	8	3000	3001

FIGURE 2.5 – Les données d'entrée (en haut) et de sortie (en bas) de l'activité act1. La source de l'image : la base de données de Chiron après l'exécution de cet exemple.

L'activation de la deuxième activité act2 fait une simple projection de la sortie de act1. figure 2.6 illustre le résultat de la deuxième activité.

---

```

1 echo "IDENT;ID;T2" >> ERelation.txt
2 echo "%IDENT%;%ID%;%T2%" >> ERelation.txt

```

---

Le flux de travail *SplitMap Workflow* génère 24 activations. 6 activations pour act1 (Split-Map) comme on a un fichier (input.dataset) de six lignes de données, et 18 activations pour act2 (Map) pour la projection de la sortie de SplitMap.

	<b>ewkfid</b> <b>integer</b>	<b>key</b> <b>[PK] integer</b>	<b>taskid</b> <b>integer</b>	<b>id</b> <b>double precision</b>	<b>t2</b> <b>double precision</b>
<b>1</b>	976	1	1976	0	1001
<b>2</b>	976	2	1977	1	1001
<b>3</b>	976	3	1978	2	1001
<b>4</b>	976	4	1979	3	2001
<b>5</b>	976	5	1980	4	2001
<b>6</b>	976	6	1981	5	2001
<b>7</b>	976	7	1982	6	3001
<b>8</b>	976	8	1983	7	3001
<b>9</b>	976	9	1984	8	3001

FIGURE 2.6 – Les données de sortie de act2

### 2.1.2.2 Filter workflow

C'est un flux de travail avec deux activités act1 et act2, (voir la structure de ce flux dans Figure 2.4). La première activité est dirigée par l'opérateur Filter, et la deuxième activité est dirigée par l'opérateur Map. L'opérateur Filter est caractérisé par la consommation de chaque n-uplet de la relation d'entrée. Ce n-uplet peut être copier sur la relation de sortie sous une certaine condition appliquée par le programme de l'activation, (voir Figure 2.7).

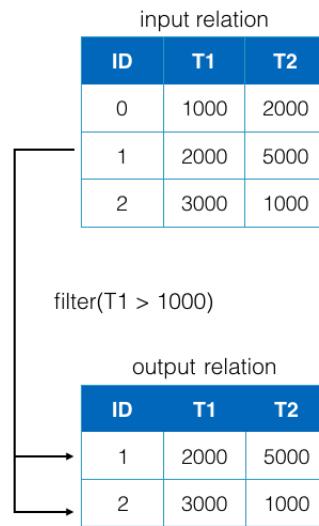


FIGURE 2.7 – Filter

Dans le code suivant on présente l'activation de la première activité act1. Cette activation filtre les valeurs de la colonne T1 de la relation d'entrée sous la condition que les champs ont une valeur supérieure à 1000.

---

```

1 [ "%=T1%" -gt "1000" ] && echo "IDENT;ID;T2" >> ERelation.txt
2 [ "%=T1%" -gt "1000" ] && echo "%=IDENT%;%=ID%;%=T2%" >> ERelation.txt

```

---

Figure 2.8 présente comment les informations des deux relations d'entrée et de sortie de la première activité act1 sont présentées dans la base de données. Comme pour le flux de travail précédent, on remarque l'enchaînement des activités du flux de travail 838 (act1) et 839 (act2), et les différentes tâches déroulées pour chaque activité.

	<b>ewkfid</b> <b>integer</b>	<b>key</b> <b>[PK] integer</b>	<b>taskid</b> <b>integer</b>	<b>id</b> <b>double precision</b>	<b>t1</b> <b>double precision</b>	<b>t2</b> <b>double precision</b>
<b>1</b>	991	1	2010	0	1000	1001
<b>2</b>	991	2	2011	1	2000	2001
<b>3</b>	991	3	2012	2	3000	3001
<b>4</b>	991	4	2013	3	2000	2002
<b>5</b>	991	5	2014	4	3000	3002
<b>6</b>	991	6	2015	5	1000	1002

	<b>ewkfid</b> <b>integer</b>	<b>key</b> <b>[PK] integer</b>	<b>previousactid</b> <b>integer</b>	<b>nextactid</b> <b>integer</b>	<b>previoustaskid</b> <b>integer</b>	<b>nexttaskid</b> <b>integer</b>	<b>id</b> <b>double precision</b>	<b>t2</b> <b>double precision</b>
<b>1</b>	996	1	838	839	2021	2026	1	2001
<b>2</b>	996	2	838	839	2022	2027	2	3001
<b>3</b>	996	3	838	839	2023	2028	3	2002
<b>4</b>	996	4	838	839	2024	2029	4	3002

FIGURE 2.8 – L'entrée (en haut) et la sortie (en bas) de act1.

Le code suivant est le code de la deuxième activité act2, dirigée par l'opérateur Map, qui fait une simple projection des données. La figure 2.9 présente le résultat de cette activité.

---

```

1 echo "IDENT;ID;T2" >> ERelation.txt
2 echo "%=IDENT%;%=ID%;%=T2%" >> ERelation.txt

```

---

	<b>ewkfid</b> <b>integer</b>	<b>key</b> <b>[PK] integer</b>	<b>taskid</b> <b>integer</b>	<b>id</b> <b>double precision</b>	<b>t2</b> <b>double precision</b>
<b>1</b>	991	1	2016	1	2001
<b>2</b>	991	2	2017	2	3001
<b>3</b>	991	3	2018	3	2002
<b>4</b>	991	4	2019	4	3002

FIGURE 2.9 – La sortie de l'activité Map (act2)

Le flux de travail Filter-Workflow génère 10 activations. 6 activations pour Filter, comme on a un fichier (input.dataset) de six lignes de données, et 4 activations pour Map pour la projection de la sortie de l'activité Filter.

Les images sont extraites de la base de données de Chiron. Elles illustrent les informations stockées dans la base et la manière dont la provenance de données est appliquées.

Dans les deux flux de travail précédents on observe bien l'exécution parallèle et distribué de flux de travail. L'exécution d'une activité génère une arborescence des fichiers. Cette arborescence représente les noeuds de calculs et la distributions des données. La figure 2.10 présente un exemple de l'arborescence générée pour une exécution d'une seule activité. Chaque noeud

est une répertoire, les feuilles sont les répertoires d'exécution de chaque tâche de l'activité. Dans chaque répertoire feuille on trouve l'activation, les logs de l'exécution de la tâche et les données de sortie.

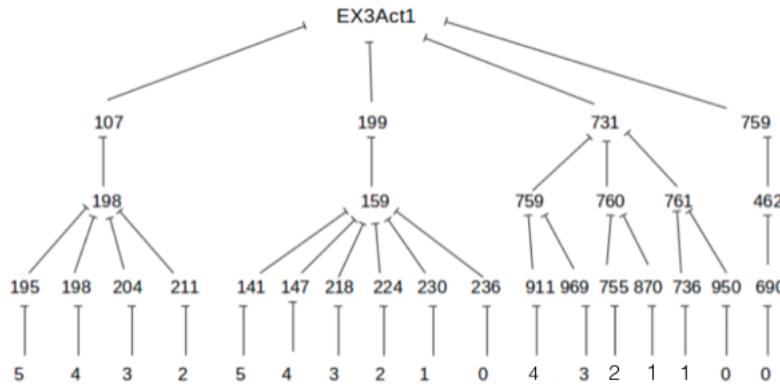


FIGURE 2.10 – Un exemple de l'arborescence générée pour une exécution d'une activité par Chiron.

Après avoir réalisé ces flux de travail gérés par Chiron, on a acquis certaine maîtrise qui nous permet d'utiliser Chiron. L'étape suivante sera consacrée à SPOC, Section 2.2 avant de combiner Chiron et SPOC, Section 2.3.

## 2.2 SPOC, analyse et prise en main

La programmation GPGPU est principalement disponible via deux outils Cuda et OpenCL. Ce sont des outils d'assez bas niveau d'abstraction, ils demandent de manipuler explicitement de nombreux paramètres matériels. SPOC (*Stream Processing with OCaml*) offre, une abstraction haut niveau en unifiant les deux environnements de développement GPGPU (Cuda et OpenCL) en une même bibliothèque. Donc le choix de SPOC est dans le but de rendre cette tâche plus souple et sûre. SPOC nous permet d'atteindre un haut niveau en simplifiant la programmation GPGPU tout en conservant les hautes performances qu'elle permet d'atteindre. Il assure aussi la portabilité, la gestion de la mémoire, il autorise le développement d'optimisations supplémentaires et fournit des squelettes qui peuvent être utilisées pour composer les Kernels<sup>1</sup>. Les squelettes sont des constructions algorithmiques basées sur des patrons de conception communs, qui peuvent être paramétrées pour adapter leur comportement. Les squelettes décrivent explicitement les relations entre les kernels et les données. En utilisant cette information, il est possible d'optimiser automatiquement le calcul global [1].

1. Dans la programmation GPU, une fonction définie par l'utilisateur qui tourne sur le GPU est appelé un Kernel.

### 2.2.1 Fonctionnement de SPOC

SPOC (Stream Processing with OCaml) consiste en une extension à OCaml<sup>2</sup> associée à une bibliothèque d'exécution. L'extension permet la déclaration des noyaux GPGPU externes utilisables depuis un programme OCaml, tandis que la bibliothèque permet de manipuler ces noyaux ainsi que l'automatisation des transferts de données nécessaires à leur exécution, [2].

### 2.2.2 Langage SAREK

Stream ARchitecture Extensible Kernels (SAREK) est un langage intégré à OCaml, dédié à la description des noyaux de calcul. Ce langage est une solution offerte par SPOC. L'utilisation de SAREK nous permettre à simplifier l'analyse du programme et ainsi d'apporter plus de vérification statique et d'optimisations automatiques. En particulier, ce langage nous offre une connexion avec le langage hôte ce qui permet un partage de données transparent entre le CPU et le GPGPU. Ce langage assure aussi la portabilité, en permettant la description des noyaux exécutables sur des architectures Cuda comme OpenCL. On test SAREK dans le Chapitre suivant, Section 3.2.3.3 et des exemples d'implémentations en SAREK se trouvent dans l'Annexe E .

### 2.2.3 Programmes réalisés en OCaml

Le but de programmer en OCaml est de préparer la combinaison entre Chiron et SPOC. On a réalisé deux programmes pour s'en servir après pour réaliser des flux de travail qui exécutent des activités des programmes en OCaml. Le premier fait des calculs arithmétiques et le deuxième explore le traitement des données sous forme de fichiers.

Le code suivant donne une idée sur le premier exemple :

---

```

1 let ars = ref []
2 let _ = Arg.parse ([])(fun s -> ars := s::!ars ) "";
3 Random.self_init();
4 let args = List.rev !ars in
5 let id, t1, t2 = match args with
6 | [id; t1; t2] -> id, t1, t2
7 | _ -> failwith "args error" in
8
9 let oc = open_out "ERelation.txt" in
10 if int_of_string(t1) > 1000 then begin
11   Printf.printf oc "%d;%d;%d\n";

```

---

2. l'implémentation la plus avancée du langage de programmation Caml. Caml provient de *Categorical Abstract Machine Language*

---

```

12     Printf.printf oc "%s;" id;
13     Printf.printf oc "%s;" t1;
14     Printf.printf oc "%s;" t2;
15     close_out oc;
16 end;

```

---

Ce programme se fait en deux étapes. La première étape correspond à l'analyse des données (prévues d'être reçues de la part de Chiron) en entrée. La deuxième étape est une étape importante car on l'utilisera lorsque on combine avec Chiron dans le Chapitre suivant. Dans cette étape on écrit un fichier nommé ERelation qui contient la relation de sortie et les données produites dans cette relation. Chiron collecte les fichiers nommés ERelation pour récolter les résultats de l'exécution d'une activité dans un flux de travail.

Le deuxième programme (voir l'extrait ci-dessus) parse de fichiers en entrée en format csv et il fait la somme entre les données de File1 et File2. La fonction `parse_ints` extrait les données d'un fichier du format csv et les met dans une liste. La fonction `sum` additionne deux listes et renvoi le résultat dans une liste. Dans le fichier ERelation.txt on spécifie les chemins de ces fichiers qui doivent exister dans un espace de mémoire partagé pour tous les nœuds de calculs de flux de travail.

---

```

1 let parse_ints f =
2   let fd = open_in f in
3   let _ = input_line fd in
4   let l = ref [] in
5   let rec aux () = match input_line fd with
6     | s -> l := int_of_string (String.trim s) :: !l ; aux ()
7     | exception End_of_file -> ()
8     | exception Failure _ -> failwith "error"
9   in aux (); close_in fd; !l in
10
11 let ints1, ints2 = parse_ints file1, parse_ints file2 in
12
13 let sortie = "file" ^ (id) ; in
14 let sum a b =
15   let rec isum a b =
16     match a, b with
17     | [], [] -> [0]
18     | [], x | x, [] -> isum [0] x
19     | ah :: at, bh :: bt ->
20       let s = ah + bh in

```

---

```

21      s :: i sum at bt in
22      i sum a b ;
23  in
24  let res = sum ints1 ints2 in
25
26  let oc1 = open_out sortie in
27  Printf.fprintf oc1 "K\n";
28  List.iter (Printf.printf oc1 "%d\n") res;
29  close_out oc1;
30
31  let oc = open_out "ERelation.txt" in
32      Printf.printf oc "id;f1;f2;f3\n";
33      Printf.printf oc "%s;" id;
34      Printf.printf oc "%s;" file1;
35      Printf.printf oc "%s;" file2;
36      Printf.printf oc "%s\n" sortie;
37  close_out oc;

```

---

## 2.2.4 Implémentation des kernels en SPOC

Dans ce travail plusieurs kernels ont été implémentés. On présente ici un kernel simple pour montrer le fonctionnement de SPOC. D'avantage de kernels implémentés lors de la combinaison Chiron/SPOC peuvent se trouver dans l'annexe D et E.

Dans cet exemple on déploie le même exemple d'addition des fichiers en OCaml (Section 2.2.3) mais cette fois sur le GPU en utilisant SPOC. Un extrait de code :

---

```

1 let threadsPerBlock = match !dev.Devices.specific_info with
2     | Devices.OpenCLInfo cli ->
3         (match cli.Devices.device_type with
4             | Devices.CL_DEVICE_TYPE_CPU -> 1
5             | _ -> 256)
6             | _ -> 256
7     in
8     let blocksPerGrid = (!vec_size + threadsPerBlock -1) / threadsPerBlock in
9     let block = { Spoc.Kernel.blockX = threadsPerBlock; Spoc.Kernel.blockY = 1 ;
10                 Spoc.Kernel.blockZ = 1; } in
11     let grid = { Spoc.Kernel.gridX = blocksPerGrid; Spoc.Kernel.gridY = 1 ; Spoc.
12                 Kernel.gridZ = 1;} in

```

---

```

11
12     vec_add#compile (~debug: true) !dev;
13     Spoc.Kernel.run !dev (block, grid) vec_add (a, b,res, !vec_size);
14     Pervasives.flush stdout;

```

---

Le code précédent présente l'initiation de GPU (ligne 1 à 9) et ensuite l'invocation du noyau de calcul. Pour l'addition des données des deux fichiers on utilise le kernel suivant :

---

```

1 #ifdef __cplusplus
2 extern "C" {
3 #endif
4 __global__ void vec_add(float *A, float* B, float* C,
5                         int size)
6 {
7     int index = blockIdx.x*blockDim.x + threadIdx.x;
8
9     if(index<size)
10     C[index] = A[index] + B[index];
11 }
12 #ifdef __cplusplus
13 }
14 #endif

```

---

Remarquons que SPOC nous permet la déclaration des noyaux GPGPU externes utilisables depuis un programme OCaml.

Maintenant, on a tous les éléments nécessaires pour réaliser la combinaison SPOC/Chiron.

## 2.3 Combiner SPOC et Chiron

L'analyse précédente de SPOC et Chiron prépare tous les éléments pour réaliser la combinaison. Dans la Section 2.2.1 on a expliqué que SPOC permet la déclaration des noyaux GPGPU externes utilisables depuis un programme OCaml. Les exemples dans les Sections 2.2.3 et 2.2.4 expliquent comment préparer les liens via des programmes OCaml pour que Chiron communique avec les activités écrites en SPOC. Dans la Section 2.1.1 on a expliqué le modèle de l'exécution dans Chiron, ici on aura une représentation analogue de la vue de haut niveau du flux de donnée dans un flux de travail de Chiron montré dans la Figure 2.1. Plus concrètement, la Figure 2.11 illustre le positionnement d'une activité de SPOC dans l'architecture globale de Chiron.

Maintenant pour lancer un exemple d'exécution parallèle sur cartes graphiques d'un sys-

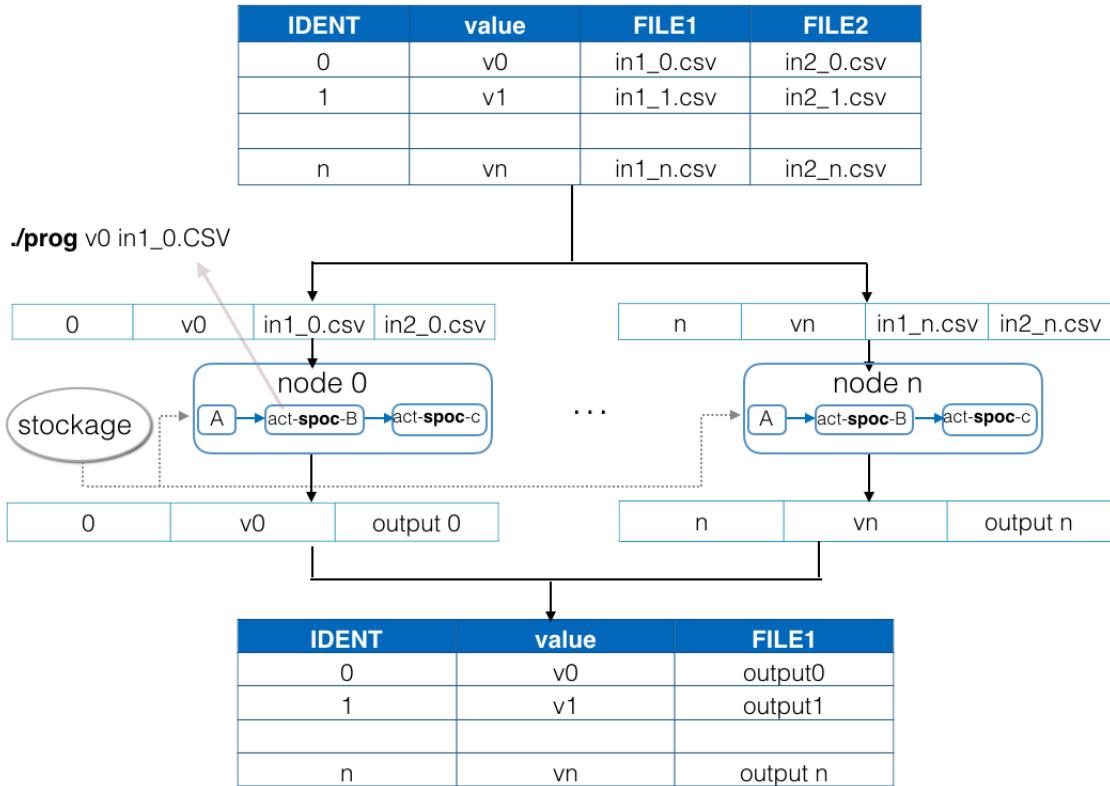


FIGURE 2.11 – l’architecture de l’exécution d’un flux de travail avec SPOC

tème de gestion de flux de travail de calculs numériques en combinant SPOC et Chiron la procédure est la suivante : (i) D’abord bien sûr il faut avoir implémenté tous les kernels nécessaires. (ii) Ensuite il faut dans SPOC prévoir l’initiation de GPU et l’invocation du noyau de calcul (voir l’exemple de la Section 2.2.4). (iii) Également en SPOC il faut comme expliqué dans la Section 2.2.3 prévoir l’écriture sur un fichier nommé ERelation pour communiquer après avec Chiron. (iv) De coté Chiron d’abord, on prépare les données d’entrée de l’activité prévue qu’on stocke dans un fichier appelé `input.dataset`. (v) Ensuite, pour invoquer une activation SPOC il faut configurer le fichier xml comme expliqué dans la Section 2.1.1, par exemple :

---

```

1 <activity tag="my_act" description="" type="REDUCE" operand="ID"
2 activation=". ./experiment.cmd;"
3 template="%=WFDIR%/bin/template">
4 <relation reltype="Input" dependency="previous_act"/>
5           <relation reltype="Output" name="OAct7"/>
6           <field name="ID" type="float" output="OAct7"
               decimalplaces="0"/>
```

---

```

7      <field name="TOTALTIME" type="text" output="OAct7"/>
8      <field name="ACTTIME" type="text" output="OAct7"/>
9      <field name="IMG1" type="text" output="OAct7"/>
10 </activity>
```

---

Le plus important est le script `experiment.cmd` qui est renseigné dans la balise `activation` (ligne 2) qui est chargée de lancer l'exécutable généré par SPOC et lié à l'activité actuel `my_act`. L'exécutable est lancé sur les données déjà renseignées dans le fichier `input.dataset`. Le traitement est fait ligne par ligne de fichier `input.dataset` de manière qu'on peut lancer plusieurs flux de travail en même temps et Chiron s'en charge de les gérer. Un exemple de fichier `input.dataset` :

---

```

1 ID; File
2 1; my_file_1.txt
3 2; my_file_2.txt
4 .
5 .
6 .
7 n; my_file_n.txt
```

---

Et le script `experiment.cmd` qui lance un exécutable SPOC (appelé `exec.byte` dans cet exemple) sur les données de fichier `input.dataset` :

---

```

1 #!/bin/bash
2 cd ~/SPOC/myAct/
3 ./exec.byte "%=ID%" "%=File%"
4 cat Erelation.txt > OLDPWD/ERelation.txt
```

---

Remarquons que dans la ligne 3, on copie les informations qui décrivent les résultats de calcul dans un fichier nommé `ERelation`.

Notons que si on travail sur une seule machine alors Chiron peut utiliser directement l'exécutable déjà compilé et généré par SPOC. Le lancement de l'exécutable se fera via le script `experiment.cmd`. Mais, en général, si on a plusieurs machines où on risque de changer l'architecture et par conséquence avoir un exécutable qui ne se lance pas, alors il faut prévoir dans le script `experiment.cmd` de compiler localement sur le nœud de calcul avant de lancer l'exécutable.

Finalement, rappelons qu'un flux de travail est en général constitué de plus qu'une seule activité. Dans ce cas il faut simplement reconfigurer le xml de Chiron pour prendre en compte l'enchaînement des différentes activités. Ci-dessus un exemple d'un fichier de configuration

xml pour un flux de travail constitué de 5 activités.

---

```

1  <SciCumulus>
2      <environment type="LOCAL" verbose="true"/>
3      <constraint workflow_execetag="workflow-map-gpu-1" cores="2"
4          performance="false"/>
5      <workspace workflow_dir="/home/racha/Documents/stage/workflow_map"/>
6      <database name="scc2" username="scc2" password="scc2" port="5433" server
7          ="localhost"/>
8      <conceptualWorkflow tag="workflow-map" description="">
9          <activity constrained="false" tag="act1" description="" type="MAP"
10             activation=".experiment.cmd" template="%=WFDIR%/
11             template_act1">
12                 <relation reltype="Input" name="IAct1"/>
13                 <relation reltype="Output" name="OAct1" />
14                 <field name="ID" type="float" input="IAct1" output="OAct1"
15                     decimalplaces="0"/>
16                 <field name="START" type="text" output="OAct1"/>
17                 <field name="ACTTIME" type="text" output="OAct1"/>
18                 <field name="IMG1" type="text" input="IAct1" output="OAct1"
19                     OAct1"/>
20             </activity>
21             <activity tag="act2" description="" type="MAP" activation="./
22                 experiment.cmd" template="%=WFDIR%/template_act2">
23                 <relation reltype="Input" dependency="act1"/>
24                 <relation reltype="Output" name="OAct2" output="OAct1"
25                     />
26                 <field name="ID" type="float" output="OAct2"
27                     decimalplaces="0"/>
28                 <field name="START" type="text" output="OAct2"/>
29                 <field name="ACTTIME" type="text" output="OAct2"/>
30                 <field name="IMG1" type="text" output="OAct2"/>
31             </activity>
32             <activity tag="act3" description="" type="MAP" activation="./
33                 experiment.cmd" template="%=WFDIR%/template_act3">
34                 <relation reltype="Input" dependency="act2"/>
35                 <relation reltype="Output" name="OAct3"/>

```

```

27      <field name="ID" type="float" output="OAct3"
28          decimalplaces="0"/>
29      <field name="START" type="text" output="OAct3"/>
30      <field name="ACTTIME" type="text" output="OAct3"/>
31      <field name="IMG1" type="text" output="OAct3"/>
32      <field name="ANGLE" type="text" output="OAct3"/>
33
34  </activity>
35  <activity tag="act4" description="" type="MAP" activation="./
36      experiment.cmd" template="%=WFDIR%/template_act4">
37      <relation reltype="Input" dependency="act3"/>
38      <relation reltype="Output" name="OAct4"/>
39      <field name="ID" type="float" output="OAct4"
40          decimalplaces="0"/>
41      <field name="START" type="text" output="OAct4"/>
42      <field name="ACTTIME" type="text" output="OAct4"/>
43      <field name="IMG1" type="text" output="OAct4"/>
44      <field name="ANGLE" type="text" output="OAct4"/>
45
46  </activity>
47  <activity tag="act5" description="" type="MAP" activation="./
48      experiment.cmd" template="%=WFDIR%/template_act5">
49      <relation reltype="Input" dependency="act4"/>
50      <relation reltype="Output" name="OAct5"/>
51      <field name="ID" type="float" output="OAct5"
52          decimalplaces="0"/>
53      <field name="TOTALTIME" type="text" output="OAct5"/>
54      <field name="ACTTIME" type="text" output="OAct5"/>
55      <field name="IMG1" type="text" output="OAct5"/>
56
57  </activity>
58
59  </conceptualWorkflow>
60  <executionWorkflow tag="workflow-map" execmodel="DYN_FAF" expdir="%
61      %=WFDIR%/exp">
62      <relation name="IAct1" filename="input.dataset"/>
63
64  </executionWorkflow>
65
66 </SciCumulus>

```

Un exemple plus concret d'un fichier de configuration du flux de travail est présenté dans l'Annexe F. Il s'agit d'un xml fait pour un exemple présenté dans le Chapitre 3 : *Canny filter*

avec le paradigme de programmation GPGPU *MapReduce*.

# Chapitre 3

## Étude expérimentale - Combinaison Chiron/SPOC

Dans ce chapitre nous illustrons quelques exemples sur l'exécution parallèle des flux du travail scientifiques. On commence par un test simple pour valider l'approche de combinaison Chiron/SPOC et ensuite on focalise sur un test avancé où on valide et compare cette combinaison selon différents paradigmes de programmation GPGPU. Les tests sont effectués sur une machine Ubuntu Linux 14.04 64-bit avec un Intel(R) Xeon(R) CPU E3-1245 @ 3.30GHz, 8GB de RAM et un GPU Nvidia Quadro 1000M.

### 3.1 Exemple arithmétique

On commence par un exemple simple pour valider la combinaison entre Chiron et Spoc. Il s'agit d'une opération d'addition entre les contenus de fichiers d'un format csv représentant des données tabulaires.

Cet exemple montre l'utilité de Chiron d'automatiser des opérations de calcul sur un grand nombre de fichiers. Pour bien expliquer la fonctionnalité et l'enchaînement de l'exécution, j'ai choisi de détailler les différents fichiers présents dans l'exemple.

- **Les configurations des données**

Le fichier `input.dataset` organise le flux de données afin de le traiter par Chiron. C'est un fichier de format "`csv`" qui contient trois colonnes `ID`, `FILE1`, `FILE2`. L'identifiant `ID` indique un flux de travail à appliquer sur les champs suivants. Les champs `FILE1` et `FILE2` contiennent les chemins des fichiers des données à traiter.

- **L'invocation d'un programme SPOC**

Pour chaque activité existe un fichier de modèle `extractor.cmd`, qui fait le lien avec Chiron en invoquant le programme Spoc implémenté.

- **La configuration de flux de travail**

Le fichier XML `Chiron.xml` décrit les détails conceptuels et les détails de l'exécution de flux de travail, voir Figure 3.1.

```
<Chiron>
    <environment verbose="true"/>
    <constraint workflow_execetag="workflow-query-1" cores="1"/>
    <workspace workflow_dir="/home/racha/Documents/stage/Spoc_Workflows/Files_Workflow"/>
    <database name="chiron" username="chiron" password="chiron" port="5433" server="localhost"/>
    <conceptualWorkflow tag="workflow-query" description="">

        <Activity tag="Ex3Act1" description="" type="Filter" activation=".experiment.cmd;./extractor.cmd"
            template="%=WFDIR%/template_act1">
            <relation reltype="Input" name="IAct1"/>
            <relation reltype="Output" name="OAct1" />
            <field name="ID" type="float" input="IAct1" output="OAct1" decimalplaces="0"/>
            <field name="FILE1" type="file" input="IAct1" output="OAct1">
                <field name="V" type="float" output="OAct1" decimalplaces="0"/>
            </field>
            <field name="FILE2" type="file" input="IAct1" output="OAct1">
                <field name="K" type="float" output="OAct1" decimalplaces="0"/>
            </field>
            <field name="FILE3" type="file" output="OAct1"/>
        </Activity>

    </conceptualWorkflow>

    <executionWorkflow tag="workflow-query" execmodel="DYN_FAF" expdir="%=WFDIR%/exp">
        <relation name="IAct1" filename="input.dataset"/>
    </executionWorkflow>
</Chiron>
```

FIGURE 3.1 – Le fichier de configuration du workflow

Comme décrit dans la Section 2.1.1, la balise `ConceptualWorkflow` définit le comportement de chaque composant dans un flux de travail. La balise `database` transmet à la base de données les informations de flux de travail. La balise `ExecutionWorkflow` est chargée d'exécuter une instance de flux de travail conceptuel. De telle façon, tous les paramètres nécessaires sont spécifiés dans ce fichier xml. Le flux de travail de cet exemple est constitué d'une seule activité de deux relations

- Input relation "iact1" : la relation `iact1` est la relation d'entrée de l'activité `act1`; `ID`, `FILE1`, `FILE2`.
- Output relation "oact1" : la relation `oact1` est la relation produite par l'activité `act1`; `ID`, `FILE1`, `FILE2`, `FILE3`.

rappelons que la description des différents éléments XML avec les valeurs de réglage est présentée en détails dans l'Annexe B.

- **L'exécutable de l'activité**

Le programme de Spoc `Add1_vecteurs.byte` prend en arguments `ID`, `FILE1`, `FILE2`.

Il applique une opération d'addition entre la colonne V de fichier FILE1 et la colonne K de fichier FILE2, et met le résultat dans un fichier de sortie. Et Il ajoute dans la relation de sortie une colonne FILE3 qui contiennent les chemins de fichiers de sorties.

- **La provenance des données**

Dans la base de données, pour "File\_Workflow" on trouve deux tables Iact1 et Oact1. La table Iact1 contient les champs de la relation d'entrée. La table Oact1 contient les champs de la relation de sortie, on trouve la colonne FILE3 qui contient les chemins des fichiers de sortie.

## 3.2 Exemple de traitement d'image

Dans cet exemple nous avons choisi d'appliquer le filtre de Canny [11, 5]. Nous avons opté ce choix car il illustre bien le flux de travail avec plusieurs activités qui s'enchainent d'une façon que la sortie d'une activité est une entrée d'une autre.

### 3.2.1 Détection des contours : *Canny filter*

Le filtre de Canny est utilisé en traitement d'images pour la détection des contours [11, 5]. L'entrée est une image grise et la sortie est une image en noir et blanc avec une ligne blanche d'une largeur d'un pixel indiquant les bords. Un bord peut être défini comme un lieu de contraste élevé. L'algorithme de Canny est conçu pour être optimal suivant trois critères clairement explicités :

- Bonne détection : faible taux d'erreur dans la signalisation des contours.
- Bonne localisation : minimisation des distances entre les contours détectés et les contours réels.
- Clarté de la réponse : une seule réponse par contour et pas de faux positifs.

Il comporte quatre étapes : (i) la réduction du bruit, (ii) le gradient d'intensité et la direction des contours (iii) la suppression des non-maxima (iv) le seuillage des contours. On a implémenté ces étapes, respectivement, par les filtres suivants [10, 17] : (I) flou gaussien, (II) filtrage de Sobel, (III) suppression non maximale, (IV) le seuillage Hystérésis. Dans la suite on décrit brièvement chaque étape.

### 3.2.2 Les étapes de filtrage

Comme mentionné précédemment le filtre de Canny est réalisé en quatre étapes.

**Flou Gaussien** Dans la première étape un flou gaussien est effectué pour réduire le bruit de l'image originale avant d'en détecter les contours. Cela est nécessaire parce que, généralement, le bruit est un contraste élevé et conduirait donc à des faux positifs. Le filtre est implémenté en

utilisant la convolution de l'image. La convolution de l'image est une opération qui remplace essentiellement chaque pixel avec une moyenne pondérée de ses voisins.

**Filtre Sobel et direction des contours** Après le filtrage, l'étape suivante est d'appliquer un gradient qui retourne l'intensité des contours. Le filtrage de Sobel remplace chaque pixel par une combinaison des approximations des gardiens horizontaux (direction  $x$ ) et verticaux (direction  $y$ ) des pixels voisins. Par conséquence, les pixels dans les zones de contraste élevé seront plus brillants que les pixels dans les zones de contraste faible. Ceci trouve essentiellement les zones où les bords sont plus susceptibles d'exister, mais il ne repère pas précisément où les bords se trouvent. Comme pour le flou gaussien, cela se fait en utilisant l'image convolution, mais la convolution est effectuée sur deux fois : une fois pour la dérivée en  $x$  et une fois pour la dérivée en  $y$ . Le pixel est alors remplacé par :  $\{(d_i/dx)^2 + (d_i/dy)^2\}^{1/2}$  où  $d_i$  représente le changement d'intensité. Au cours de cette étape, la direction du gradient est également calculée pour chaque pixel qui est nécessaire pour une suppression non maximale. Les orientations des contours sont déterminées par la formule :

$$\theta = \pm \arctan \left( \frac{G_y}{G_x} \right)$$

Nous obtenons finalement une carte des gradients d'intensité en chaque point de l'image accompagnée des directions des contours.

**Suppression des non-maxima** La carte des gradients obtenue précédemment fournit une intensité en chaque point de l'image. Une forte intensité indique une forte probabilité d'une présence de contour. Toutefois, cette intensité ne suffit pas à décider si un point correspond à un contour ou pas. Seuls les points correspondants à des maxima locaux sont considérés comme des correspondants à des contours, et ils sont conservés pour la prochaine étape de la détection. Un maximum local est présent sur les extrêmes du gradient, c'est-à-dire là où sa dérivée s'annule. La suppression non-maxima exclut les pixels qui font partie d'un bord, mais ne définissent pas le bord. Le résultat est que nous condensons ces bords d'un gradient important en une seule ligne d'un seul pixel. Notez que le résultat produit n'est toujours pas le produit final, ces lignes ne sont pas purement blanches et ne signifient pas nécessairement un bord, elles représentent encore la probabilité d'un bord. La suppression du non-maxima est effectuée en utilisant la direction du gradient trouvée à l'étape précédente et en comparant le pixel courant avec des pixels voisins de chaque côté. Si le pixel est plus faible en intensité que le seuil de ces pixels voisins, alors il n'est pas considéré comme bord, et sa valeur sera remplacée par 0. Si le pixel est d'une intensité supérieure de ses voisins dans la direction du gradient, il peut être sur le bord, et sa valeur sera conservée.

**Seuillage des contours** La différenciation des contours sur la carte générée se fait par seuillage à *Hysteresis*. Cela nécessite deux seuils, un haut et un bas, seront comparés à l'intensité du gradient de chaque point. Le critère de décision est le suivant : pour chaque point, si l'intensité de son gradient est inférieure au seuil bas, le point est rejeté, et si elle est supérieure au seuil haut, le point est accepté comme formant un contour. Entre le seuil bas et le seuil haut, le point est accepté s'il est connecté à un point déjà accepté. Une fois ceci réalisé, l'image obtenue est binaire avec les pixels appartenant aux contours en blanc et les autres pixels en noir.

Dans la suite on présent les résultats de l'implémentation de l'algorithme de *Canny filter* en combinant Chiron et Spoc. Ensuite une implémentation modifiée qui utilise SAREK (Section 2.2.2), une implémentation basé sur une version MapReduce, et finalement une implémentation CPU afin de comparer le temps d'exécution entre les différentes versions et voir le gain obtenu par rapport à cette version CPU.

### 3.2.3 *Canny filter* via Chiron et SPOC

Dans la Section 3.2.1 on a défini le filtre de Canny et on a expliqué qu'il prend en entrée une image grise et qu'il s'applique en quatre étapes avant de produire l'image de sortie. Dans ce travail, on a ajouté alors un filtre en plus qui est appliqué avant le filtre de Canny afin de transformer une image d'entrée normale à une image grise qui sera prête à passer en entrée pour le filtre de Canny.

#### 3.2.3.1 Le flux de travail

La structure de flux de travail est précisée dans un fichier XML comme expliqué dans l'exemple précédent, Section 3.1. Le flux de travail est constitué de cinq tâches, chaque tâche correspond à un filtre, un filtre gray et ensuite les quatre filtres qui composent le filtre de Canny.

#### 3.2.3.2 Implémentation de base

On commence à tester la combinaison Chiron/SPOC avec une implémentation basique GPGPU des différents filtre de flux de travail. On a appliqué les différents filtres sur plusieurs images d'une taille variable et les temps d'exécution sont collectés dans la table<sup>1</sup> 3.1.

Les images résultants après chaque étape de filtrage pour l'IMG1 sont présentées dans la Table 3.2. L'application des différents filtres sur d'autres images se trouve dans l'Annexe C.

---

1. Les tests sont effectués sur une carte graphique NVIDIA Quadro 1000M

Image	Taille	Gray	Gaussian	Sobel	Non-max	Hysteresis
IMG1	185850	1.11	0.42	0.71	1.21	0.42
IMG2	786432	1.62	1.12	1.26	2.54	2.18
IMG3	4194304	12.67	9.63	11.56	13.13	9.17
IMG4	11935651	26.82	20.43	21.33	29.14	20.55
IMG5	14441362	28.66	22.12	25.53	33.75	22.01

TABLE 3.1 – Temps de calcul en seconds, Section 3.2.3.2

### 3.2.3.3 Implémentation SAREK

Rappelons que SAREK (Stream ARchitecture Extensible Kernels) est une solution offerte par SPOC consiste à un langage intégré à OCaml, dédié à la description des noyaux de calcul. Afin d'explorer cette solution on a réalisé les deux filtres *gray filter* et *Hysteresis filter* en SAREK, et on propose dans la suite, Tableau 3.3, une comparaison de performance en terme du temps entre une version écrite avec kernel externe et une version écrite avec le langage SAREK. Tableau 3.3 montre une faible différence en temps de simulation avec un avantage légère pour la version basique de la Section 3.2.3.2. SAREK nous fournit donc une solution qui simplifie la programmation tout en garantissant la performance.

### 3.2.3.4 Implémentation MapReduce

Dans cette version de *Canny filter* notre but est de faire un traitement distribué et parallèle des données. Dans ce modèle d'exécution le flux de travail principal est subdivisé en un ensemble des flux de travail qui s'exécutent en parallèle. En partitionnant l'image principale, chaque partition est traité par un flux de travail qui s'exécute en parallèle.

Pour implémenter une version MapReduce du flux de travail de Canny. On ajoute deux activités. Une première activité dirigée par l'opérateur *SplitMap* découpe l'image en plusieurs sous-images. Cet opérateur *SplitMap* produit un set de tuples dans la relation de sortie pour chaque tuple Image consommé en entrée. La deuxième activité ajoutée est l'activité finale de ce flux de travail qui s'exécute en dernière. C'est une activité dirigée par l'opérateur *Reduce*. L'intérêt de ce modèle d'exécution est de gagner en performance en parallélisant et en distribuant le travail. Tableau 3.4 illustre les résultats obtenus avec cette version MapReduce et compare avec l'implémentation basique (Section 3.2.3.2). On présent le temps d'exécution GPU consommé seulement sur les étapes de filtrage (colonne MapReduce (filtrage)) ainsi que le temps total c-à-d, le filtrage (GPU) avec le découpage (CPU) en 4 image et l'assemblage (CPU), présenté dans la colonne MapReduce (total). On remarque un gain important en temps de simulation avec l'approche MapReduce.

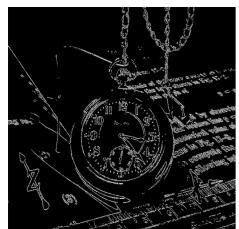
Input	Activity	Output
	act1 gray filtre ⇒	
	act2 Gaussian filtre ⇒	
	act3 Sobel filtre ⇒	
	act4 Non-max filtre ⇒	
	act5 Hysteris filtre ⇒	

TABLE 3.2 – Le flux de travail de Canny

Image	Taille	Temps Chiron/SPOC	Temps SAREK
IMG1	185850	4 s	4 s
IMG2	786432	14 s	15 s
IMG3	4194304	58 s	61 s
IMG4	11935651	154 s	159 s
IMG5	14441362	185 s	192 s

TABLE 3.3 – Temps de calcul en seconds, Section 3.2.3.3

Image	Taille	Chiron/SPOC	MapReduce (filtrage)	MapReduce (total)
IMG1	185850	4 s	2 s	3 s
IMG2	786432	14 s	8 s	10 s
IMG3	4194304	58 s	35 s	43 s
IMG4	11935651	154 s	93 s	117 s
IMG5	14441362	185 s	113 s	142 s

TABLE 3.4 – Temps de calcul en seconds, Section 3.2.3.4

### 3.2.4 Canny filter via une implémentation CPU

Après avoir implémenter différents paradigmes de programmation GPGPU on a pensé à implémenter une version CPU toujours avec le langage OCaml avant de faire une comparaison globale et consulter le gain de performance obtenu avec les différents approches. Tableau 3.5 collecte les temps de simulations consommés lors de l'application des filtres sur les différentes images tests. On compare toujours avec notre première implémentation de la Section 3.2.3.2 où on remarque le gain important déjà prévu.

Image	Taille	Chiron/SPOC	Chiron/SPOC with MapReduce	CPU
IMG1	185850	4 s	3 s	6 s
IMG2	786432	14 s	10 s	24 s
IMG3	4194304	58 s	43 s	120 s
IMG4	11935651	154 s	117 s	333 s
IMG5	14441362	185 s	142 s	400 s

TABLE 3.5 – Temps de calcul en seconds, Section 3.2.4

Figure 3.2 illustre les différents résultats avec tous les implémentations précédentes et compare le facteur de gain obtenu ou on peut remarquer que l'implémentation MapReduce est la plus performante avec un facteur de gain autour de 3.

Les différents tests de performances effectués illustrent la combinaison entre Chiron et SPOC et montrent qu'on peut avoir un gain significative par rapport à l'implémentation CPU. Lorsque on est basé sur une version MapReduce on arrive à un facteur de gain autour

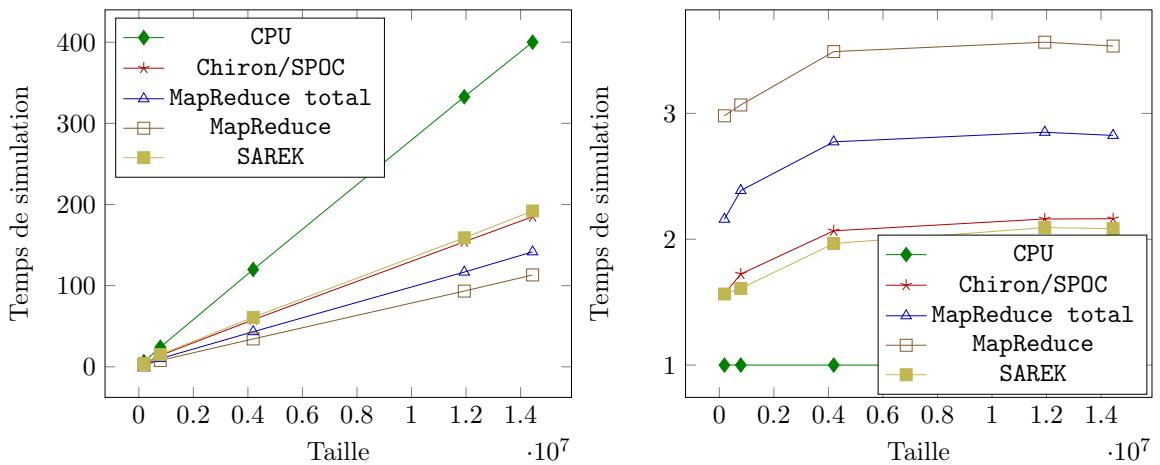


FIGURE 3.2 – Temps d'exécution en secondes (à gauche), Facteur de gain (à droite) : *Canny filter*

de 3 en coupant l'image en 4. On estime alors un gain plus important si on coupe plus l'image avec l'approche MapReduce. Plus de tests sont demandés pour confirmer.

Via les exemples précédents on remarque qu'on peut simplement combiner entre Chiron et SPOC et alors gérer des flux de travail et les lancés facilement sur GPU avec la possibilité d'implémenter différents paradigmes de programmation GPGPU. Les premières tests et comparaisons de temps d'exécution avec l'implémentation CPU montre un gain significatif de performance. Les kernels implémentés et quelques résultats supplémentaires peuvent se trouver dans l'Annexe D et E.



## Conclusion

Dans mon stage intitulé « Exécution parallèle sur cartes graphiques d'un système de gestion de flux de travail de calculs numériques », j'ai étudié le système de gestion de flux de travail Chiron. J'ai analysé le modèle de l'exécution dans Chiron et réalisé des flux de travail sur Chiron dont les activations sont écrites comme des scripts en langage shell. Ensuite j'ai étudié SPOC, l'extension à OCaml qui permet la déclaration des noyaux GPGPU externes utilisables depuis un programme OCaml. J'ai testé SPOC et réalisé des programmes en OCaml avant de combiner Chiron et SPOC. J'ai concrétisé des études expérimentales pour illustrer et évaluer la performance de cette combinaison. J'ai focalisé les comparaisons sur un flux de travail de traitement d'images qui s'effectue en plusieurs étapes : "The canny filter". Les comparaisons sont faites avec une implémentation CPU et différents paradigmes de programmation GPGPU. Les résultats montrent un gain de performance remarquable surtout lorsqu'on fait un traitement distribué et parallèle des données réalisé dans la version MapReduce. Plus précisément lorsque le flux de travail principal est subdivisé en un ensemble des flux de travail qui s'exécutent en parallèle.

A la fin de mon stage j'ai analysé plus profondément les codes sources de Chiron à fin d'ajouter un environnement GPGPU spécifique pour Chiron pour les traitements des flux de travail qui prend en considération le comportement de la combinaison fait et que j'estime améliorer (limiter les transferts, envoyer plusieurs flux de travail en même temps selon la disponibilité de GPU, ...etc). Mais, cette partie n'est pas finalisée.



## Logiciels

Durant ce stage on s'appuie principalement sur deux logiciels :

### A.1 Chiron <http://chironengine.sourceforge.net/>

Chiron est un système de gestion de flux de travail de calculs numériques (*scientific workflow management system*), il exécute ces simulations comme une chaîne d'activités (programmes) et un flux de données (dataflow) sur ces activités. Ce système fournit la gestion des simulations scientifiques, leur exécution parallèle tout en enregistrant la provenance des données. Chiron implémente l'approche algébrique dans un style MapReduce. L'utilisation de MapReduce comme approche de programmation permet aux scientifiques de programmer d'une façon plus simple la procédure du calcul en cachant le parallélisme, qui peut être complexe à gérer [15].

### A.2 SPOC <http://www.algo-prog.info/s poc/>

. SPOC (Stream Processing with OCaml) consiste en une extension à OCaml associée à une bibliothèque d'exécution. L'extension permet la déclaration de noyaux GPGPU externes utilisables depuis un programme OCaml, tandis que la bibliothèque permet de manipuler ces noyaux ainsi que l'automatisation des transferts de données nécessaires à leur exécution. SPOC offre, de plus, une abstraction supplémentaire en unifiant les deux environnements de développement GPGPU (Cuda et OpenCL) en une même bibliothèque [1, 2].



## Fichier de configuration de flux de travail XML

- Database name : name of the created database in PostgreSQL relational database system.
- Server : The name of the configured server.
- Port : The opened port to server.
- Username : The username to access database.
- Password : The password to access database with the username informed.
- Workflow identification : The name of workflow that needs to be unique. It is used to select this conceptual workflow in future queries.
- Workflow description : The description about this workflow.
- Activity identification : The name of activity needs to be unique in this workflow definition. It is used to select this activity in future queries.
- Activity description : The description about this activity.
- Operator : The name of operator inspired on relation algebra. Each operator presents a specific behavior, according the consumption and production of relations. This parameter can assumes the following values : MAP, SPLIT\_MAP, FILTER, REDUCE and MR\_QUERY. [1] [2]
- Activation : The command line to be executed by this activity.
- Template directory : Directory with necessary files to execute this activity.
- Relation name : The name of relation. This value needs to be unique in this workflow definition.
- Relation type : The relation can be of two types : Input and Output. The first value is responsible to consume value of each field, while the second value produces the values.
- Dependency : The activity identification of activity that relation depends.

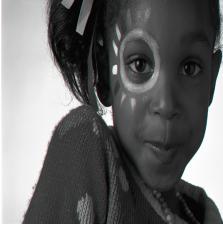
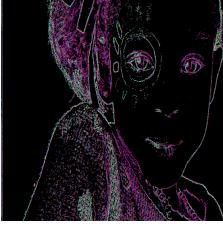
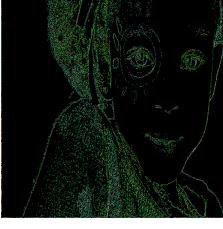
- Field name : The name of field. This value needs to be unique in an activity definition.
- Field type : The type of field can assume three types : float, string or file. The first value is a numeric value, which needs to inform attribute decimalplaces. The second value presents an array of characters. Also, the last value presents a file name, that needs the attribute instrumented assignment as true, if it is necessary substitute at least one tag in this file with default tags of Chiron.
- Input relation : The input relation name that consumes this field.
- Output relation : The output relation name that produces this field.
- Decimal places : It determines how many decimal places are used in field of type float.
- Instrumentation : In a field of type file, it is used to determine if the default tags of Chiron need to be substituted in this file. So, it can only assume a boolean value.
- Machine name : The name of machine, which executes the Chiron.
- Machine address : The address of machine.
- Machine username : The username to access machine.
- Machine password : The password to access machine.
- Execution model : The execution model of Chiron, that can assumes the following values : DYN\_FTF, DYN\_FAF, STA\_FTF and STA\_FAF. [3]
- Execution tag : The execution identification for this instance of a specific conceptual workflow. This value needs to be unique.
- Workflow directory : The path used by Chiron to execute workflow. Also, the template directories need to be found here.
- Experiment directory : The path used by Chiron to manipulate files. Also, the relation files need to be found here.
- Relation file name : It is the file name of an input relation of workflow. It is used to set up values for every fields of this relation.

Annexe

# C

Résultats : *Canny filter*

Input	Activity	Output
	act1 gray filtre ⇒	
	act2 Gaussian filtre ⇒	
	act3 Sobel filtre ⇒	
	act4 Non-max filtre ⇒	
	act5 Hysteris filtre ⇒	

Input	Activity	Output
	act1 gray filtre ⇒	
	act2 Gaussian filtre ⇒	
	act3 Sobel filtre ⇒	
	act4 Non-max filtre ⇒	
	act5 Hysteris filtre ⇒	



# Annexe D

## Les kernels de calcul : *Canny filter*

### D.1 Gaussian

---

```
1 #ifdef __cplusplus
2 extern "C" {
3 #endif
4 #define L_SIZE 16
5
6 __constant__ float gaus[3][3] = { {0, 1, 0},
7                                 {1, -3, 1},
8                                 {0, 1, 0} };
9
10
11 __global__ void gauss_kernel( int* data,
12                               int* out,
13                               int cols,int rows)
14 {
15     int g_row = blockIdx.x * blockDim.x + threadIdx.x;
16     int g_col = blockIdx.y * blockDim.y + threadIdx.y;
17     int pos = g_col * cols + g_row;
18
19     int l_row = threadIdx.x + 1;
20     int l_col = threadIdx.y + 1;
21
22     int sum=0;
```

```

24    __shared__ int l_data[L_SIZE+2][L_SIZE+2];
25
26    // copy to local
27    l_data[l_row][l_col] = data[pos];
28
29    // top most row
30    if (l_row == 1)
31    {
32        l_data[0][l_col] = data[pos-cols];
33        // top left
34        if (l_col == 1)
35            l_data[0][0] = data[pos-cols-1];
36
37        // top right
38        else if (l_col == L_SIZE)
39            l_data[0][L_SIZE+1] = data[pos-cols+1];
40    }
41    // bottom most row
42    else if (l_row == L_SIZE)
43    {
44        l_data[L_SIZE+1][l_col] = data[pos+cols];
45        // bottom left
46        if (l_col == 1)
47            l_data[L_SIZE+1][0] = data[pos+cols-1];
48
49        // bottom right
50        else if (l_col == L_SIZE)
51            l_data[L_SIZE+1][L_SIZE+1] = data[pos+cols+1];
52    }
53
54    if (l_col == 1)
55        l_data[l_row][0] = data[pos-1];
56    else if (l_col == L_SIZE)
57        l_data[l_row][L_SIZE+1] = data[pos+1];
58
59
60
61    for (int i = 0; i < 3; i++)
62        for (int j = 0; j < 3; j++)

```

---

```

63     sum += gaus[i][j] * l_data[i+l_row-1][j+l_col-1];
64
65
66 out[pos] = max(0,sum); ;
67
68     return;
69 }
70
71 #ifdef __cplusplus
72 }
73 #endif

```

---

## D.2 Sobel

---

```

1
2 #ifdef __cplusplus
3 extern "C" {
4 #endif
5     __constant__ int sobx[3][3] = { { -1, 0, 1 },
6                                     { -2, 0, 2 },
7                                     { -1, 0, 1 } };
8
9     __constant__ int soby[3][3] = { { -1,-2,-1 },
10                                 { 0, 0, 0 },
11                                 { 1, 2, 1 } };
12
13 // Sobel kernel. Apply sobx and soby separately, then find the sqrt of their
14 // squares.
15 // data : image input data with each pixel taking up 1 byte (8Bit 1Channel)
16 // out : image output data (8B1C)
17 // theta : angle output data
18 __global__ void sobel_kernel( int *data,
19                             int *out,
20                             float *theta,
21                             int rows,
22                             int cols)
23 { // Some of the available convolution kernels

```

```

24
25 // collect sums separately. we're storing them into floats because that
26 // is what hypot and atan2 will expect.
27 const float PI = 3.14159265;
28     int l_row = threadIdx.y + 1;
29     int l_col = threadIdx.x + 1;
30
31
32     int g_row = threadIdx.y + (blockIdx.y * blockDim.y);;
33     int g_col = threadIdx.x + (blockIdx.x * blockDim.x);;
34
35     int pos = g_row * cols + g_col;
36
37     __shared__ int l_data[18][18];
38
39 // copy to local
40     l_data[l_row][l_col] = data[pos];
41
42 // top most row
43     if (l_row == 1)
44     {
45         l_data[0][l_col] = data[pos - cols];
46         // top left
47         if (l_col == 1)
48             l_data[0][0] = data[pos - cols - 1];
49
50         // top right
51         else if (l_col == 16)
52             l_data[0][17] = data[pos - cols + 1];
53     }
54 // bottom most row
55     else if (l_row == 16)
56     {
57         l_data[17][l_col] = data[pos + cols];
58         // bottom left
59         if (l_col == 1)
60             l_data[17][0] = data[pos + cols - 1];
61
62         // bottom right

```

```

63         else if (l_col == 16)
64             l_data[17][17] = data[pos+cols+1];
65     }
66
67 // left
68 if (l_col == 1)
69     l_data[l_row][0] = data[pos-1];
// right
71 else if (l_col == 16)
72     l_data[l_row][17] = data[pos+1];
73
74
75 float sumx = 0, sumy = 0, angle = 0;
76 // find x and y derivatives
77 for (int i = 0; i < 3; i++)
78 {
79     for (int j = 0; j < 3; j++)
80     {
81         sumx += sobx[i][j] * l_data[i+l_row-1][j+l_col-1];
82         sumy += soby[i][j] * l_data[i+l_row-1][j+l_col-1];
83     }
84 }
85
86 // The output is now the square root of their squares, but they are
87 // constrained to 0 <= value <= 255. Note that hypot is a built in function
88 // defined as : hypot(x,y) = sqrt(x*x, y*y).
89 out[pos] = min(255,max(0, (int)hypot(sumx,sumy) ));
90
91 // Compute the direction angle theta in radians
92 // atan2 has a range of (-PI, PI) degrees
93 angle = atan2(sumy,sumx);
94
95 // If the angle is negative,
96 // shift the range to (0, 2PI) by adding 2PI to the angle,
97 // then perform modulo operation of 2PI
98 if (angle < 0)
99 {
100     angle = fmod((angle + 2*PI),(2*PI));
101 }
```

```

102
103     // Round the angle to one of four possibilities : 0, 45, 90, 135 degrees
104     // then store it in the theta buffer at the proper position
105     theta[pos] = ((int)(57.29577951 * (angle * (PI/8) + PI/8-0.0001) / 45) * 45) % 180;
106 }
107 #ifdef __cplusplus
108 }
109#endif

```

---

### D.3 Non max suppression

---

```

1
2 #ifdef __cplusplus
3 extern "C" {
4 #endif
5 // Non-maximum Supression Kernel
6 // data : image input data with each pixel taking up 1 byte (8Bit 1Channel)
7 // out : image output data (8B1C)
8 // theta : angle input data
9 __global__ void non_max_supp_kernel(int *data,
10                                     int *out,
11                                     int *theta,
12                                     int rows,
13                                     int cols)
14 {
15     // These variables are offset by one to avoid seg. fault errors
16     // As such, this kernel ignores the outside ring of pixels
17     int l_row = threadIdx.y + 1 ;
18     int l_col = threadIdx.x + 1;
19     int g_row = threadIdx.y + (blockIdx.y * blockDim.y);
20     int g_col = threadIdx.x + (blockIdx.x * blockDim.x);
21
22     int pos = g_row * cols + g_col;
23
24     __shared__ int l_data[18][18];
25
26     // copy to l data

```

```

27     l_data[l_row][l_col] = data[pos];
28
29     // top most row
30     if (l_row == 1)
31     {
32         l_data[0][l_col] = data[pos - cols];
33         // top left
34         if (l_col == 1)
35             l_data[0][0] = data[pos - cols - 1];
36
37         // top right
38         else if (l_col == 16)
39             l_data[0][17] = data[pos - cols + 1];
40     }
41     // bottom most row
42     else if (l_row == 16)
43     {
44         l_data[17][l_col] = data[pos + cols];
45         // bottom left
46         if (l_col == 1)
47             l_data[17][0] = data[pos + cols - 1];
48
49         // bottom right
50         else if (l_col == 16)
51             l_data[17][17] = data[pos + cols + 1];
52     }
53
54     if (l_col == 1)
55         l_data[l_row][0] = data[pos - 1];
56     else if (l_col == 16)
57         l_data[l_row][17] = data[pos + 1];
58
59     int my_magnitude = l_data[l_row][l_col];
60
61     // The following variables are used to address the matrices more easily
62     switch (theta[pos])
63     {
64         // A gradient angle of 0 degrees = an edge that is North/South
65         // Check neighbors to the East and West

```

```

66     case 0:
67         // suppress me if my neighbor has larger magnitude
68         if (my_magnitude <= l_data[l_row][l_col+1] || // east
69             my_magnitude <= l_data[l_row][l_col-1]) // west
70         {
71             out[pos] = 0;
72         }
73         // otherwise, copy my value to the output buffer
74         else
75         {
76             out[pos] = my_magnitude;
77         }
78         break;

79
80     // A gradient angle of 45 degrees = an edge that is NW/SE
81     // Check neighbors to the NE and SW
82     case 45:
83         // suppress me if my neighbor has larger magnitude
84         if (my_magnitude <= l_data[l_row-1][l_col+1] || // north east
85             my_magnitude <= l_data[l_row+1][l_col-1]) // south west
86         {
87             out[pos] = 0;
88         }
89         // otherwise, copy my value to the output buffer
90         else
91         {
92             out[pos] = my_magnitude;
93         }
94         break;

95
96     // A gradient angle of 90 degrees = an edge that is E/W
97     // Check neighbors to the North and South
98     case 90:
99         // suppress me if my neighbor has larger magnitude
100        if (my_magnitude <= l_data[l_row-1][l_col] || // north
101            my_magnitude <= l_data[l_row+1][l_col]) // south
102        {
103            out[pos] = 0;
104        }

```

```

105         // otherwise, copy my value to the output buffer
106     else
107     {
108         out[pos] = my_magnitude;
109     }
110     break;
111
112     // A gradient angle of 135 degrees = an edge that is NE/SW
113     // Check neighbors to the NW and SE
114     case 135:
115         // suppress me if my neighbor has larger magnitude
116         if (my_magnitude <= l_data[l_row-1][l_col-1] || // north west
117             my_magnitude <= l_data[l_row+1][l_col+1]) // south east
118         {
119             out[pos] = 0;
120         }
121         // otherwise, copy my value to the output buffer
122         else
123         {
124             out[pos] = my_magnitude;
125         }
126         break;
127
128     default:
129         out[pos] = my_magnitude;
130         break;
131     }
132 }
133 #ifdef __cplusplus
134 }
135#endif

```

---

## D.4 Hysterises

---

```

1
2 #ifdef __cplusplus
3 extern "C" {

```

```
4  #endif
5
6  __global__ void hys_kernel( int* data,
7          int* out,
8          int rows,
9          int cols)
10
11 { float lowThresh = 60;
12     float highThresh = 170;
13
14     int g_row = threadIdx.y + (blockIdx.y * blockDim.y);
15     int g_col = threadIdx.x + (blockIdx.x * blockDim.x);
16     int pos = g_col * cols + g_row;
17
18
19
20     const int EDGE = 16777215;
21
22     int magnitude = data[pos];
23
24     if (magnitude >= highThresh)
25         out[pos] = EDGE;
26     else if (magnitude <= lowThresh)
27         out[pos] = EDGE;
28     else
29     {
30
31         float med = (highThresh + lowThresh)/2;
32
33         if (magnitude >= med)
34             out[pos] = EDGE;
35         else
36             out[pos] = 0;
37     }
38 }
39 #ifdef __cplusplus
40 }
41#endif
```

---

# Annexe E

## L'implémentation de Gray et Hystereses en SAREK : *Canny filter*

### E.1 Gray

---

```
1
2 #use "readppm.ml";
3 open Spoc
4 open Images
5 open Kirc
6
7 let devices = Spoc.Devices.init ()
8
9 open Kirc
10
11 let gpu_to_gray = kern v w h->
12   let open Std in
13   let tid = thread_idx_x + block_dim_x * block_idx_x in
14   if tid <= (w*h) then (
15     let i = (tid*3) in
16     let res = int_of_float ((0.21 *. (float (v.[<i>]))) +
17                           (0.71 *. (float (v.[<i+1>]))) +
18                           (0.07 *. (float (v.[<i+2>])))) in
19     v.[<i>] <- res;
20     v.[<i+1>] <- res;
21     v.[<i+2>] <- res )
```

```

22
23
24 let measure_time s f =
25   let t0 = Unix.gettimeofday () in
26   let a = f () in
27   let t1 = Unix.gettimeofday () in
28   Printf.printf "Time %s : %Fs\n%" s (t1 -. t0);
29   a;;
30
31 let dev = ref devices.(1)
32 let auto_transfers = ref false
33 let verify = ref true
34 let files = ref []
35
36 let start = Unix.time ()
37
38
39 let _ =
40   Arg.parse [] (fun s -> files := s::!files) "";
41   Random.self_init();
42   let args = List.rev !files in
43   let id, file1 = match args with
44     | [id; file1] -> id, file1
45     | _ -> failwith "args error"
46   in
47
48
49 let img = load_ppm file1 in
50
51
52
53 let a = Spoc.Vector.create Vector.int32 (img.Rgb24.height * img.Rgb24.width * 3) in
54 Printf.printf "Allocating Vectors (on CPU memory ) %d\n" (Spoc.Vector.length a -
55   1);
56 flush stdout;
57
58 let height = img.Rgb24.height in
59 let width = img.Rgb24.width in
60 let f = ref 0 in

```

```

60
61   for i=0 to height-1 do
62     for j=0 to width-1 do
63       let color = Rgb24.get img j i in
64       a.[<!f>] <- (Int32.of_int (color.r)) ;
65       f := !f+1;
66       a.[<!f>] <- (Int32.of_int (color.g)) ;
67       f := !f+1;
68       a.[<!f>] <- (Int32.of_int (color.b)) ;
69       f := !f+1;
70     done;
71   done;
72
73 begin
74   Printf.printf "Computing \n";
75   flush stdout;
76
77
78 let threadsPerBlock = match !dev.Devices.specfic_info with
79   | Devices.OpenCLInfo cli ->
80     (match cli.Devices.device_type with
81      | Devices.CL_DEVICE_TYPE_CPU -> 1
82      | _ -> 256)
83    | _ -> 256 in
84 let blocksPerGrid =
85   ((img.Rgb24.height * img.Rgb24.width) + threadsPerBlock -1) / threadsPerBlock
86 in
87 let block = { Spoc.Kernel.blockX = threadsPerBlock; Spoc.Kernel.blockY = 1 ;
88               Spoc.Kernel.blockZ = 1;} in
89 let grid = { Spoc.Kernel.gridX = blocksPerGrid; Spoc.Kernel.gridY = 1 ; Spoc.
90               Kernel.gridZ = 1;} in
91
92 ignore(Kirc.gen ~only:Devices.OpenCL
93         gpu_to_gray);
94 measure_time """
95   (fun () -> Kirc.run gpu_to_gray (a, img.Rgb24.height , img.Rgb24.width) (block,
96                                grid) 0 !dev);

```

```

96     Printf.printf "Fin\n";
97
98     let t1 = Unix.time () in
99
100    let curDir = Sys.getcwd () in
101    let dir = List.nth (Str.split (Str.regexp "/bin/") curDir) 0 in
102    let path = dir^"/Output/"^id in
103    Unix.mkdir path 0o777;
104    let sortie = path^"/Gray.ppm" in
105
106        let oc = open_out sortie in
107            Printf.fprintf oc "P6\n";
108            Printf.fprintf oc "%d %d \n" width height ;
109            Printf.fprintf oc "%d\n" 255;
110
111        for i = 0 to Vector.length a - 1 do
112            let c = Int32.to_int a.[<i>] in
113            output_byte oc c;
114        done;
115        close_out oc;
116
117
118    let oc = open_out "Erelation.txt" in
119        Printf.fprintf oc "ID;START;ACTTIME;IMG1\n";
120        Printf.fprintf oc "%s;" id;
121        Printf.fprintf oc "%F;" start;
122        Printf.fprintf oc "%F;" (t1 -. start);
123        Printf.fprintf oc "%s\n" sortie;
124        close_out oc;
125
126 end;

```

---

## E.2 Hysterises

---

```

1
2 #use "areadppm.ml";
3 open Spoc

```

```
4  open Images
5  open Kirc
6
7
8  let read_ascii_24 c =
9    (c.r * 256 + c.g) * 256 + c.b;
10 ;;
11
12 let color_to_rgb c =
13   let r = c / 65536 and g = c / 256 mod 256 and b = c mod 256 in
14   (r,g,b)
15 ;;
16
17 (*allouer Tresh avant sur le GPU!*)
18 let devices = Spoc.Devices.init ()
19
20 let gpu_hys = kern v w h ->
21   let lowThresh = 170 in
22   let highThresh = 90 in
23   let med = (highThresh + lowThresh)/2 in
24   let edge = 16777215 in
25   let open Std in
26   let tid = thread_idx_x + block_dim_x * block_idx_x in
27   if tid <= (w*h) then (
28     let i = (tid*3) in
29
30     let magnitude = (((v.[<i>] * 256 ) + (v.[<i+1>])) * 256 + (v.[<i+2>])) in
31     if (magnitude >= med) then
32
33       let r = edge / 65536 in
34       let g = edge / 256 mod 256 in
35       let b = edge mod 256 in
36
37       v.[<i>] <- r;
38       v.[<i+1>] <- g;
39       v.[<i+2>] <- b
40
41     else
42       v.[<i>] <- 0;
```

```

43     v.[<i+1>] <-0;
44     v.[<i+2>] <-0 )
45
46 let measure_time s f =
47   let t0 = Unix.gettimeofday () in
48   let a = f () in
49   let t1 = Unix.gettimeofday () in
50   Printf.printf "Time %s : %Fs\n%" s (t1 -. t0);
51   a;;
52
53
54 let dev = ref devices.(1)
55 let auto_transfers = ref false
56 let verify = ref true
57
58 let start = Unix.time ()
59
60 let files = ref [] in
61 Arg.parse ([])(fun s -> files := s::!files ) "";
62 Random.self_init();
63 let args = List.rev !files in
64 let id, file1= match args with
65   | [id; file1] -> id, file1
66   | _ -> failwith "args error"
67 in
68
69
70 let img = load_ppm file1 in
71 let height = img.Rgb24.height in
72 let width = img.Rgb24.width in
73
74
75 let a = Spoc.Vector.create Vector.int32 (img.Rgb24.height * img.Rgb24.width * 3) in
76 Printf.printf "Allocating Vector (on CPU memory ) %d\n" (Spoc.Vector.length a -
77   1);
78 flush stdout;
79
80 let f = ref 0 in
81 for i=0 to height-1 do

```

```

81   for j=0 to width-1 do
82     let color = Rgb24.get img j i in
83     a.[<!f>] <- (Int32.of_int (color.r)) ;
84     f := !f+1;
85     a.[<!f>] <- (Int32.of_int (color.g)) ;
86     f := !f+1;
87     a.[<!f>] <- (Int32.of_int (color.b)) ;
88     f := !f+1;
89   done;
90 done;
91
92 let threadsPerBlock = match !dev.Devices.specific_info with
93 | Devices.OpenCLInfo cli ->
94   (match cli.Devices.device_type with
95   | Devices.CL_DEVICE_TYPE_CPU -> 1
96   | _ -> 256)
97 | _ -> 256 in
98 let blocksPerGrid =
99   ((img.Rgb24.height * img.Rgb24.width) + threadsPerBlock -1) / threadsPerBlock
100 in
101 let block = { Spoc.Kernel.blockX = threadsPerBlock; Spoc.Kernel.blockY = 1 ; Spoc.
102   Kernel.blockZ = 1;} in
103 let grid = { Spoc.Kernel.gridX = blocksPerGrid; Spoc.Kernel.gridY = 1 ; Spoc.
104   Kernel.gridZ = 1;} in
105 ignore(Kirc.gen ~only:Devices.OpenCL
106           gpu_hys);
107 measure_time @@
108   (fun () -> Kirc.run gpu_hys (a, img.Rgb24.height , img.Rgb24.width) (block,grid) 0
109     !dev);
110
111
112
113 let t1 = Unix.time () in
114
115 (* let list = Str.split (Str.regexp "Non-max") file1 in
116   let name, ext= match list with

```

```

117      | [name; ext] -> name, ext
118      | _ -> failwith " error "
119      in
120      let sortie = name^"Hysteresis.ppm" in
121      *)
122      let sortie = "Hysteresis.ppm" in
123      let oc = open_out sortie in
124      Printffprintf oc "P6\n";
125      Printffprintf oc "%d %d \n" (width) (height);
126      Printffprintf oc "%d\n" 255;
127
128
129      for i = 0 to Vector.length a - 1 do
130          let c = Int32.to_int a.[<i>] in
131          output_byte oc c;
132      done;
133      close_out oc;
134
135      let oc = open_out "Erelation.txt" in
136      Printffprintf oc "ID;TOTALTIME;ACTTIME;IMG1;\n";
137      Printffprintf oc "%s;" id;
138      (* Printffprintf oc "%F;" (t1 -. float_of_string(st));*)
139      Printffprintf oc "%F;" (t1 -. start);
140      Printffprintf oc "%s;" sortie;
141
142      close_out oc;

```

---

Annexe F

Le fichier de configuration du workflow :  
*Canny filter* avec le paradigme de  
programmation GPGPU *MapReduce*

---

```
1 <SciCumulus>
2     <environment type="LOCAL" verbose="true"/>
3     <constraint workflow_execetag="workflow-new" cores="2" performance="false"/>
4     <workspace workflow_dir="/home/racha/Documents/stage/
      workflow_Canny_MapReduce"/>
5 <database name="scc2" username="scc2" password="scc2" port="5433" server="localhost"/>
6     <conceptualWorkflow tag="workflow_new" description="">
7
8         <activity constrained="false" tag="divide_act" description="" type="SPLIT_MAP" activation="../experiment.cmd" template="%=WFDIR%/Divide_template">
9             <relation reltype="Input" name="IAct1"/>
10            <relation reltype="Output" name="OAct1" />
11
12            <field name="ID" type="float" input="IAct1" output="OAct1" decimalplaces="0"/>
13            <field name="START" type="text" output="OAct1"/>
14            <field name="ACTTIME" type="text" output="OAct1"/>
```

```

15      <field name="IMG1" type="text" input="IAct1" output="OAct1"/>
16
17  </activity>
18
19      <activity tag="gray_act" description="" type="MAP" activation=
20        "./experiment.cmd" template="%=WFDIR%/Gray_template">
21          <relation reltype="Input" dependency="divide_act"/>
22          <relation reltype="Output" name="OAct2" output="OAct1"
23            />
24
25          <field name="ID" type="float" output="OAct2"
26            decimalplaces="0"/>
27          <field name="START" type="text" output="OAct2"/>
28          <field name="ACTTIME" type="text" output="OAct2"/>
29          <field name="IMG1" type="text" output="OAct2"/>
30
31  </activity>
32
33
34
35
36
37
38      <activity tag="gaussian_act" description="" type="MAP"
39        activation="./experiment.cmd" template="%=WFDIR%/
40        Gaussian_template">
41          <relation reltype="Input" dependency="gray_act"/>
42          <relation reltype="Output" name="OAct3" output="OAct2"
43            />
44          <field name="ID" type="float" output="OAct3"
45            decimalplaces="0"/>
46          <field name="START" type="text" output="OAct3"/>
47          <field name="ACTTIME" type="text" output="OAct3"/>
48          <field name="IMG1" type="text" output="OAct3"/>
49
50
51
52
53
54
55
56
57
58      <activity tag="sobel_act" description="" type="MAP" activation
59        ="./experiment.cmd" template="%=WFDIR%/Sobel_template">
60          <relation reltype="Input" dependency="gaussian_act"/>
61          <relation reltype="Output" name="OAct4"/>
62
63
64          <field name="ID" type="float" output="OAct4"
65            decimalplaces="0"/>
66          <field name="START" type="text" output="OAct4"/>

```

---

```

44         <field name="ACTTIME" type="text" output="OAct4"/>
45         <field name="IMG1" type="text" output="OAct4"/>
46         <field name="ANGLE" type="text" output="OAct4"/>
47
48     </activity>
49
50     <activity tag="non-max_act" description="" type="MAP"
51         activation=".//experiment.cmd" template="%=WFDIR%/Non-
52         max_template">
53         <relation reltype="Input" dependency="sobel_act"/>
54         <relation reltype="Output" name="OAct5"/>
55         <field name="ID" type="float" output="OAct5"
56             decimalplaces="0"/>
57         <field name="START" type="text" output="OAct5"/>
58         <field name="ACTTIME" type="text" output="OAct5"/>
59         <field name="IMG1" type="text" output="OAct5"/>
60         <field name="ANGLE" type="text" output="OAct5"/>
61
62     </activity>
63
64     <activity tag="hysteries_act" description="" type="MAP" activation="./
65         experiment.cmd" template="%=WFDIR%/Hysteries_template">
66         <relation reltype="Input" dependency="non-max_act"/>
67         <relation reltype="Output" name="OAct6"/>
68
69         <field name="ID" type="float" output="OAct6"
70             decimalplaces="0"/>
71         <field name="START" type="text" output="OAct6"/>
72         <field name="ACTTIME" type="text" output="OAct6"/>
73         <field name="IMG1" type="text" output="OAct6"/>
74
75     </activity>
76     <activity tag="reassemble_act" description="" type="REDUCE" operand="
77         ID" activation=".//experiment.cmd;./reassemble.byte oact6;./experiment2.
78         cmd" template="%=WFDIR%/bin/reassemble">
79         <relation reltype="Input" dependency="hysteries_act"/>
80         <relation reltype="Output" name="OAct7"/>
81         <field name="ID" type="float" output="OAct7"
82             decimalplaces="0"/>
83         <field name="TOTALTIME" type="text" output="OAct7"/>

```

---

```
75      <field name="ACTTIME" type="text" output="OAct7"/>
76      <field name="IMG1" type="text" output="OAct7"/></
77          activity>
78      </conceptualWorkflow>
79
80      <executionWorkflow tag="workflow_new" execmodel="DYN_FAF" expdir="
81          %=WFDIR%/exp">
82          <relation name="IAct1" filename="input.dataset"/>
83      </executionWorkflow>
84  </SciCumulus>
```

---

# Bibliographie

- [1] Mathias Bourgoin and Emmanuel Chailloux. Gpgpu composition with ocaml. In *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, page 32. ACM, 2014.
- [2] Mathias Bourgoin, Emmanuel Chailloux, and Jean-Luc Lamotte. Efficient abstractions for gpgpu programming. *International Journal of Parallel Programming*, 42(4) :583–600, 2014.
- [3] Marc Bux and Ulf Leser. Parallelization in scientific workflow management systems. *CoRR*, abs/1303.7195, 2013.
- [4] Steven P Callahan, Juliana Freire, Emanuele Santos, Carlos E Scheidegger, Cláudio T Silva, and Huy T Vo. Vistrails : visualization meets data management. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 745–747. ACM, 2006.
- [5] John Canny. A computational approach to edge detection. *IEEE Transactions on pattern analysis and machine intelligence*, (6) :679–698, 1986.
- [6] Peter Couvares, Tevfik Kosar, Alain Roy, Jeff Weber, and Kent Wenger. Workflow management in condor. In *Workflows for e-Science*, pages 357–375. Springer, 2007.
- [7] Ewa Deelman, Gurmeet Singh, Mei-Hui Su, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Karan Vahi, G Bruce Berriman, John Good, et al. Pegasus : A framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming*, 13(3) :219–237, 2005.
- [8] Juliana Freire, David Koop, Emanuele Santos, and Cláudio T Silva. Provenance for computational tasks : A survey. *Computing in Science & Engineering*, 10(3) :11–21, 2008.
- [9] Scott J. Krieder, Justin M. Wozniak, Timothy Armstrong, Michael Wilde, Daniel S. Katz, Benjamin Grimmer, Ian T. Foster, and Ioan Raicu. Design and evaluation of the gemtc framework for gpu-enabled many-task computing. In *Proceedings of the 23rd International*

*Symposium on High-performance Parallel and Distributed Computing*, HPDC '14, pages 153–164. ACM, 2014.

- [10] Yuancheng “Mike” Luo and Ramani Duraiswami. Canny edge detection on nvidia cuda. *Perceptual Interfaces and Reality Laboratory Computer Science UMIACS, University of Maryland, College Park*, 2008.
- [11] David Marr and Ellen Hildreth. Theory of edge detection. *Proceedings of the Royal Society of London B : Biological Sciences*, 207(1167) :187–217, 1980.
- [12] Marta Mattoso, Jonas Dias, Kary ACS Ocaña, Eduardo Ogasawara, Flavio Costa, Felipe Horta, Vítor Silva, and Daniel de Oliveira. Dynamic steering of hpc scientific workflows : A survey. *Future Generation Computer Systems*, 46 :100–113, 2015.
- [13] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *Queue*, 6(2) :40–53, March 2008.
- [14] Eduardo Ogasawara, Jonas Dias, Daniel Oliveira, Fábio Porto, Patrick Valduriez, and Marta Mattoso. An algebraic approach for data-centric scientific workflows. *Proc. of VLDB Endowment*, 4(12) :1328–1339, 2011.
- [15] Eduardo Ogasawara, Jonas Dias, Vitor Silva, Fernando Chirigati, Daniel Oliveira, Fábio Porto, Patrick Valduriez, and Marta Mattoso. Chiron : A parallel engine for algebraic scientific workflows. *Concurrency and Computation : Practice and Experience*, 25(16) :2327–2341, 2013.
- [16] Daniel Oliveira, Eduardo Ogasawara, Kary Ocaña, Fernanda Baião, and Marta Mattoso. An adaptive parallel execution strategy for cloud-based scientific workflows. *Concurrency and Computation : Practice and Experience*, 24(13) :1531–1550, 2012.
- [17] Victor Podlozhnyuk. Image convolution with cuda. *NVIDIA Corporation white paper*, June, 2097(3), 2007.