République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
Université Ibn Khaldoun de Tiaret
Faculté des Mathématiques et de l'Informatique

Département d'Informatique

THÈSE

Présentée par
ZOUANEB Imane

EN VUE DE L'OBTENTION DU DIPLOME DE DOCTORAT LMD

Filière : Informatique
Option : Systèmes Embrqués et Temps Réel

Thème

Mise en place d'une méthodologie formelle pour la simulation et la validation des architectures à base de GPU

Soutenue le : 30 Mai 2023 Devant le Jury composé de :

Nom et Prénom	Grade		
Mr. BOUZIDANE Ahmed	Professeur	Univ. Ibn Khaldoun de Tiaret	Président
Mr. CHOUARFIA Abdellah	Professeur	Univ. USTO, Oran	Rapporteur
Mr. MERATI Medjeded	MCA	Univ. Ibn Khaldoun de Tiaret	Examinateur
Mr. DEBAKLA Mohamed	MCA	Univ. Mustapha Stambouli, Mascara	Examinateur
Mr. BELARBI Mostefa	Professeur	Univ. Ibn Khaldoun de Tiaret	Invité

Année Universitaire : 2022-2023

République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
Université Ibn Khaldoun de Tiaret
Faculté des Mathématiques et de l'Informatique

Département d'Informatique

THÈSE

Présentée par
ZOUANEB Imane

EN VUE DE L'OBTENTION DU DIPLOME DE DOCTORAT LMD

Filière : Informatique
Option : Systèmes Embrqués et Temps Réel

Thème

Mise en place d'une méthodologie formelle pour la simulation et la validation des architectures à base de GPU

Soutenue le : 30 Mai 2023 Devant le Jury composé de :

Nom et Prénom	Grade		
Mr. BOUZIDANE Ahmed	Professeur	Univ. Ibn Khaldoun de Tiaret	Président
Mr. CHOUARFIA Abdellah	Professeur	Univ. USTO, Oran	Rapporteur
Mr. MERATI Medjeded	MCA	Univ. Ibn Khaldoun de Tiaret	Examinateur
Mr. DEBAKLA Mohamed	MCA	Univ. Mustapha Stambouli, Mascara	Examinateur
Mr. BELARBI Mostefa	Professeur	Univ. Ibn Khaldoun de Tiaret	Invité

Année Universitaire : 2022-2023

Remerciement

Au terme de ce travail de recherche, je suis convaincu que la thèse est loin d'être un travail solitaire. En effet, l'aboutissement de ce travail n'aurait été possible sans l'intervention, le soutien et la générosité de certaines personnes. Qu'elles trouvent ici l'expression de mes plus sincères remerciements.

En premier lieu, j'exprime ma reconnaissance et ma gratitude à mon directeur de thèse : Monsieur Abdellah Chouarfia, Professeur à l'Université d'USTO, Oran. Monsieur, je tiens à vous remercier pour la confiance que vous m'avez accordée en acceptant de diriger cette thèse, pour les conseils que vous m'avez prodigués et le temps que vous m'avez consacré tout au long de ces années de recherches. J'espère que vous trouverez ici l'expression de mon respect pour vos qualités humaines et scientifiques. Je vous en saurais infiniment gré.

J'adresse mes sincères remerciements à Monsieur Mostefa Belarbi, Professeur à l'Université Ibn Khaldoun de Tiaret, merci de votre disponibilité et de vos encouragements. Je tiens également à vous remercier pour votre écoute et pour tous les conseils que vous m'avez donnés sur bien des sujets.

J'adresse mes vifs remerciements à Monsieur Ahmed Bouzidane, Professeur à l'Université Ibn Khaldoun de Tiaret, qui m'a fait l'honneur de présider le jury de thèse.

J'adresse également mes plus sincères remerciements à Monsieur Medjeded Merati, MCA à l'Université Ibn Khaldoun de Tiaret, et Monsieur Mohammed Debakla, MCA à l'université Mustapha Stambouli de Mascara, pour avoir accepté d'être examinateur de ma thèse.

Ces remerciements seraient incomplets si je n'en adressais pas Monsieur Abdelkader Senouci le directeur de laboratoire LIM, Professeur à l'université Ibn Khaldoun de Tiaret, pour le soutient, l'orientation et la disponibilité.

Je remercie également tous les membres du Laboratoire d'informatique et de mathématiques (LIM) qui par leur bonne humeur, leur disponibilité et leur aide m'ont non seulement permis de mener cette thèse à bien mais également de profiter d'une agréable ambiance de travail. Je cite : Abed Abdelmalek, Daoud Hayat Boularadj Khaldia et Benahmed Khaldia; Un grand Merci à vous.

Evidemment, mes remercîments les plus chaleureux vont à ma source d'inspiration, ma très chère mère, mon très cher père ainsi que mes sœurs (Gherissia et Soulef), mon frère El-Emir Abdelkader, ma belle-sœur Saliha, ma chère nièce Chahd, qui ont été toujours là pour me soutenir et m'encourager.

Enfin, je remercie toutes les personnes (nombreuses) que je n'ai pas citées et qui, à un moment ou un autre, m'ont donné l'envie et la force de continuer.

Imane ZOUANEB

Dédicace

Aux plus chers à mon cœur : mes parents,

mes sœurs Gherissia et Soulef,

mon frère El-Amir , ma belle-sœur Saliha

ma nièce Chahd

ملخص

نظام الشريحة (SoC) هو نظام إلكتروني متكامل مدمج على شريحة واحدة. يمكن أن يتكون من وحدة حاسوبية واحدة أو أكثر بما في ذلك وحدة معالجة الرسومات (GPU). تعتبر وحدة معالجة الرسومات بمثابة معالج مساعد يسمح بموازاة تنفيذ المهام في نظام الشريحة وتفريغ وحدة المعالجة المركزية. مرحلة التصميم و الوصف الدقيق للنظام المدمج المراد تنفيذه على شريحة هي أصعب مرحلة في تطوير الأنظمة المدمجة على شريحة. هناك العديد من الطرق لتصميم هذا النوع من الأنظمة. نقترح الجمع بين طريقتين: طريقة تصميم و نمذجة شبه رياضية باستعمال UML و البروفايل MARTE و طريقة تصميم و نمذجة رياضية باستعمال Event B و تعتبر طريقة آمنة وصالحة ومثبتة بواسطة أداة ذات صلة تسمى RODIN . قمنا بإنشاء كود GPU (CUDA و OpenCL) من خلال التحسينات المتتالية للنمذجة الرياضية للنظام SOC بواسطة Event B. كما قمنا باقتراح أنواع بيانات جديدة باستعمال Theory. لقد قمنا بتطوير بعض التطبيقات المتوازية على وحدة معالجة الرسومات مثل: جمع الجداول، الكشف عن قمم الرسومات البيانية لجهاز Raman وتصنيف البيانات بواسطة ACP لتحسين وقت التنفيذ وإجراء مقارنة مع الكود الذي تم إنشاؤه تلقائيًا.

كلمات مفتاحية: نظام الشريحة، نظام مدمج ذو وقت فعلي، وحدة معالجة الرسومات، طريقة UML، البروفايل MARTE، الطريقة Event B، إنشاء الكود، الكشف عن القمم، طريقة التصنيف ACP.

Résumé

Un système sur Puce (SoC) est un système électronique complet intégré sur une puce. Il peut être constitué d'une ou plusieurs unités de calcul dont le GPU. Le GPU est considéré comme un coprocesseur permettant de paralléliser l'exécution des tâches sur le SoC et de décharger le CPU. La modélisation et la spécification d'un Système sur Puce embarqué n'est pas une tâche facile à faire. Il existe de nombreuses méthodes pour modéliser ce type d'applications. Nous proposons de faire le couplage entre deux méthodes : une spécification semi-formelle par UML et le profil MARTE chargé de la modélisation des systèmes embarqués temps réel, et une spécification formelle en Event B sûre, valide et prouvée par un outil pertinent appelé RODIN. Nous avons généré un code GPU (CUDA et OpenCL) à travers les raffinements successifs de la spécification formelle du SoC en Event B et nous avons aussi proposé de nouveaux types en utilisant le Theory plug-in. Nous avons développé quelques applications parallèles sur GPU tels que l'addition vectoriel, la détection de pics de spectre Raman et la classification par ACP pour optimiser le temps d'exécution et faire une comparaison avec le code généré automatiquement.

Mots clés : Système sur Puce, Système Embarqué Temps Réel, GPU, UML, MARTE, Event B, Theory Event B, Génération de code, Détection de pics, ACP.

Abstract

A System on a Chip (SoC) is a fully integrated electronic system on a chip. It could be composed of one or more computing units, including the GPU. The GPU is regarded as a coprocessor, allowing task execution on the SoC to be parallelized and the CPU to be offloaded. It is not easy to model and specify an embedded System-on-Chip. There are many methods to model this type of applications. We propose to couple two methods together: a semi-formal specification by UML and the MARTE profile in charge of modeling real-time embedded systems, and a formal specification in Event B that is safe, valid, and proved by a relevant tool known as RODIN. We generated GPU code (CUDA and OpenCL) through successive refinements of the formal SoC specification in Event B, and we proposed new types using the Theory plug-in. We have developed some parallel GPU applications, such as vector addition, Raman spectrum peak detection, and PCA classification, to optimize execution time and compare to automatically generated code.

Keywords: System on Chip, Embedded Real Time System, GPU, UML, MARTE, Event B, Event B Theory, Code generation, Peak detection, PCA.

Introduction générale

Introduction Générale

1 Problématique

Les méthodes formelles sont des modèles d'ingénierie pour le développement des logiciels. Elles sont basées sur la logique des mathématiques qui les rend plus efficaces et valides. L'utilisation de ces méthodes a permis d'avoir des conceptions non ambiguës, justifiés et valides d'applications diverses et variées et plus particulièrement les systèmes embarqués temps réel.

Le système sur puce (SoC : System on Chip) est un système électronique complet intégré sur une puce comme les téléphones, les GPS, les photocopieuses, etc. Un SoC peut être constitué d'une ou plusieurs unités de calcul et de traitement, une mémoire, un bus et une unité de traitement spécialisé. On s'intéresse aux SoCs contenant deux types d'unité de traitement et de calcul : le CPU (Central Processing Unit) et le GPU (Graphics Processing Unit). C'est une architecture homogène ou le CPU est chargé d'exécuter les threads séquentiels et le GPU a pour rôle d'exécuter les threads parallèles. Pour optimiser la communication entre les composants d'un SoC et l'ordonnancement des tâches sur le SoC, plusieurs systèmes d'interconnexion ont été proposés. Le plus récent est le réseau sur puce (NoC : Network on Chip) inspiré des réseaux informatiques. Un NoC assure l'échange des informations point à point en offrant une meilleure performance que la topologie des bus au niveau de la bande passante, la Qualité de Service et la tolérance aux pannes.

La modélisation et la spécification d'un Système sur Puce embarqué n'est pas une tâche facile à faire. Il existe de nombreux méthodes pour modéliser ce type d'applications ; Parmi elles, les méthodes formelles. Ces dernières reposent sur des notions mathématiques qui en font une spécification sûre et avérée.

Une spécification et une analyse formelles permettent :

Validation : les utilisateurs créent-ils le bon produit ?

Vérification : le créent-ils correctement ?

Est-ce qu'ils génèrent un code sans ambiguïté, réaliste, vérifiable et évolutif.

2 Objectifs de thèse

Le travail de thèse porte sur le développement d'une architecture sur Puce assurant l'ordonnancement des tâches, la représentation des contraintes temporelles des

tâches et la communication entre CPU/GPU. A travers ce travail de recherche, nous essayerons d'atteindre les objectifs suivants :

- Spécifier un système sur Puce (architecture matérielle et architecture logicielle) en utilisant une méthode semi-formelle par le biais de :
    - Les diagrammes UML(Unified Modeling Language) permettant la modélisation du système, des composants, et des relations entre les composants.
    - MARTE (Modeling Analysis Real Time Embedded Systems) dédié à la représentation des aspects des systèmes embarqués tels que : la communication matériel-logiciel, l'ordonnancement, le temps,…etc.
- Ensuite, spécifier le système sur Puce à travers une spécification formelle en langage Event B et la prouver à l'aide d'un outil de vérification valide.
- Pouvoir faire un couplage entre la spécification semi- formelle et la spécification formelle et proposer des règles de passage d'une spécification semi-formelle à une spécification formelle.
- Générer du code exécutable (CUDA et OpenCL) sur GPU à partir de la spécification formelle Event B par raffinement de modèle.
- Développer de nouveaux types des langages (CUDA et OpenCL) en utilisant le Theory plug-in appliqué sur les modèles d'Event B qui permettent d'optimiser le code généré.
- Développer des applications parallèles sur GPU et les tester par l'approche de spécification proposée pour pouvoir générer un pré-code avec les langages OpenCL et Cuda.

3 Contributions

Les principales contributions de cette thèse, peuvent se résumer à travers quatre axes :

- La spécification semi-formelle d'un Système sur puce à base de GPU en utilisant UML et les profils de MARTE traitant l'ordonnancement des tâches, les contraintes de temps, le partage des ressources,…etc.
- La spécification détaillée d'exécution des tâches sur GPU par le raffinement successif de la méthode Event B et les diagrammes d'état-transition de l'UML.
- Le développement d'une application parallèle de détection de pics des images spectrales générées de l'appareil de spectroscopie.
- Génération d'un pré-code parallèle (Cuda et OpenCL) par deux méthodes :

1) Event B seulement à travers les raffinements successifs.
2) Event B et le Theory Plug in.

4 Structure du manuscrit de thèse

L'ensemble des travaux de cette thèse sont synthétisés dans ce manuscrit composé de quatre chapitres, outre l'introduction et la conclusion.

- Le premier chapitre contient une présentation du domaine de thèse : architectures parallèles GPU et ses propriétés et langages de programmation,
- Le deuxième chapitre présente les méthodes de spécification : formelle et semi-formelle.
- Le troisième chapitre résume l'état de l'art de génération de code et présente le plug in Theory.
- Le quatrième chapitre comporte et discute le travail réalisé. Nous commençons par la définition de la problématique. Ensuite nous présentons la spécification formelle et semi-formelle de l'architecture GPU et d'une application parallèle sur GPU. Puis, nous présentons le pré-code parallèle généré par Event B et Event B Theory plug in à travers quelques études de cas.
- Finalement, le manuscrit de thèse se termine par une conclusion générale, qui rappelle à la fois la problématique abordée dans cette thèse, ainsi que nos principales contributions tant sur le plan théorique que sur le plan pratique. Nous citons une liste de perspectives de recherche qui nous semblent présenter un intérêt particulier pour cette thématique de recherche.

Chapitre 1 :
Architecture GPU

Introduction

De nos jours, les GPUs (Graphics Processing Unit) sont devenus très utilisés pour accélérer les applications dans des domaines aussi variés que le traitement d'image, le calcul scientifique ou la bio-informatique. Les GPU sont des coprocesseurs hautement spécialisés, dotés d'une structure multi-cœur leurs permettant d'exécuter des threads en parallèle. Cette concurrence d'exécution repose sur différents ordonnanceurs permettant de recouvrir les latences et d'être efficace.

Dans ce chapitre, nous étudions les systèmes sur puces, les GPUs, le calcul scientifique par GPU et les langages de programmation des GPUs.

1 Système sur Puce

1.1 Définition

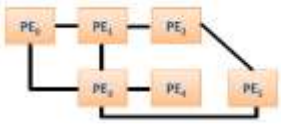
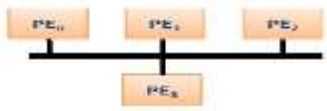
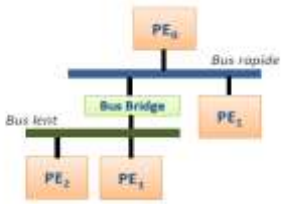
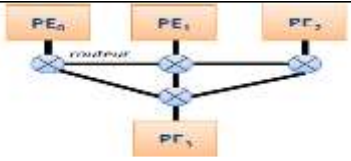
Appelé encore SoC (System-on-Chip), le terme SOC désigne l'intégration d'un système complet sur une seule pièce de silicium. Un SOC est un système complet sur une seule puce composée de différents éléments : processeurs (CPU/GPU), DSP, mémoires, bus, convertisseurs, blocs analogiques, etc. Par exemple les téléphones/Fax, les GPS, les photocopieuses,…etc. [1,2,3].

1.2 Les systèmes d'interconnexion sur puce utilisés dans un SOC

De plus en plus de fonctionnalités sont introduites dans un même système ce qui conduit à la mise en communication (système d'interconnexion) d'un grand nombre de blocs fonctionnels .Les liens de communication n'évoluent pas à la même vitesse et deviennent un goulot d'étranglement.

Les systèmes d'interconnexion sur puce sont généralement des adaptations architecturales à échelle réduite de solutions existantes à plus grande échelle, comme par exemple, des clusters de processeurs sur des cartes électroniques communiquant sur un bus partagé ou encore un réseau de processeurs sur une même carte [3,4,5].

Il y a une multitude de systèmes d'interconnexion qui peuvent être utilisés sur un Système sur Puce (SoC) mentionné dans le tableau 1 [3,4,5,6,7,8] :

Système d'interconnexion	**Présentation graphique**	**Inconvénients**
Le système **point-à-point**		Connexion fixe entre un maître et un esclave, ou éventuellement avec un point intermédiaire.
Le système **de bus partagé**		Plusieurs noeuds qui partagent le même bus.
Le système de **bus hiérarchique**		Bus classique ayant un bridge permettant la connexion de plusieurs sous-systèmes.
Le système **réseau sur puce (NOC)**		La complexité liée à la convergence des applications sur une même puce.

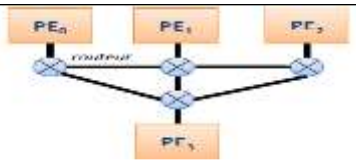
Le système **réseau sur puce (NOC)**		L'hétérogénéité des systèmes nécessite que des outils issus de communautés scientifiques différentes puissent communiquer.

Tableau 1. Les systèmes d'interconnexion sur puce

2 Processeur Graphique (GPU)

2.1 Définition du GPU

Un GPU (Graphics Processing Unit) est aussi appelé unité de traitement graphique et **processeur graphique**. Le GPU a été conçu pour effectuer la visualisation du traitement d'image indépendamment sur le CPU. Ainsi Les GPUs dédiés ont leur propre matériel avec un stockage mémoire haute vitesse et des processeurs multi-cœurs pouvant effectuer des tâches très rapidement. Cette évolution a été principalement motivée par la demande de la technologie des jeux informatiques. Cependant, aujourd'hui, cette technologie peut également être utilisée dans d'autres domaines informatiques [9]. Le GPU est le cœur de la carte graphique[1]. C'est là où sera réalisé tout le traitement d'une image. Il comporte le pipeline graphique mais aussi des unités de calcul arithmétiques et logiques [10]. D'une autre façon, le GPU correspond à un microprocesseur classique sur lequel on aurait supprimé les unités de traitement générique pour les remplacer par d'autres unités hautement spécialisées et dédiées à des opérations mathématiques tels que l'addition, la multiplication des matrices. Son but est de décharger au maximum le processeur central (CPU). Doté de puissantes fonctions mathématiques, le GPU doit être en mesure de traiter plusieurs centaines de millions de données en une seconde. La vitesse du GPU peut atteindre les 500 Mhz. Si l'on comparait un processeur graphique à un coureur cycliste, la fréquence correspondrait en fait à la vitesse à laquelle il pédale, alors que l'architecture correspondrait au plateau utilisé [11]. Le calcul par le GPU permet de paralléliser les tâches et d'offrir un maximum de performances dans de nombreuses applications : le GPU accélère les portions de code les plus lourdes en ressources de calcul, le reste de l'application est

[1] Carte Graphique : est un composant d'une unité de traitement qui permet de convertir des données numériques en données graphiques pouvant être affichées sur un périphérique de sortie (écran, rétroprojecteur, etc.).

affecté au CPU [12,13]. Les applications des utilisateurs s'exécutent donc bien plus rapidement. D'une autre manière, Le GPU est un coprocesseur exploitant le parallélisme de données. Il repose pour cela sur de nombreuses unités de calcul dédiées et organisées hiérarchiquement. Ces processeurs ont été popularisés par le succès d'environnements de développement dédiés au GPGPU[14,15,16]. La Figure 1.1 présente une comparaison entre un processeur multi-cœur (CPU) et un GPU. D'une part, il est très clair que le nombre des unités de calcul au GPU est supérieur au nombre des unités de calcul au CPU et que la mémoire cache au GPU est limitée par rapport à la mémoire cache au CPU. Cette mémoire cache permet aux différents cœurs de communiquer entre eux lors de l'exécution des différentes tâches. D'autre part, une unité de calcul de CPU est plus puissante par rapport à une unité de calcul de GPU mais un ensemble d'unités de calcul (Multi-processor) GPU est très puissant en le comparant avec une seule unité de calcul CPU.

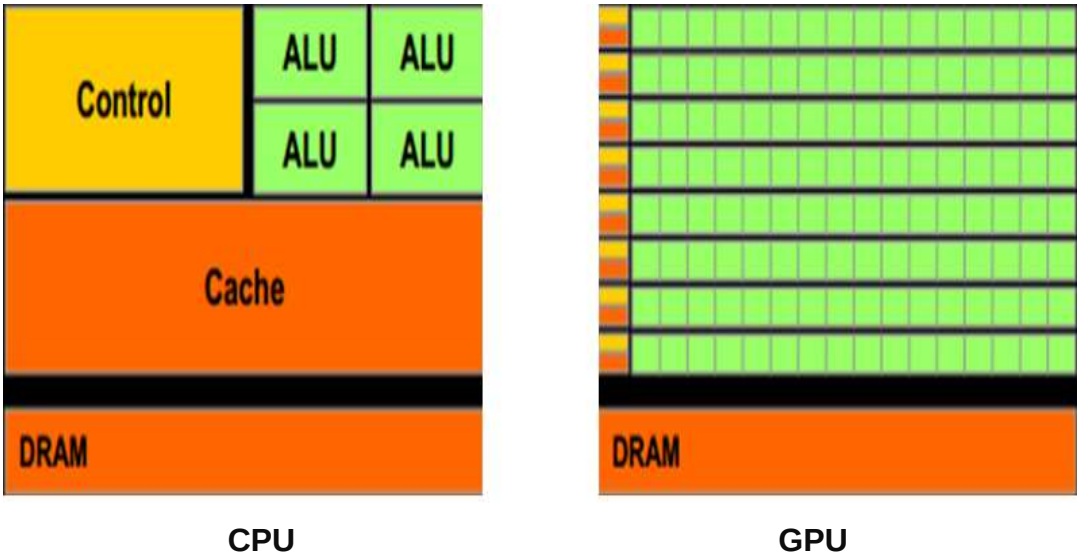


Figure 1.1 Comparaison entre CPU ET GPU

L'utilisation des processeurs graphiques prend tout son sens pour des applications hautement parallèles qui nécessitent des milliers de taches indépendantes. Il est en effet nécessaire d'occuper totalement et durablement le processeur graphique pour amortir le coût de ses communications avec la machine hôte [1].

2.2 GPGPU

Le GPGPU (General Purpose Graphics Processing Unit) est une nominalisation donnée à l'utilisation des GPUs pour le calcul scientifique en plus du traitement graphique [17 ,18]. Les cartes graphiques, sur lesquelles repose l'exécution des jeux, ont dû évoluer et permettent maintenant non seulement de l'affichage d'objets 3D mais

également d'effectuer des opérations arithmétiques. De plus, compte tenu de la puissance de calcul demandée, les GPUs sont devenus plus puissants que les CPUs traditionnels. La Figure 1.2 montre Le développement des architectures CPU et GPU. La différence est particulièrement flagrante pour le calcul en simple précision [19,20,21].

Le General-Purpose Processing on Graphics Processing Unit (GPGPU) est un domaine de l'informatique visant à effectuer des calculs génériques (sans obligatoirement de lien avec les calculs graphiques) sur les GPUs. L'objectif du GPGPU est de tirer le maximum de profit de l'architecture hautement parallèle des GPUs fournissant une forte puissance de calcul permettant de réaliser certains traitements en parallèle. Ainsi, il y a bénéfice de l'architecture pensée pour le calcul parallèle [21].

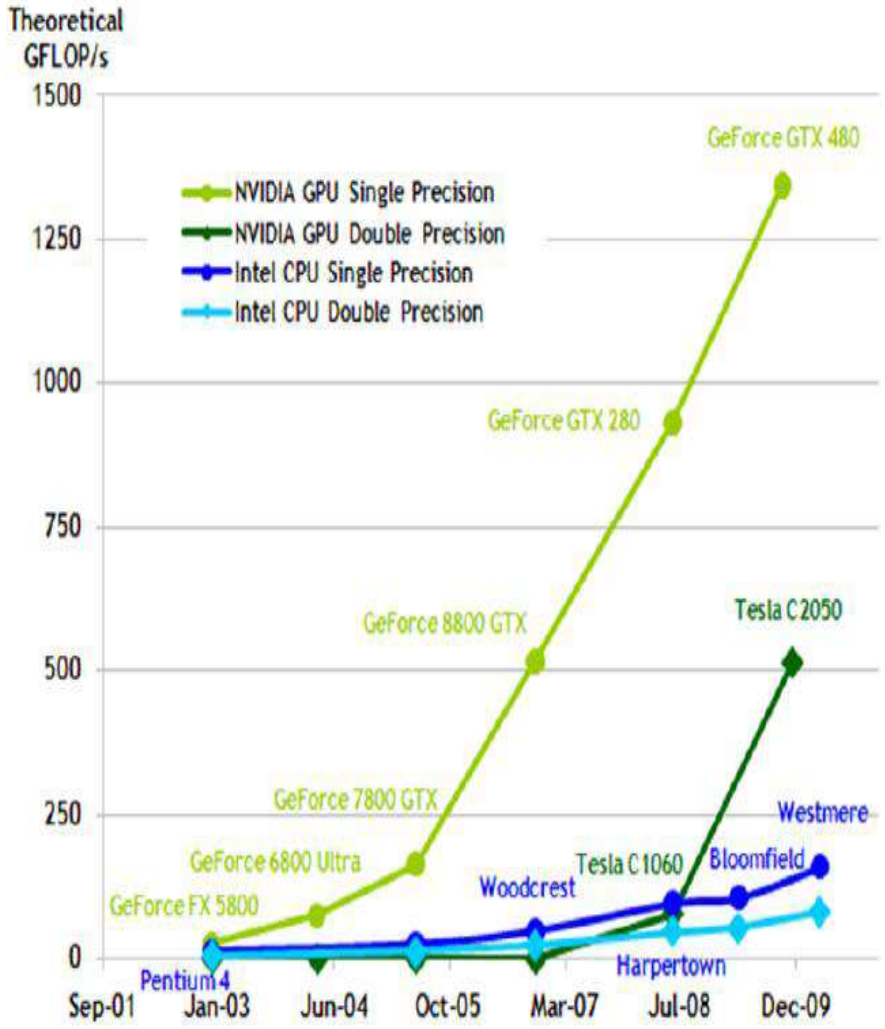


Figure 1.2. Développement des architectures CPU et GPU

La grande puissance des cartes graphiques est exploitée depuis une dizaine d'années par la communauté scientifique. Les premiers calculs sur GPU étaient effectués en détournant les API graphiques. Depuis 2007, les cartes graphiques sont programmables grâce à l'introduction de CUDA par NVIDIA. Ce langage dérive du C/C++, il permet de gérer l'architecture particulière des cartes graphiques [19,21].

2.3 Architecture NVIDIA Fermi GF100 GPU

Un GPU NVIDIA est composé entre autre, d'une mémoire globale et d'un ensemble de streaming multi-processors (SM). Un Streaming multi-processor contient un ensemble de cœurs appelés streaming processors (SP) et d'une mémoire partagée entre ces derniers. Le programme exécuté sur GPU est appelé KERNEL [22,23,24].
Cette architecture fournit un ensemble de capacités par rapport aux GPUs précédentes crée par NVIDIA. L'architecture Fermi illustrée dans la Figure 1.3 offre plus de 512 cœurs CUDA en plus des caractéristiques pour les jeux et la haute performance de calcul. Gravée en 40 nm par TSMC, la puce GeForce 100 est cadencée à 700 MHz alors que ses unités de calcul opèrent à 1401 MHz. Couplée à la mémoire GDDR5 interfacée sur 384 bits, la puce est censée bénéficier d'une bande passante mémoire de 177,4 Go/s. Utilisant 3 billions de transistors [24].

Figure 1.3. Architecture du GPU GF100

Le GPU GF100 est constitué de quatre GPC (Graphic Processing Cluster) ou chaque cluster est constitué de quatre SMs (Streaming Multi-processor) et un Raster

engine connecté aux quatre SMs. Les SMs sont connectés à une mémoire cache partagée de deuxième niveau L2, une interface host, un ordonnanceur Giga Thread engine et multiple interfaces DRAM. Un SM est constitué de 32 SPs (Streaming Processor), le total est de 512 cœurs CUDA sur le GPU. Le GPU GF100 permet de paralléliser l'exécution de threads [24,25].

2.3.1 Le Streaming Multi-processor du GPU GF100

Un Streaming Multi-processor de GF100 est constitué de 32 cœurs CUDA, une unité LOAD/STORE, quatre Special Function (SFU) exécutant les instructions (sin, cos, reciprocal et racine carré) et unités Texture. Chaque SM contient une mémoire cache configurable et une mémoire locale. Le total est de 48 Kbytes de mémoire partagée avec 16Kbytes de mémoire locale cache de premier niveau (L1) appelé également SRAM. Les SMs sont reliées à une mémoire cache du deuxième niveau L2 comme illustré dans la Figure 1.4. Chaque SM possède un double scheduler relié à un moteur Polymorph ayant des fonctions spécialisées pour le traitement des attributs graphiques et pour la tessellation[2] . Dans chaque SM, il existe des fonctions câblées pour le graphisme : Texture filtering, Texture cache, tessellation**,** Vertex Fetch, Attribute Setup, Stream Output, Viewport Transform [26,27,28].

[2] la tessellation fait référence à la division des ensembles de données de polygones présentant des objets dans une scène en structures appropriées pour le rendu. Surtout pour le rendu en temps réel, les données sont pavées en triangles.

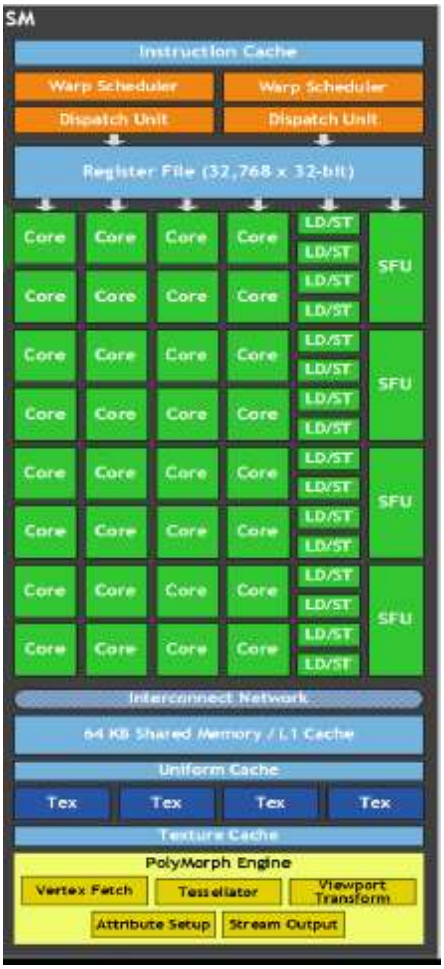


Figure 1.4. Le Streaming Multi-processor du GPU GF100

2.3.2 Le streaming Processor (SP) du GPU GF100

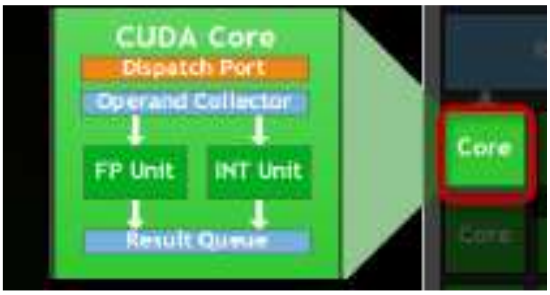


Figure 1.5. Le Streaming Processor

Comme illustré dans la Figure 1.5, chaque cœur CUDA est composé d'une arithmetic logic unit (ALU), d'une floating point unit (FPU), d'un collecteur de port et d'une queue de résultats pour sauvegarder les résultats.

L'architecture Fermi GF100 implémente le nouveau standard IEEE 754-2008 de la virgule flottante (floating point) fournissant l'instruction fused-multiply add

(FMA) pour une précision numérique (simple et double). FMA améliore l'exécution vers des instructions multiply-add (MAD) en effectuant la multiplication et l'addition avec une étape finale d'arrondi et l'assurance de non perte de précision dans l'addition.

Dans Fermi, la nouvelle conception d'ALU d'entier (interger ALU) supporte une précision de 32bits pour toutes les instructions et la consistance avec le standard des langages de programmation. L'unité ALU d'integer est optimisée aussi pour supporter l'architecture 64 bits avec les opérations de précision.

Plusieurs instructions sont supportées telles que, booléenne, shift, move, compare, convert, bit field extract, bit reverse insert et le calcul de population [1,20,28].

2.4 Architectures AMD RADEON HD7770, HD7750

Pour ses Radeon HD7700, AMD a conçu Cape Verden, un GPU de 1,5 milliard de transistors, il représente grossièrement un tiers du GPU Tahiti. Comme illustré dans la Figure 1.6, l'AMD Cape Verden reprend 10 blocs d'unité de calcul en portant le total à 640 unités de calcul et 40 unités de texturing ainsi qu'un bus mémoire de 128 bits . Le GPU Juniper dispose 800 unités de calcul, organisées en 160 unités vec5. Une organisation qui offre cependant un rendement moindre. En plus de sa nouvelle architecture, Cape Verde compte sur la montée en fréquence pour se démarquer. La Radeon HD7770 présenté voit son GPU atteindre pour un modèle de série, la barre de 1 GHz. AMD en profite, au passage, pour mettre en place une campagne de communication autour des GPU « GHz Edition ». Il n'y aura toutefois pas de déclinaison avec une fréquence inférieure [29].

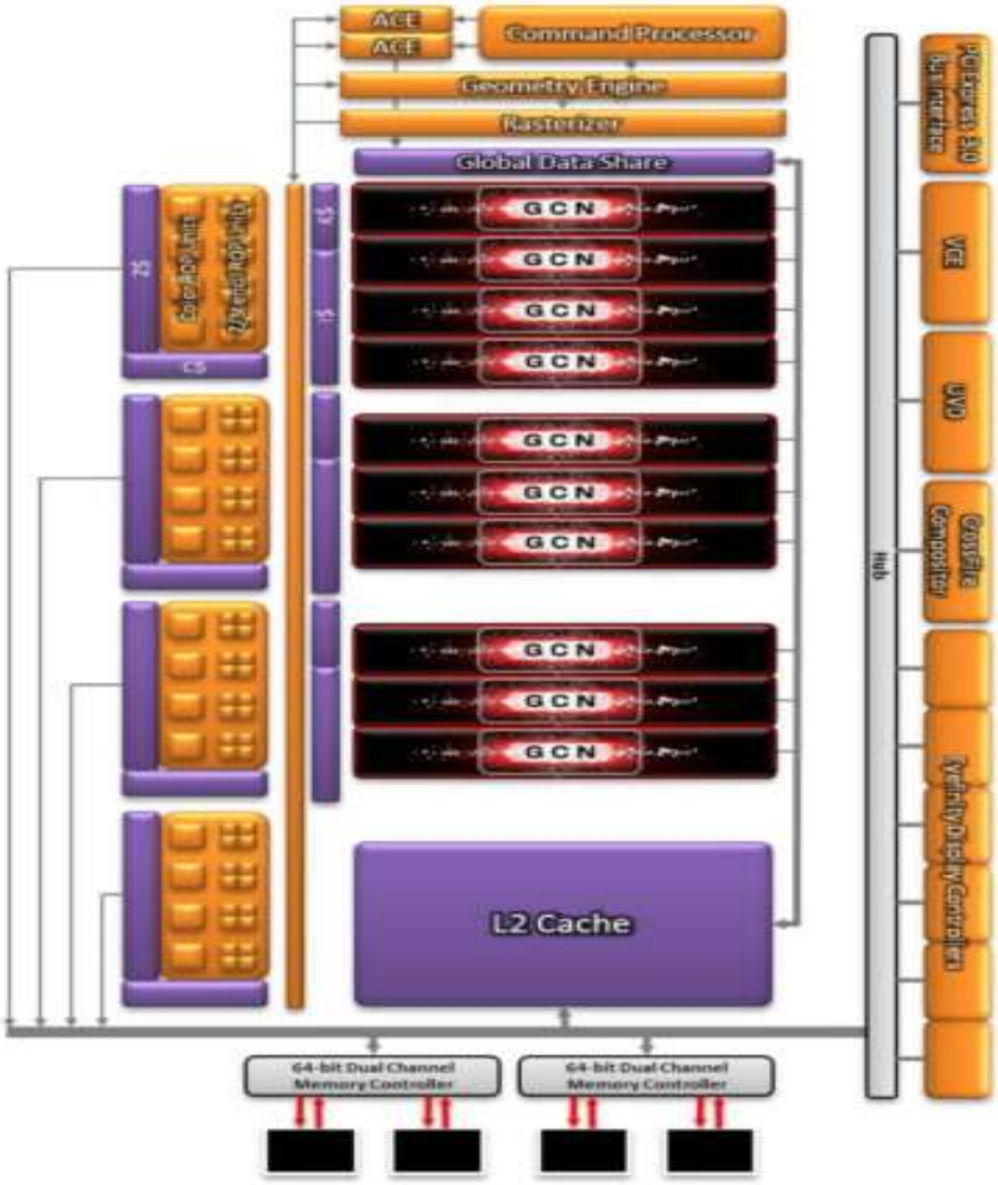


Figure 1.6. Architecture Interne de Radeon HD 7770

Chaque bloc d'unités de calcul, nommé CU, intègre 4 unités vec16 alimentées par une logique de gestion différente. Une Radeon HD6900 présente, par exemple, une instruction vec4 pour 16 pixels, alors qu'une Radeon HD7000 présente une simple instruction scalaire pour 64 pixels. Etant donné qu'il n'est pas possible de vectoriser tout le code, cette seconde approche est plus efficace, mais plus gourmande en logique de gestion. Le GPU computing profite, de son côté, de l'ajout de processeurs de commande dédiés (les ACE ou *Asynchronous Compute Engines*) pour réduire la latence, l'overhead et autoriser le multi-threading, ainsi que d'une structure de caches L1 et L2 en lecture/écriture pour améliorer les performances et la flexibilité du sous-système mémoire. Cape Verde reprend l'intégralité de ces évolutions, permettant à AMD de disposer d'un socle commun pour faciliter le travail des développeurs [29].

Le tableau 2 résume les caractéristiques des GPUs de AMD (Voir Section 2.4) et de NVIDIA (Voir Section 2.3)

	HD 7770 GDDR5	HD 7750 GDDR 5	HD 5770 GDDR5	GF Fermi GDDR5	**GF Fermi GDDR3**	**GF Tesla GDDR3**
GPU	CapeVerde	Cape Verde	Juniper	GF 100	**GT 630**	**GF 210**
Unités de calcul	640	512	160	448	**448**	**240**
Unités de texturing	40	32	40	56	**96**	**80**
Render Output Unit	16	16	16	32	**32**	**32**
Fréquence GPU (MHz)	1000	800	850	607	**810**	**589**
Fréquence Mémoire (MHz)	*1000*	800	850	1215	**1800**	**400**
Taille de mémoire (Mo)	1024	1024	1024	1280	**1024**	**1024**
Bus Mémoire (bits)	128	128	128	320	**128**	**128**
BP mémoire (Gio/s)	67.1	67.1	71.5	127.7	**28.5**	**08**

Tableau 2. Comparaison des caractéristiques de GPUs AMD et NVIDIA

2.5 Langages de programmation des GPUs

2.5.1 CUDA

CUDA (Compute Unified Device Architecture) fournit un ensemble de bibliothèques logicielles, un environnement d'exécution et un ensemble de pilotes pour plusieurs langages de programmation (C, C++, Fortran). C'est un langage, dérivé du C, fourni avec son compilateur, supportant 9 nouveaux mots clés, 24 nouveaux types et 62 nouvelles fonctions [23,29]. L'environnement CUDA développé par NVIDIA est un environnement orienté calcul vectoriel hautes performances. Il repose sur une architecture, un langage, un compilateur, un pilote, divers outils et des bibliothèques. Dans un programme CUDA typique, les données sont envoyées de la mémoire centrale vers la mémoire GPU, le CPU envoie des commandes au GPU, ensuite le GPU exécute les noyaux de calcul en ordonnançant le travail sur le matériel disponible pour enfin copier le résultat de la mémoire GPU vers la mémoire CPU [15,30].

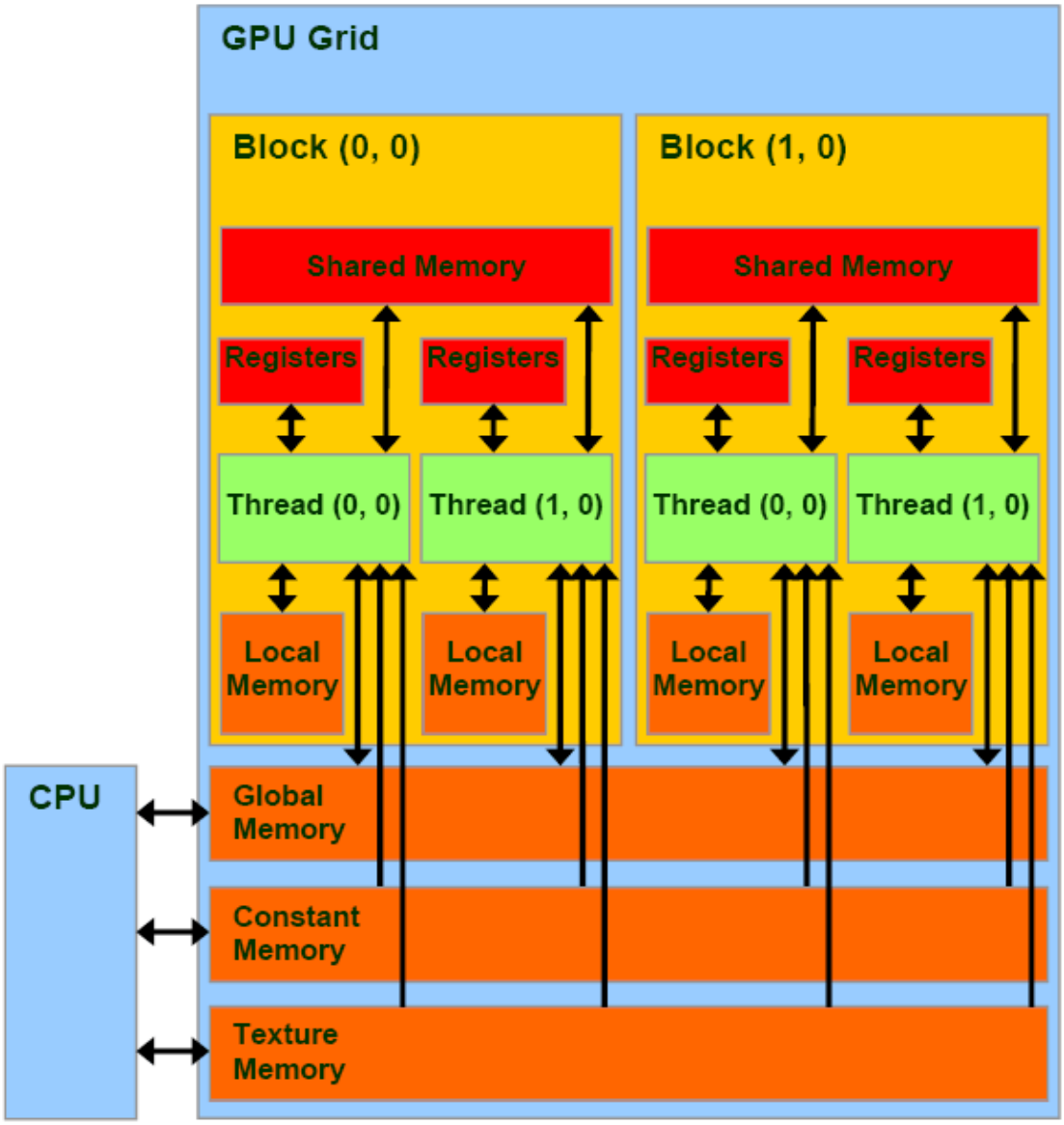


Figure 1.7. Architecture CUDA

Comme illustré dans la Figure 1.7, l'architecture matérielle utilisée par CUDA comporte un processeur hôte, une mémoire hôte, d'une carte graphique NVIDIA prenant en charge CUDA. Les GPUs ont un mode d'exécution SIMT (Single Instruction Multiple Threads). L'organisation matérielle est liée à l'organisation du parallélisme défini au niveau du langage dans CUDA. Le langage de programmation est une extension du langage C avec des extensions pour indiquer si une fonction est exécutée sur le CPU ou le GPU. Les fonctions exécutées sur le GPU sont appelées kernels. CUDA donne la possibilité au développeur de définir des variables résidées dans l'espace d'adressage du GPU et le type de parallélisme pour l'exécution des kernels en termes de dimension de grille, de bloc, et de thread [11,15,31].

Le code présenté ci-dessous est un programme d'addition de deux vecteurs A et B, le résultat est sauvegardé dans le tableau C. La fonction, de type "__global__" est activée par le CPU dans le code principal mais elle s'exécute sur le GPU. Ce dernier démarrera autant de copies de cette fonction qu'il y aura de "threads" à exécuter. Les éléments à additionner sont indicés par "i" et la position du bloc dans la grille (blockIdx) ainsi qu'en connaissant le format de la grille [27].

```
#define N 10
__global__ void add( int *a, int *b, int *c ) {
int tid = blockIdx.x; // traite l'élément à cet indice
if (tid < N)
c[tid] = a[tid] + b[tid];
}
```

Le code cité ci-dessous montre les étapes de chargement des données, ensuite le lancement des instances de kernel en parallèle. Ce code est lancé sur l'hôte (CPU).

```
#define N 10
int main( void ) {
int a[N], b[N], c[N];
int *dev_a, *dev_b, *dev_c;
// Allocation mémoire sur le GPU
HANDLE_ERROR( cudaMalloc( (void**)&dev_a, N * sizeof(int) ) );
HANDLE_ERROR( cudaMalloc( (void**)&dev_b, N * sizeof(int) ) );
HANDLE_ERROR( cudaMalloc( (void**)&dev_c, N * sizeof(int) ) );
// Remplissage des tableaux a et b sur le CPU
for (int i=0; i<N; i++) {
a[i] = -i;
b[i] = i * i;
}// Copie des tableaux a et b sur le GPU
HANDLE_ERROR( cudaMemcpy( dev_a, a, N * sizeof(int),
cudaMemcpyHostToDevice ) );
HANDLE_ERROR( cudaMemcpy( dev_b, b, N * sizeof(int),
cudaMemcpyHostToDevice ) );
add<<<N,1>>>( dev_a, dev_b, dev_c );
// Copie du tableau c du GPU vers le CPU
HANDLE_ERROR( cudaMemcpy( c, dev_c, N * sizeof(int),
cudaMemcpyDeviceToHost ) );
// Affichage du résultat
for (int i=0; i<N; i++) {
printf( "%d + %d = %d\n", a[i], b[i], c[i] );
}
// Libération de la mémoire allouée sur le GPU
cudaFree( dev_a );
cudaFree( dev_b );
cudaFree( dev_c );
return 0;
}
```

2.5.2 OpenCL

OpenCL (Open Computing Language) offre un format portable et exécutable sur tout type de plateforme hétérogène. OpenCL reste tout de même axé sur les plateformes à base d'un processeur hôte et d'un GPU. OpenCL est une interface de programmation ouverte [19,23]. Un standard pour les calculs parallèles contenant un langage, une API, une librairie et un système de runtime. La Figure 1.8 montre que OpenCL est basé sur une plateforme qui divise le système en un hôte(host) et un ou plusieurs dispositifs (devices) de calcul. Ces derniers agissent en tant que coprocesseur (ex. GPU) pour le host. Les dispositifs sont subdivisés en plusieurs unités de calcul (CUs) à leur tour divisés en multiple élément de traitement (PEs). Une application OpenCL est exécutée sur un hôte qui envoie des instructions définies sous forme de fonctions appelées kernels au dispositifs (Compute devices) [32].

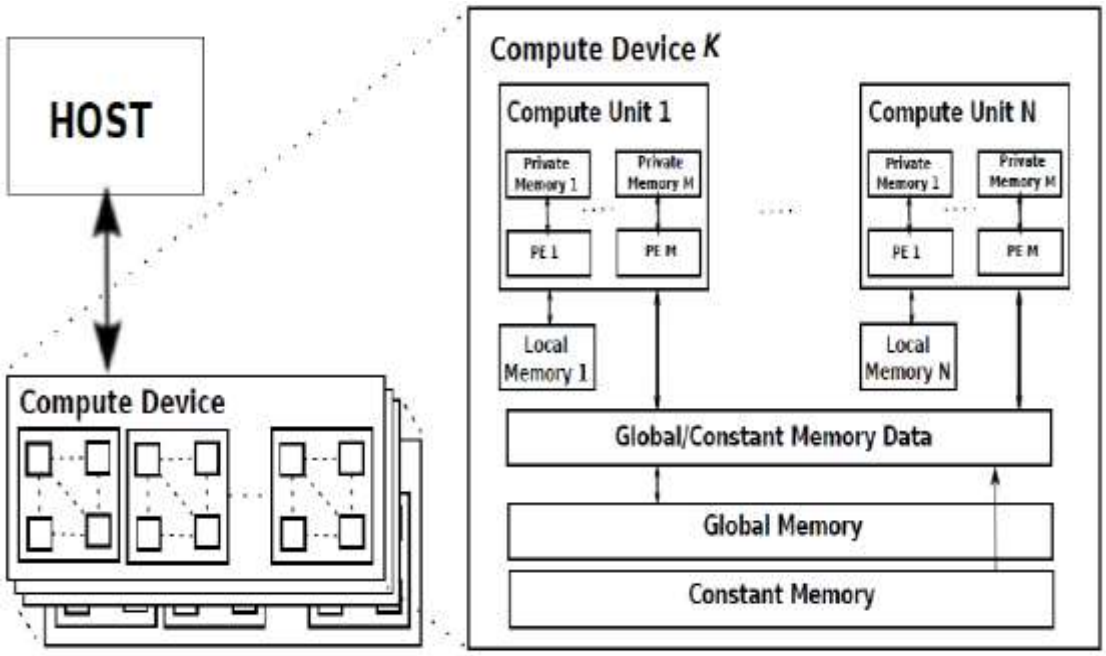


Figure 1.8. La plateforme d'OpenCL

Il y a deux types de parallélisme dans OpenCL : data-parallel model (parallélisme de données) et task-parallel model (parallélisme de tâches). Une device exécute un multiple d'instances de kernel en parallèle sur différentes données (data). Chaque instance de kernel est appelée un **work-item (WI)**. Les work-items peuvent être collectionnés dans un **work-group (WG)**. Chaque work-item est identifié par un ID (local ou global). Les work-groups sont affectés au CUs où les WIs de chaque groupe sont exécutés en parallèle sur les PEs. La synchronisation des work-items est possible

au niveau de WG seulement. Les CUs prennent la forme de grille. OpenCL fournit un langage de programmation (en extension de C) pour écrire le code des kernels [32].
Le code-source illustré ci-dessous est un exemple d'un programme écrit en OpenCL. Comme le langage Cuda, Il y a un code qui s'exécute sur GPU (kernel) et un code qui s'exécute sur l'hôte. Le code de kernel est :

```
__kernel void  add(__global const int *A, __global const int *B, __global int *C) {

// Récupère l'index de l'élément courant à traiter

int i = get_global_id(0);

// Faire l'opération
C[i] = A[i] + B[i];
}
```

Le code hôte :

```
int main(void) {
// Créer les deux vecteurs d'entrée
int i;
const int LIST_SIZE = 10;
int *A = (int*)malloc(sizeof(int)*LIST_SIZE);
int *B = (int*)malloc(sizeof(int)*LIST_SIZE);
for(i = 0; i < LIST_SIZE; i++) {
A[i] = i;
B[i] = LIST_SIZE – i;
}

// Calcul sur l'hôte
int *C = (int*)malloc(sizeof(int)*LIST_SIZE);
for(i = 0; i < LIST_SIZE; i++) {
C[i] = A[i] + B[i];
}

// Chargez le code source du noyau dans le tableau source_str
FILE *fp;
char *source_str;
size_t source_size;

fp = fopen(« add_kernel.cl », « r »);
if (!fp) {
fprintf(stderr, « Failed to load kernel.\n »);
exit(1);
}
source_str = (char*)malloc(MAX_SOURCE_SIZE);
source_size = fread( source_str, 1, MAX_SOURCE_SIZE, fp);
fclose( fp );

// Obtenir des informations sur la plateforme et le dispositif
cl_platform_id platform_id = NULL;
cl_device_id device_id = NULL;
```

```
cl_uint ret_num_devices;
cl_uint ret_num_platforms;
cl_int ret = clGetPlatformIDs(1, &platform_id, &ret_num_platforms);
ret = clGetDeviceIDs( platform_id, CL_DEVICE_TYPE_DEFAULT, 1,
&device_id, &ret_num_devices);
// Create an OpenCL context
cl_context context = clCreateContext( NULL, 1, &device_id, NULL, NULL, &ret);

// Créer une queue de commandes
cl_command_queue command_queue = clCreateCommandQueue(context,
device_id, 0, &ret);

// Créer des mémoires tampons sur l'appareil pour chaque vecteur
cl_mem a_mem_obj = clCreateBuffer(context, CL_MEM_READ_ONLY,
LIST_SIZE * sizeof(int), NULL, &ret);
cl_mem b_mem_obj = clCreateBuffer(context, CL_MEM_READ_ONLY,
LIST_SIZE * sizeof(int), NULL, &ret);
cl_mem c_mem_obj = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
LIST_SIZE * sizeof(int), NULL, &ret);

// Copiez les listes A et B dans leurs mémoires tampons respectives
ret = clEnqueueWriteBuffer(command_queue, a_mem_obj, CL_TRUE, 0,
LIST_SIZE * sizeof(int), A, 0, NULL, NULL);
ret = clEnqueueWriteBuffer(command_queue, b_mem_obj, CL_TRUE, 0,
LIST_SIZE * sizeof(int), B, 0, NULL, NULL);

// Créer un programme à partir de la source du noyau
cl_program program = clCreateProgramWithSource(context, 1,
(const char **)&source_str, (const size_t *)&source_size, &ret);

// Lancer le programme
ret = clBuildProgram(program, 1, &device_id, NULL, NULL, NULL);

// Créer la functiondu  noyau
cl_kernel kernel = clCreateKernel(program, «  add », &ret);

// Définir les arguments du noyau
ret = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&a_mem_obj);
ret = clSetKernelArg(kernel, 1, sizeof(cl_mem), (void *)&b_mem_obj);
ret = clSetKernelArg(kernel, 2, sizeof(cl_mem), (void *)&c_mem_obj);

// Exécutez le noyau OpenCL sur la liste
size_t global_item_size = LIST_SIZE; // Process the entire lists
//size_t local_item_size = 1; // Process one item at a time
size_t local_item_size = 32; // Process one warp at a time
ret = clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL,
&global_item_size, &local_item_size, 0, NULL, NULL);

// Lire la mémoire tampon C sur l'appareil dans la variable locale C
int *C = (int*)malloc(sizeof(int)*LIST_SIZE);
ret = clEnqueueReadBuffer(command_queue, c_mem_obj, CL_TRUE, 0,
LIST_SIZE * sizeof(int), C, 0, NULL, NULL);
```

```
// Afficher le résultat à l'écran
for(i = 0; i < LIST_SIZE; i++)
printf(« %d + %d = %d\n », A[i], B[i], C[i]);

// Libérer les ressources
ret = clFlush(command_queue);
ret = clFinish(command_queue);
ret = clReleaseKernel(kernel);
ret = clReleaseProgram(program);
ret = clReleaseMemObject(a_mem_obj);
ret = clReleaseMemObject(b_mem_obj);
ret = clReleaseMemObject(c_mem_obj);
ret = clReleaseCommandQueue(command_queue);
ret = clReleaseContext(context);
free(A);
free(B);
free(C);
return 0;
}
```

2.5.3 BrookGPU, Brook+

BrookGPU est une implémentation GPU du langage Brook, tous deux développés par le Stanford University Graphics Lab. C'est un langage basé sur la gestion de flux de données. AMD/ATI a également proposé Brook+, une amélioration de BrookGPU, pouvant être utilisé uniquement sur leurs cartes. Folding@home, projet mondial de calcul distribué simulant les repliements de protéines dans le but d'en tirer des solutions médicales, utilise en partie Brook+ [33].

3 Exécution d'un programme sur GPU

Le nombre d'unités de traitement sur le GPU permet un traitement parallèle. Le GPU est utilisé pour améliorer les performances des applications telles que le multimédia et les programmes informatiques à grande échelle [14,33]. Un programme qui contient des tâches répétitives et non indépendantes peut être exécuté sur GPU. Ce programme est transformé en un ou plusieurs kernels. Ces programmes se trouvent dans plusieurs domaines comme le traitement d'image, traitement de signal. Un GPU ne peut pas exécuter un programme sans passer par le CPU. L'exécution d'un programme sur GPU se déroule comme suit :

- Allocation de la mémoire sur GPU.

- ❖ Transfert des données du CPU (hôte) au GPU.
- ❖ Exécution du kernel.
- ❖ Transfert des données du GPU au CPU (hôte).
- ❖ Libération de la mémoire allouée sur GPU.

L'exécution parallèle des programmes sur GPU a beaucoup d'avantages tels que la certitude des résultats, la minimisation de temps d'exécution. De l'autre part cette technique a aussi des inconvénients tels que la consommation de l'énergie et l'encombrement et le blocage d'exécution à cause de la communication entre tâches au cours de l'exécution minimale.

4 Synthèse des travaux de validation et d'analyse de code sur GPU

Pour résoudre les problèmes cités ci-dessus de l'exécution des programmes sur GPU, plusieurs travaux ont proposés des solutions. On cite :

4.1 Inter Block GPU communication via FastBarrier Synchronisation

[34] traite la communication inter-bloc au niveau du GPU à travers la mémoire globale en utilisant *une barrière de synchronisation*. La synchronisation des blocs a été effectuée uniquement sur le CPU. [34] propose un modèle performant pour l'exécution des kernels sur GPU et assure la communication inter-blocs à travers des barrières de synchronisation sur GPU. Donc si on estime le temps d'exécution de ce type de programme, on remarque que les deux premières étapes (lancement des kernels sur GPU, calcul sur GPU) prennent 50% et l'étape de communication inter-blocs consomme 50% ou plus. Alors le but de ce travail est d'améliorer la synchronisation inter-blocs au niveau du GPU. Trois approches ont été proposées dans le tableau 3 [34].

GPU simple synchronisation	**GPU tree-based synchronisation**	**GPU lock free synchronisation**
Utilisation d'une ou de plusieurs variables globales de type mutex pour compter le nombre de blocs de threads qui atteignent le point de synchronisation	1) Utilisation d'une ou de plusieurs variables globales de type mutex pour compter le nombre de bloc de threads qui atteignent le point de synchronisation. 2) Les blocs sont divisés en groupes, chaque groupe est synchronisé avec GPU simple synchronisation (GSS)	1) Utilisation de deux tableaux au lieu d'une variable global mutex 2) assignation d'une variable de synchronisation pour chaque bloc de threads. 3) Elimination du besoin d'opération atomique 4) le temps de synchronisation est constant.

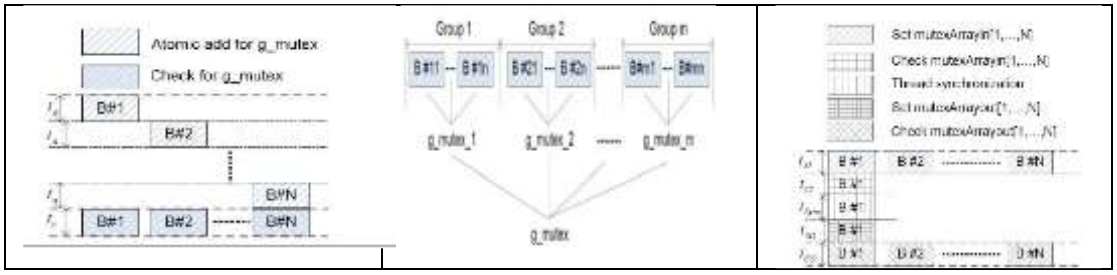


Tableau 3. Approches de synchronisation proposée

4.2 GPUVerify : A verifier for GPU Kernels

[35] propose une approche de vérification de la rapidité et de la divergence des kernels écrits dans des langages de programmation mainstream tels que : OpenCL et CUDA. Basé sur une nouvelle sémantique d'opération formelle pour la programmation GPU appelée sémantique de synchronous, delayed, visibility (SDV). Elle fournit une barrière de divergence précise pour les kernels GPU ce qui permet la vérification de la concurrence. Une exécution prédictive a été utilisée pour la vérification du code généré par les compilateurs GPU. GPUverify a été testé directement sur 163 codes CUDA et OpenCL de sources publiques et commerciaux [35].

4.3 GPUDET : A deterministic GPU architecture

Le but de [31] est de fournir un environnement déterministe pour faciliter les tests et le debug des applications. [31] propose des modifications architecturales rendant les exécutions sur GPU déterministes. La Figure 1.9 représente l'architecture proposée.

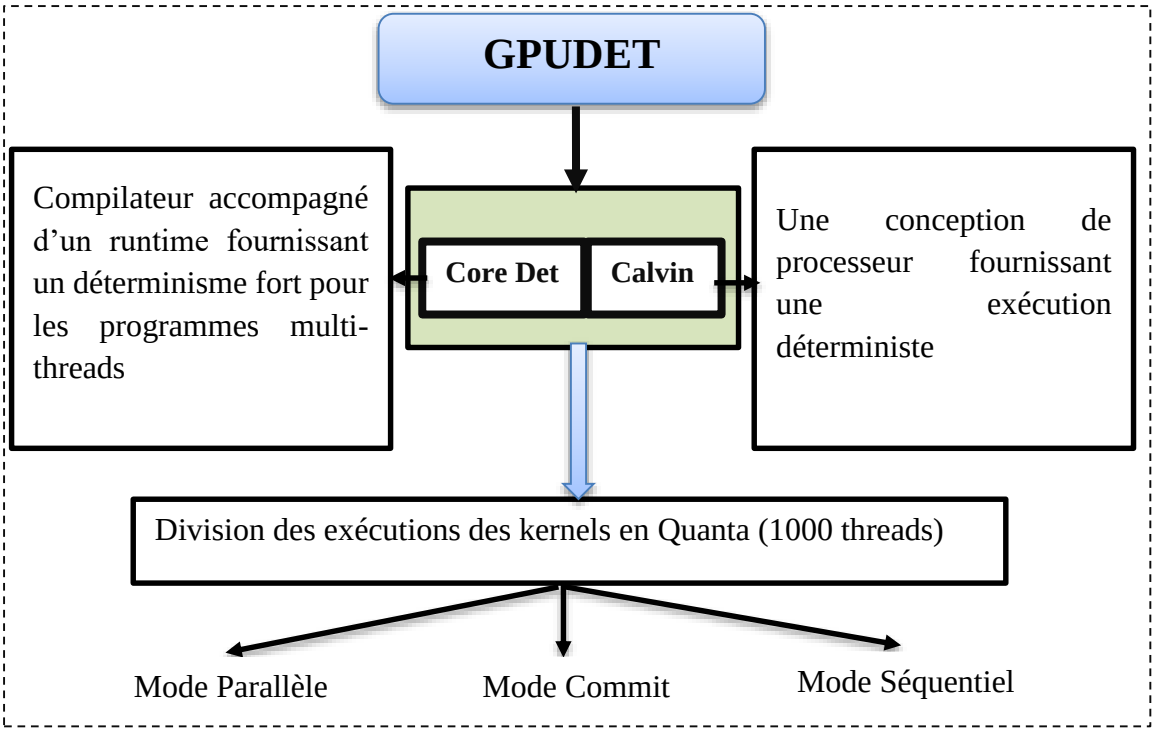


Figure 1.9. Architecture de GPUDET

4.4 To GPU synchronize or not GPU synchronize

Les GPUs sont devenus des processeurs programmables massivement parallèles. Le GPGPU limite la classe de l'applicative cible aux applications intégrant un fort parallélisme de données et dans une moindre mesure du parallélisme de tâches. [37] part du constat que lorsqu'un kernel est lancé, les blocs ne peuvent pas communiquer entre eux. [37] propose des barrières de synchronisation entres les SMs d'un GPU en utilisant la mémoire globale. Cette proposition avait pour but de comparer les performances de différentes barrières de synchronisation et de délivrer des résultats corrects. Ils ont remarqué que NVIDIA a exposé aux développeurs la fonction threadfence() garantissant l'exactitude de la communication inter-SM. Cependant, les performances de la synchronisation GPU (directe) sont bien en deçà de la synchronisation indirecte via le CPU [37].

4.5 Impact des schedulers sur la prédictibilité dans les GPU

[38] propose une mesure de prédictibilité d'exécution des tâches sur GPU NVIDIA ainsi que les tests CUDA. Les schedulers définissent un certain nombre de concept comme le déterminisme et la prédictibilité. Le déterminisme est vu comme l'observation d'un comportement identique pour un jeu de données et un programme identique. La prédictibilité est le fait d'être capable de connaitre à l'avance le résultat

ou le comportement d'un processus pour une suite identique d'états et d'actions. Plusieurs sources d'indéterminisme sont identifiées :

- ❖ Les GPUs ont plusieurs domaines d'horloges, ainsi, le circuit de synchronisation entre ces domaines peut introduire l'indéterminisme.
- ❖ Le temps d'accès à la mémoire hors puce est variable.
- ❖ Les ordonnanceurs peuvent modifier l'ordre d'exécution et potentiellement les assignations entre logiciels et matériels [38].

La mesure de prédictibilité proposée est basée sur la lecture des registres d'horloges ou sur l'ordre d'accès à une donnée en mémoire globale (en comparant l'ordre d'accès relatif à ces données). Une implémentation CUDA des mesures de prédictibilités a été proposée [33,37,38].

Après avoir étudié les travaux cités-ci-dessus on peut dire que :

- ✓ tous ces travaux ont pour but d'améliorer la qualité des codes CUDA et OpenCL sur GPU et de vérifier la capacité des GPU à exécuter ces algorithmes dans un temps prédictible, avec un processus déterministe. Puisque la structure des GPU NVIDIA est très complexe, lorsqu'on lance un kernel on ne peut pas prédire les interactions entre les SM et l'ordre des exécutions des threads. Une solution consiste à prédire le comportement des GPU.
- ✓ L'exécution des blocs se fait séparément, c'est-à-dire, les blocs ne peuvent pas communiquer entre eux. Des travaux [35] proposent des barrières de synchronisation entre les SMs pour les communications et éviter les problèmes liés à la concurrence d'accès à la mémoire global.

Conclusion

Au cours de ce chapitre, nous avons essayé de donner une vision sur notre domaine de recherche en définissant les processeurs graphiques et les langages de programmation tels que CUDA et OpenCL. La programmation GPGPU est devenue très essentielle dans certains domaines pour améliorer la capacité des systèmes, pour atteindre un temps minimale d'exécution et pour exploiter les cœurs d'architecture GPU dans le calcul scientifique [39,40]. En plus, nous avons décrit les architectures GPU NVIDIA et les architectures AMD pour détailler l'architecture du GPU et le fonctionnement de ce dernier.

D'après ce que nous avons vu dans ce chapitre, nous remarquons qu'il y a beaucoup de détails d'ordonnancement, de synchronisation, de communication et de temps qui sont très importants au développement des applications. Pour cela la définition et la spécification de ces critères nécessitent des outils spécialisés pour être traitées par la suite. Dans le deuxième chapitre, nous présentons quelques méthodes de spécification qui sont appropriés pour spécifier les systèmes sur puces contenant des GPUs.

Chapitre 2 :
Les méthodes de spécification

Introduction

Les systèmes embarqués et temps réel sont des systèmes informatiques(SETR), leur fonctionnement est contrôlé par des contraintes temporelles. Le système temps réel est divisé en tâches qui utilisent des ressources partagées (processeurs, mémoires,…etc.) ce qui nécessite des techniques d'ordonnancement afin de finaliser l'exécution du système temps réel[41]. Avec l'évolution des technologies de traitement, il y a des architectures multi cœurs qui permettent une exécution multitâches à la fois. Le GPU est l'une des architectures multitâche ayant optimisé le temps et la performance des systèmes grâce à sa structure multi-cœurs. La complexité de ces systèmes nécessite des outils spécifiques pour la conception et le développement. Le système temps réel (STR) doit suivre un cycle de vie. Le cycle de vie d'un logiciel STR consiste à faire : 1) la spécification, 2) la conception, 3) le codage, 4) Mise en œuvre et les tests unitaires et la validation[13].

La spécification et la conception sont des étapes primordiales dans le cycle de vie d'un système. Pour cela les chercheurs ont toujours consacré leurs temps à développer et à créer des méthodes et des outils assurant une bonne spécification. Ce chapitre a pour objectif de présenter deux axes très importants dans la spécification : une spécification semi-formelle en UML et MARTE et une spécification formelle en Event B. Au cours de ce chapitre, nous étudions de façon approfondie les méthodes de spécifications formelles et la spécification semi-formelle en UML et MARTE.

1 Approche d'Ingénierie Dirigée par les Modèles (IDM)

L'approche IDM (en anglais MDE : Model Driven Engineering) considère qu'on peut atteindre des gains substantiels en traitant automatiquement l'information contenue dans des ***modèles***, dans le but d'***automatiser*** des tâches de développement. IDM a pour but d'automatiser l'implémentation par la transformation des modèles de spécification en code exécutable. Il y a trois niveaux de base des approches IDM : le modèle, le méta-modèle et le méta méta-modèle. Le tableau 4 montre la différence entre les niveaux de l'approche IDM [42].

Modèle	Méta-modèle	Méta méta-modèle
Un modèle est une représentation simplifiée (ou abstraction) d'un système réel.	Un méta modèle est un langage qui permet d'exprimer des modèles. Dans un méta modèle, on définit les concepts ainsi que les relations entre les concepts permettant d'exprimer des modèles.	Les éléments du langage d'expression d'un méta modèle doivent aussi satisfaire un ensemble de règles et contraintes. Ces règles et contraintes sont exprimées sous forme d'un méta langage ou méta méta modèle.

Tableau 4. Niveaux principale de l'approche IDE

Comme illustré dans la Figure 2.1, dans l'approche MDE, un modèle de transformation est un processus de compilation qui transforme un modèle de source en un modèle cible. Un modèle de transformation est basé sur un ensemble de règles. Chaque règle identifie les concepts des méta-modèles de la source et de la cible. Ces derniers facilitent l'extensibilité du processus de compilation. D'autres règles étendent le processus de compilation et chaque règle peut être modifiée indépendamment. Les règles sont représentées à l'aide des langages spécifiques. Le langage peut être impératif (décrire comment une règle s'exécute) ou déclaratif (décrire ce qui est créé par la règle). Les langages déclaratifs sont souvent utilisés dans l'approche MDE parce que les objectifs des règles peuvent être spécifiés d'après l'exécution. Une représentation graphique est une bonne approche pour représenter les règles exprimées par un langage déclaratif. Il y a deux types de transformation : *model to model* et *model to text* [32].

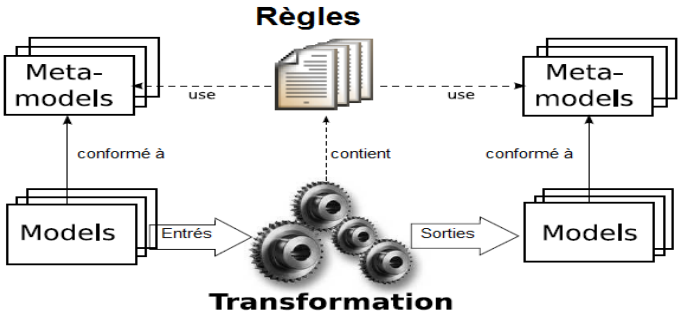


Figure 2.1. Illustration de transformation de modèle

2 Spécification semi-formelle : UML/MARTE

2.1 UML

UML signifie Unified Modeling Language, qui se veut un instrument de capitalisation des savoir-faire puisqu'il propose un langage qui soit commun à tous les experts du logiciel, il va dans le sens de cet assouplissement des contraintes méthodologiques [43]. UML est le langage de modélisation de l'approche objet, standard utilisé pour la conception et la modélisation des systèmes. UML est défini pour modéliser divers types de systèmes, de taille quelconque et pour tous les domaines d'applications (gestion, scientifique, temps réel, système embarqué) [44]. Comme illustré dans la Figure 2.2, UML offre un nombre de diagrammes modélisant chaque aspect du système. Ces diagrammes peuvent être divisés en deux catégories :

- ***Diagrammes structurels ou diagrammes statiques (UML Structure) :*** Diagramme de classes, diagramme d'objets, diagramme de composants, diagramme de déploiement, diagramme de paquet (Package), diagramme de structures composites.
- ***Diagrammes comportementaux ou diagrammes dynamiques (UML Behavior):*** Diagramme de cas d'utilisation, diagramme d'activités, diagramme de machines à état, diagramme de séquences, diagramme de communication, diagramme global d'interaction, diagramme de temps [45].

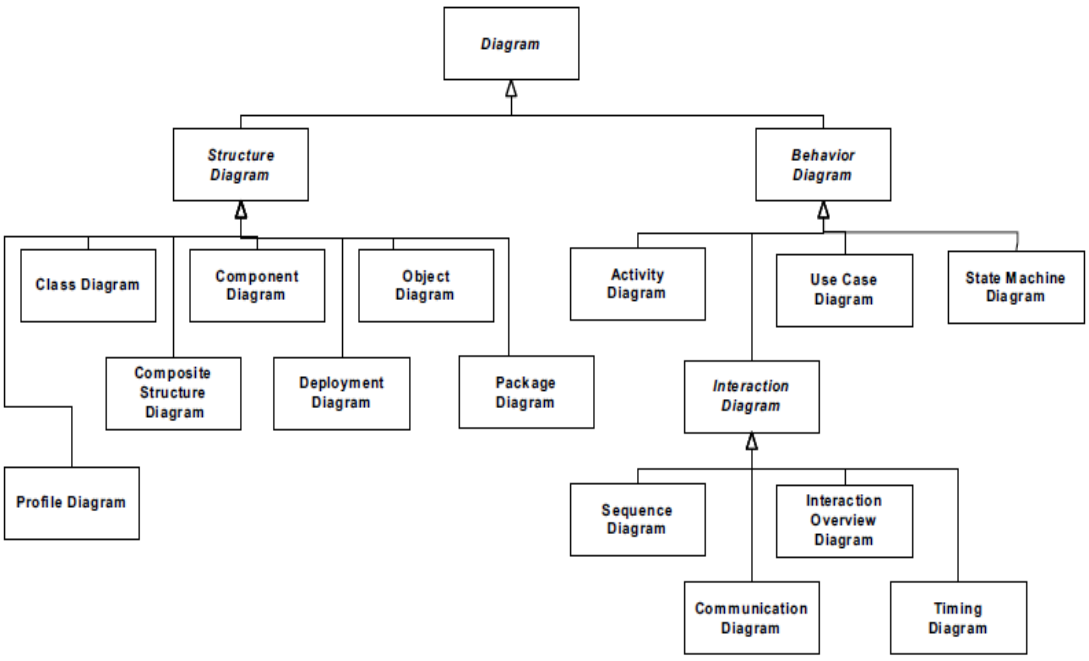


Figure 2.2. Classification des diagrammes d'UML

UML est un langage unifié pour la modélisation objet.

- UML est un **langage** (**de modélisation** objet), il propose une notation et une sémantique associée à cette notation (i.e. des modèles), sans processus (i.e. de démarche proposant un enchaînement d'étapes et d'activités menant à la résolution d'un problème posé) ; UML n'est donc pas une méthode.
- UML **unifie** des méthodes objet, et plus particulièrement les méthodes Booch'93 de G. Booch, OMT-2 (Object Modeling Technique) de J. Rumbaugh et OOSE (Object-Oriented Software Engineering) d'I. Jacobson. UML reprend, en particulier, les notions de partitions en sous-systèmes de Booch'93, de classes et d'associations d'OMT-2, et d'expression des besoins par les interactions entre les utilisateurs et le système d'OOSE.
- UML a une approche entièrement couvrant tout le cycle de développement : analyse, conception et réalisation **objet** et non fonctionnelle. Le système est décomposé en objets collaborant (plutôt qu'en tâches décomposées en fonctions plus simples à réaliser) [44].

2.2 Le profile MARTE

MARTE (Modeling and Analysis of Real Time Embedded Systems) est défini par le Groupe de travail ProMARTE de l'OMG pour le développement et l'analyse dirigée par les modèles des systèmes embarqués temps réel. MARTE vise à remplacer le profil UML SPT (Schedulability, Performance, Time). Il est basé sur le méta-modèle UML2.0, Object Constraint Language OCL2 et Meta-Object Facility Query/ View/ Transformation MOF 2.0QVT [46].

MARTE traite de nouvelles exigences telles que la spécification des modèles logiciels et matériels, la séparation entre modèles abstraits d'applications et plates-formes d'exécution, la modélisation de l'allocation d'un modèle d'application sur un modèle de plate-forme, ainsi que la modélisation de différentes notions de temps et de propriétés extra-fonctionnelles. Il hérite de plusieurs aspects UML tels que les composants pour la conception structurale, les FSMs hiérarchiques et des data-flow graphs pour les comportements. Il fournit, de plus, des fonctions d'annotation standard appelées "stéréotypes" pour représenter des propriétés fonctionnelles et extra-fonctionnelles, ainsi que des moyens pour en introduire de nouvelles propriétés. En conséquence, une spécialisation plus poussée de MARTE permettrait l'encodage des

notions IP-XACT[1] telle que Vendor Extension, de manière accessible et interprétable. Les méthodes issues de l'ingénierie des modèles concernant les transformations de modèle devraient ensuite permettre à l'information pertinente d'être redirigée vers n'importe quel outil propriétaire capable de comprendre ce format public. MARTE intègre un modèle de temps logique permettant la description de contraintes liées à différentes vues d'un modèle comme le timing, la sécurité ou encore la consommation et éventuellement une méthode assistée de rangement entre différents niveaux de représentation de modèles. La modélisation des structures répétitives (RSM) définie dans le profil standard UML MARTE simplifie également la représentation de structures paramétrables complexes [47].

Comme illustrée en Figure 2.3, l'architecture MARTE est fondée sur quatre paquetages : le paquetage de base, le modèle de conception, le modèle d'analyse, et les annexes.

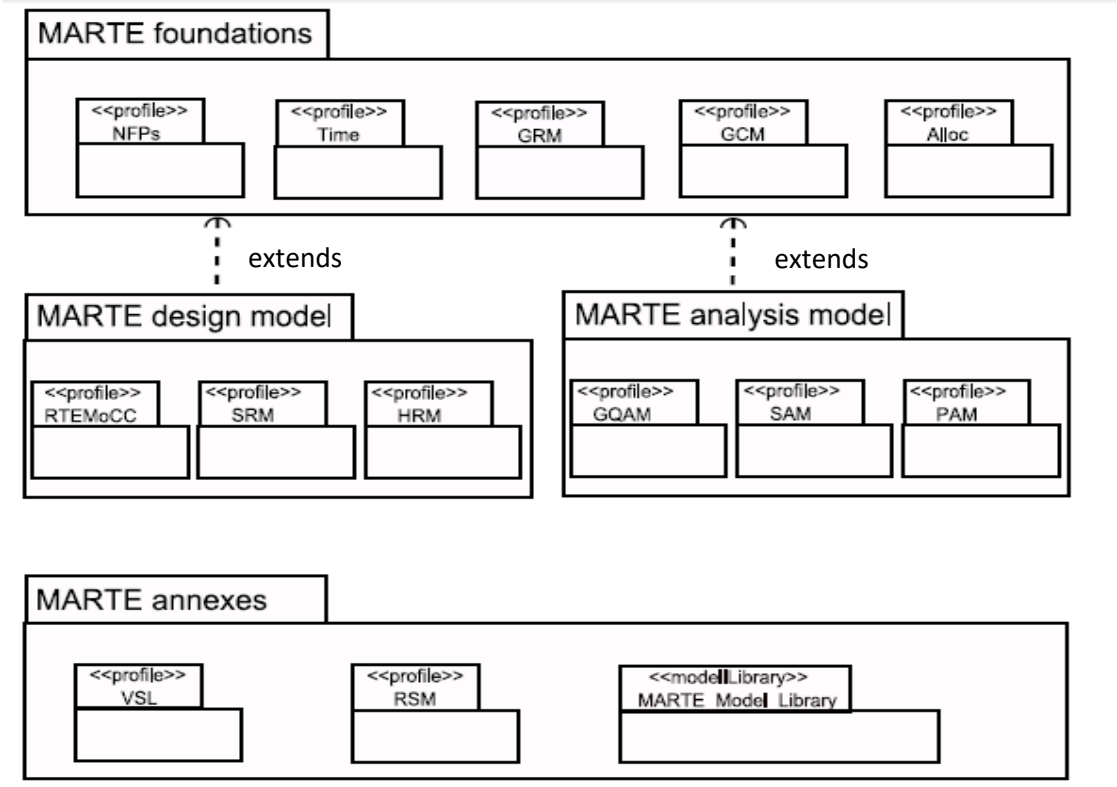


Figure 2.3. Architecture du profil MARTE

MARTE distingue l'aspect matériel de l'aspect logiciel du système. Ce qui fournit une méthode flexible pour échanger les parties logiciels et les parties matériels.

[1] Un design IP-XACT est un assemblage d'objets composant. Il représente un système ou un sous-système définissant l'ensemble des instances de composants et leurs interconnexions.

D’autre part, il permet de tester la partie logicielle sur différents types d’architectures matérieles. En comparant avec UML, MARTE peut décrire le matériel d’une façon bien structurée grâce à la notion de composant [46,48].

2.2.1 Les profils de base de la spécification MARTE

- **Core Element :** décrit les concepts de base du cœur dans la spécification MARTE tels que les modèles, les éléments de modèles (comme classifiers) et leurs comportements associés. Le comportement peut être un état de machine, diagramme d’activité, une interaction (diagramme de séquence/ Timing diagramme), un automate, etc.
- **Non Functional Properties (NFP) :** permettent de décrire les propriétés qui ne sont pas reliées à des aspects fonctionnels tels que les ressources consommées, la mémoire utilisée, etc.
- **Time Modeling :** Le package Time permet de spécifier les concepts utilisés dans le domaine synchrone tel que le temps discret dans les systèmes temps réel comme les FPGA. Le plus important dans ce package est l’utilisation des contraintes temporelles dans les modèles UML de comportement comme le digramme de séquence et le digramme d’état-machine.
- **Generic Ressource Modeling (GRM) :** ce package introduit le concept des ressources et les services de ressources. Les ressources peuvent être classifiées par types : ressources de calcul, ressources de stockage et ressources de synchronisation. Les ressources peuvent être gérées et ordonnancées. Cette notion permet de modéliser les ressources partagées et exclues mutuellement.
- **Allocation :** Ce package permet l’allocation du modèle d’application sur le modèle d’architecture.
- **Generic Component Modeling (GCM) :** ce package permet de définir les concepts tels que les composants, les ports et les instances. Les concepts SysML du diagramme de bloc et des ports ont été intégrés dans MARTE.
- **High Level Application Modeling (HLAM) :** ce package permet de distribuer les caractéristiques et les fonctionnalités reliées au système Temps Réel[49].
- **Detailed Resource Modeling (DRM) :** ce package est composé de deux sous packages :

- **Software Resource Modeling (SRM) :** il est utilisé pour décrire les parties standardisées ou conçues basées sur les APIs de temps réel tel que le multi-threading.
- **Hardware Resource Modeling (HRM) :** les concepts du matériel permettent de représenter l'architecture MARTE par différentes vues. Les vus peuvent être fonctionnelles, physiques ou hybrides.

- **Generic Qualitative AnalysisModeling (GQAM) :** permet aux concepteurs de se concentrer sur l'analyse à travers le profil MARTE. L'analyse peut être sur le comportement du logiciel comme la performance et la planification ou l'ordonnancement en plus des aspects tels que la force, l'énergie, la tolérance aux fautes, etc. Le package GQAM fournit la description de l'utilisation du comportement de système des ressources disponibles.
- **Schedulability Analysis Modeling (SAM) :** le package SAM est une extension du package GQAM pour l'analyse de l'ordonnancement optimisant le système.
- **Performance Analysis Modeling (PAM) :** ce package est une autre extension du package GQAM pour l'analyse des propriétés temporelles des systèmes embarqué et Temps réel.
- **Value Specification Language (VSL) :** ce langage est utilisé dans les contraintes NFPs (Non functional properties). VSL définit le type d'informations, les paramètres, les constantes et les expressions. Les expressions VSL peuvent être utilisées pour spécifier les paramètres non fonctionnels, les valeurs, les opérations et les dépendances entre les différentes valeurs dans le système.
- **Repetitive Structure package (RSM) :** permet de représenter les applications massivement parallèles, les architectures de Grid, les topologies d'interconnexion (NOCs et les réseaux d'interconnexion multi-niveaux) à travers les expressions.
- **Clock handling Facilities :** cette annexe fournit la syntaxe abstraite pour spécifier les dépendances d'horloge et les valeurs d'horloge.
- **Les librairies MARTE :** cette annexe définit les librairies de MARTE pour les types primitifs, les types d'informations, les énumérations et les concepts de temps[49].

MARTE a apporté de nombreux avantages en fournissant un support pour la spécification, l'analyse, la conception, la vérification, et la validation des systèmes. Il fournit un moyen de modélisation des aspects matériels et logiciels de systèmes embarqués temps réel en vue d'améliorer la communication entre les développeurs, tout en favorisant la construction de modèles pouvant être utilisés pour faire des prédictions quantitatives [46].

2.2.2 Gaspard2

Gaspard 2 (Graphical Array Specification for Parallel and Distributed Computing) est un environnement de développement intégré pour co-modéliser, simuler, tester et générer du code logiciel et matériel pour des applications intensives sur SoC [50,51,52].

Le but de ce dernier est de permettre une conception indépendante des concepts d'implémentation, c'est-à-dire permettre la conception du système à partir d'un modèle indépendant de la cible finale d'implémentation [53]. Gaspard2 est un frawework de conception de SoC orienté MDE. Bien qu'il utilise un sous-ensemble étendu de MARTE, Gaspard2 a fortement contribué au développement du profil. Le profil MARTE RSM, par exemple, avec son modèle de calcul associé Array-OL, est directement inspiré de Gaspard2. Les profiles HRM et Allocation ont également profité de certains aspects déjà présents dans cet outil [54].

Il est similaire à l'environnement de programmation Eclipse. La Figure 2.4 montre l'interface de Gaspard 2.

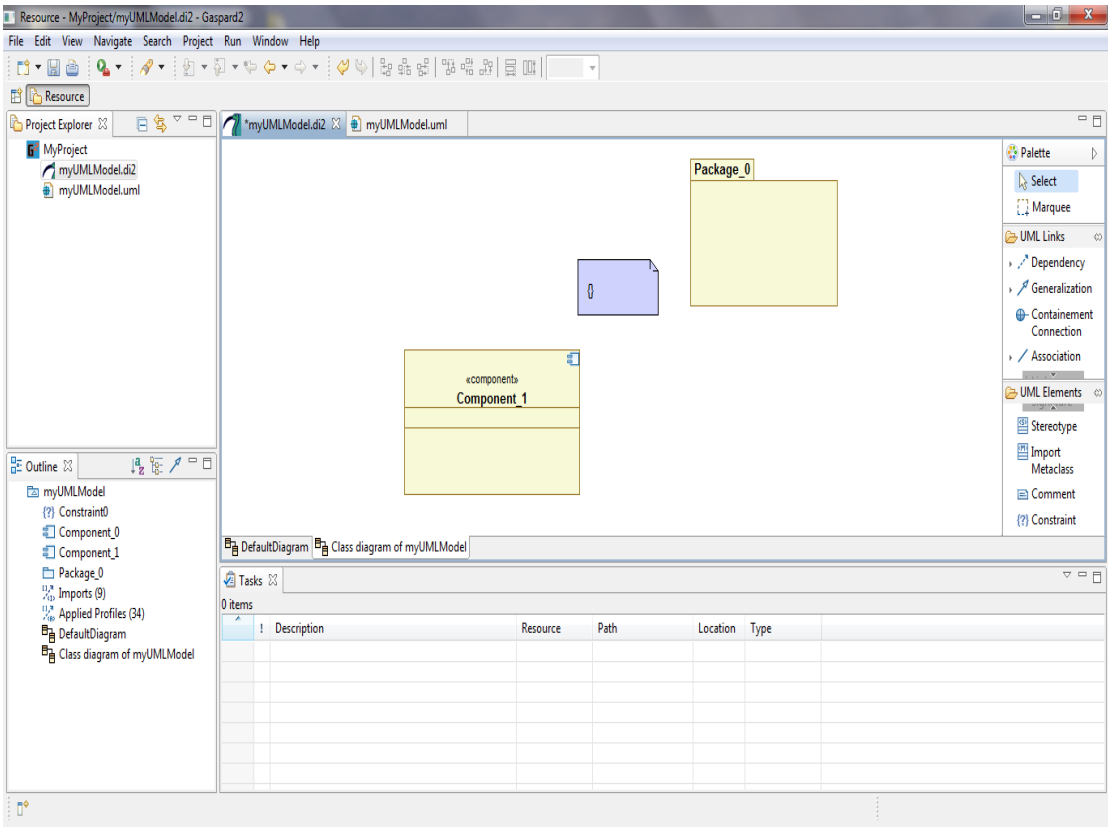


Figure 2.4. Interface du logiciel Gaspard2

Pour modéliser un système embarqué, l'utilisateur de Gaspard2 doit saisir quelques diagrammes UML décrivant la fonctionnalité à réaliser et les ressources matérielles à exploiter. Ce modèle UML est conforme au profil MARTE. L'utilisateur choisit ensuite une des cibles implémentées au sein de Gaspard2 en vue de générer le code correspondant au modèle qu'il a défini. Un moteur transforme alors le modèle initial en code pour la cible, en faisant passer le modèle par une suite de transformations automatiques, visant à l'enrichir, le complexifier et le rendre plus dépendant de la cible choisie, jusqu'à la dernière étape qui consiste à transformer le modèle en code source (sous forme de fichier texte) correspondant à cette cible. Gaspard2, dans son état actuel, implémente les cibles suivantes : SequentialC, PThread, OpenCL, OpenMP, SystemC, Lustre, VHDL [53]. Comme illustré dans la Figure 2.5, le flot de conception des systèmes embarqués dans GASPARD2 suit les étapes suivantes :

- **La modélisation du système :** Les trois premiers modèles dans GASPARD2 sont basés sur le profil MARTE. Le modèle d'application décrit les tâches logicielles et matérielles. Le modèle de l'architecture décrit l'architecture sur

laquelle l'application s'exécutera. Le modèle d'allocation est utilisé afin de faire le lien entre les deux modèles. Ensuite, le modèle de déploiement, basé sur le profil GASPARD2, vient relier chaque composant élémentaire de l'application ou de l'architecture à un code qui existe déjà dans une bibliothèque d'IP.

- **Les transformations de modèles :** organisées en chaînes de transformations qui peuvent partager les transformations liées à des notions communes entre les cibles.
- **La génération de code** : A la fin d'une chaîne de transformations, on obtient un modèle PSM (Platform Specific Model) avec des détails techniques permettant la génération du code lié à la technologie cible. GASPARD2 permet la génération de code ciblant différents langages tels que Fortran, Lustre, SystemC, OpenCL, C et VHDL [55].

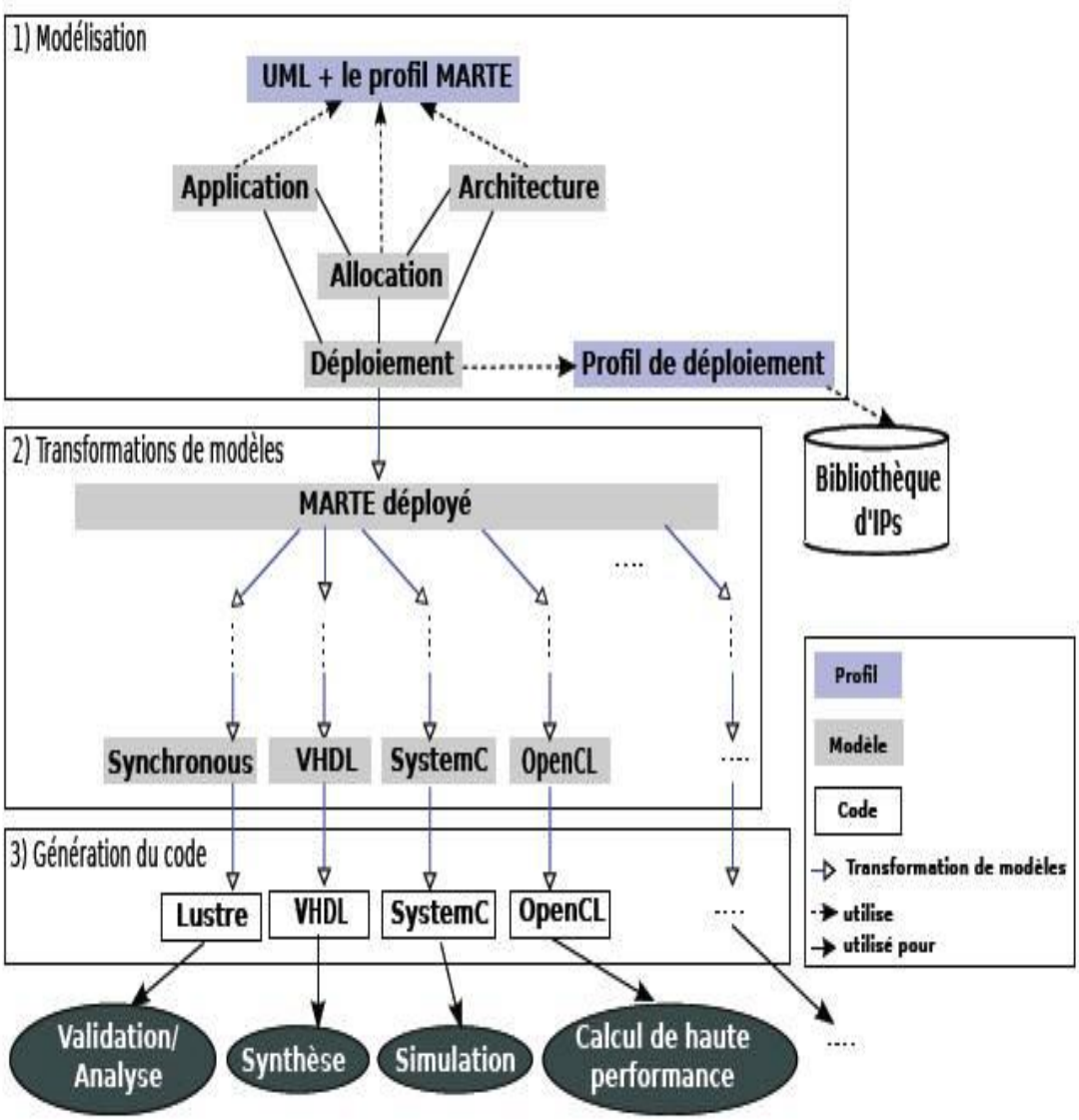


Figure 2.5. Le flot de conception des systèmes embarqués sous Gaspard2

3 Spécification formelle

3.1 Définitions

Une méthode formelle au sens le plus large est une méthode d'ingénierie pour le développement de systèmes basés sur des concepts logiques et mathématiques rigoureux et bien identifiés. Une méthode formelle définit généralement un ou plusieurs langages de développement non ambigus et des principes de raisonnement explicites, justifiés et vérifiables [57,58].

Les méthodes formelles sont des techniques permettant de raisonner rigoureusement, à l'aide de la logique mathématique sur des programmes afin de démontrer leur validité par rapport à leur spécification. Ces méthodes permettent d'obtenir une forte assurance de l'absence de bug. Elles sont coûteuses en ressources humaines et matérielles et actuellement réservées aux logiciels les plus critiques [57,58].

Les méthodes formelles permettent de spécifier rigoureusement un système à l'aide d'un modèle, et d'en décrire précisément les propriétés. Elles permettent également d'analyser avec précision ces propriétés [56].

3.2 Eléments caractéristiques des méthodes formelles

3.2.1 Capacité de raffinement

La modélisation d'un système demande du temps et une certaine expertise. Cependant, pour procéder à des analyses très tôt sur un système, le modèle créé ne reflètera pas le système final : il est courant de raffiner les exigences ou les propriétés d'un système tout au long de son cycle de développement. De ce fait, il est important de pouvoir ajouter à moindre coût des informations sur le modèle formel, tout en réutilisant les analyses effectuées sur un modèle moins restreint.

3.2.2 Composition

Le développement d'un système met en œuvre différentes équipes travaillant séparément dont les travaux sont ensuite assemblés pour construire le système final [58,59].

3.2.3 Méthodes d'analyse

Il existe quatre catégories de méthodes d'analyse : la simulation, le test, la vérification par inférence, le model-checking.

Faire de la simulation revient à faire des expérimentations sur un modèle du système. Cela implique d'avoir à disposition des informations sur le matériel en jeu, ce qui peut arriver tard dans le processus de développement de l'application répartie [57,59].
Elles permettent de réaliser différents types d'analyses :

- L'analyse **«statique»** permettant de vérifier la cohérence d'une description en termes de typage et de composition d'interface ;
- L'analyse **«dynamique»** traitant plus spécifiquement de l'évolution comportementale du système au cours du temps.

3.2.4 Automatisation

Certaines méthodes formelles offrent la possibilité de procéder à des analyses de façon automatique telle que le model-checking ; d'autres nécessitent l'intervention d'experts. Dans l'optique de la réalisation d'un guide de sélection des méthodes formelles dans un processus de développement, il est utile de comparer ces méthodes sur cette caractéristique, puisqu'elle aura un impact fort sur le coût de mise en œuvre [59].

3.3 Réseaux de Petri

Ces réseaux présentent des caractéristiques intéressantes telles que la modélisation et la visualisation de comportements parallèles, de la synchronisation et partage de ressources. C'est un outil de modélisation utilisé généralement en phase préliminaire de conception de système pour sa spécification fonctionnelle, leur modélisation et leur évaluation. Il permet notamment :

- la modélisation des systèmes informatiques,
- l'évaluation des performances des systèmes discrets, des interfaces homme-machines,
- la commande des ateliers de fabrication,
- la conception de systèmes temps réel
- la modélisation des protocoles de communication,
- la modélisation des chaines de production [60,61].

Un Réseau de Petri (RdP) est un graphe orienté comprenant deux types de sommets : les places et les transitions. Un RdP est défini par un ensemble d'éléments appelés *nœuds* ou *sommets* et un ensemble de relations appelées *arrêtes* ou *arcs*.

Place Transition

Comme illustré dans la Figure 2.6, un arc relie soit une place à une transition, soit une transition à une place jamais une place à une place ou une transition à une transition. Tout arc doit avoir à son extrémité un sommet (place ou transition) [61].

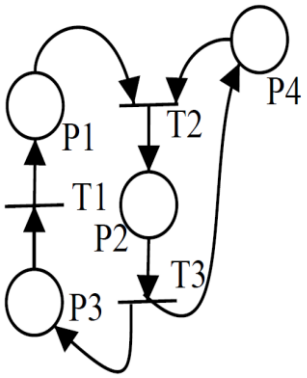


Figure 2.6. Exemple de Réseau de Petri

3.4 Automates temporisés

Les automates temporisés ont été introduits par Alur et Dill en 1990. Ce modèle permet de faire de la vérification de propriétés de sûreté sur des systèmes de transitions utilisant des variables temporelles à valeurs dans N, Q_+ ou même R_+. Les automates temporisés ont été implantés dans plusieurs logiciels de vérification tels que Kronos, Uppaal et CMC. Ils permettent de valider ou corriger un certain nombre de protocoles ou algorithmes utilisant des variables temporelles dans leur déroulement [62].

Il s'agit d'automates classiques munis d'horloges qui évoluent de manière continue et synchrone avec le temps. Chaque transition contient une *garde* sur la valeur des horloges décrivant quand la transition peut être exécutée et un ensemble d'horloges qui doivent être remises à zéro lors du franchissement de la transition. Chaque état de contrôle contient un *invariant*, une contrainte sur les horloges, qui peut restreindre le temps d'attente dans l'état et forcer l'exécution d'une transition d'action [63].

3.5 La méthode formelle B

La méthode B, proposée par Jean-Raymond ABRIAL en 2005, est une méthode formelle couvrant toutes les phases d'un cycle de développement. En effet, les modèles B de la Figure 2.7 expriment différents niveaux d'abstraction allant de la phase de conception préliminaire jusqu'à la phase de codage. Le passage d'une phase à une autre

dans un processus de développement en B correspond à un incrément de spécifications en cours de développement [64,65].

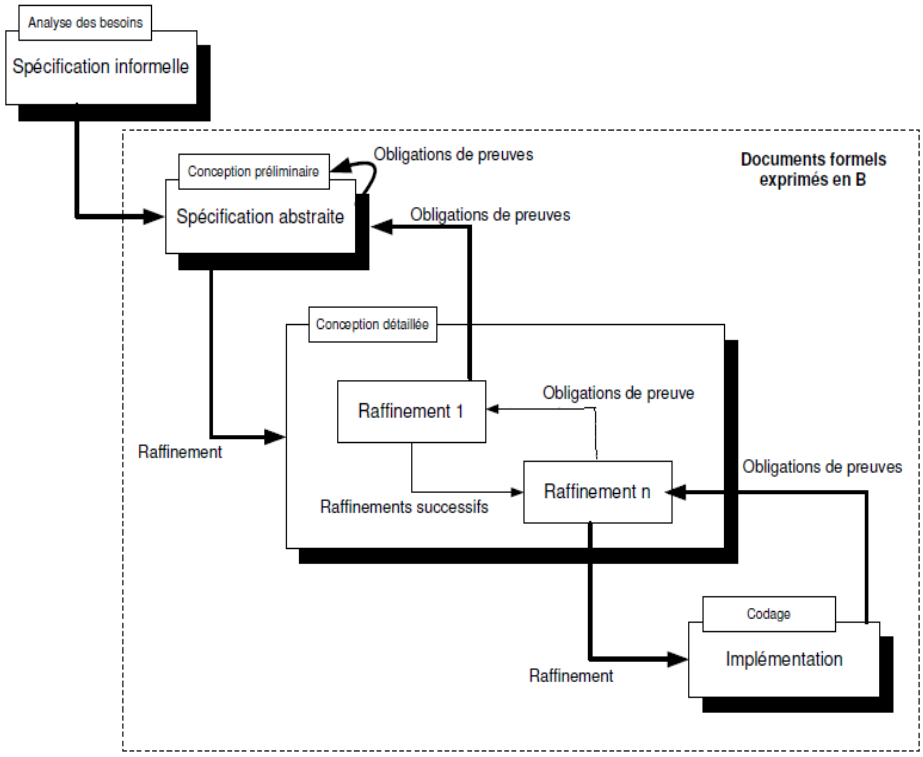


Figure 2.7. Processus de développement en B

La méthode B s'appuie sur les concepts mathématiques de la théorie des ensembles. Elle est basée sur les notions de machine abstraite, de raffinement vérifié par la preuve. Le cycle de développement commence par la construction d'une machine abstraite de niveau N0. Cette machine abstraite B est constituée de variables d'état, d'un invariant exprimant des propriétés sur les variables et d'opérations décrivant les transformations d'états correspondant à des changements de valeur des variables. Le développement d'un système complexe ne peut se faire en une seule étape et B propose une méthode de construction incrémentale par raffinements successifs. L'abstraction est une technique générale employée pour maîtriser la complexité des systèmes, un développement en B consiste, alors, à construire un premier modèle abstrait, dans lequel certains détails ne sont pas pris en compte. Les étapes suivantes consistent à compléter le modèle abstrait pour obtenir un modèle concret. A l'issue de chaque étape, la vérification consiste à prouver des obligations de preuves engendrées par l'outil de développement [66,67,68].

3.6 Event B

Event-B (B événementiel) est une évolution du langage B[68]. Un modèle B Événementiel décrit un système états/transitions où les variables représentent l'état du système et les évènements représentent les transitions d'un état à un autre. Un modèle est caractérisé par l'ensemble des séquences d'événements autorisés sous couvert des propriétés énoncées. La description du modèle B Événementiel est accompagnée de la génération automatique d'obligations de preuves qui doivent être déchargées afin d'assurer la preuve de correction du modèle.

Comme illustré dans la Figure 2.8, un modèle B Événementiel est constitué d'une partie statique représentée par le composant CONTEXT et d'une partie dynamique représentée par le composant MACHINE. Un modèle peut contenir uniquement des contextes, uniquement des machines, ou les deux à la fois [68,69].

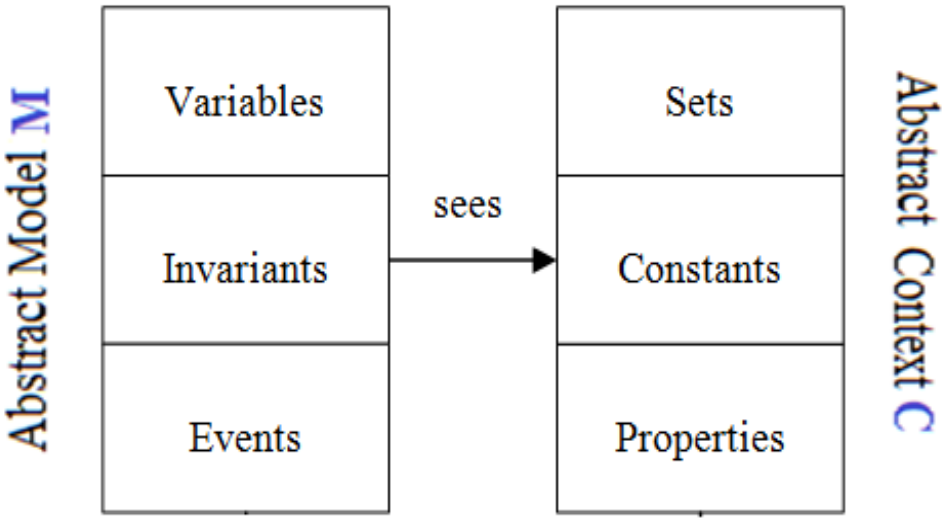


Figure 2.8. Modèle Event B

3.6.1 Notions de base d'Event B

Comme pour la méthode formelle B, les machines abstraites, le raffinement et les substitutions généralisés sont au cœur du langage Event-B. Ces notations ont vu certains changements qui seront expliqués par la suite.

3.6.1.1 Le composant Context

Le composant CONTEXT ,illustré dans la Figure 2.9, formalise la partie statique du système modélisé. Il contient les définitions des types abstraits et des constantes ainsi que des propriétés liées aux constantes. Il est composé des clauses suivantes :

– CONTEXT représentant le nom du composant, unique dans un modèle.

– EXTENDS déclarant la liste des contextes qu'étend le contexte décrit.

– SETS définissant un ensemble de types abstraits ou d'ensembles énumérés.

– CONSTANTS décrivant les noms de constantes utilisées par le modèle.

– AXIOMS exprimant des propriétés liées aux constantes à l'aide d'expressions de la logique du premier ordre. Il s'agit des hypothèses considérées lors du processus de preuve.

– THEOREMS exprimant un ensemble de propriétés pouvant être déduites à partir des axiomes [69].

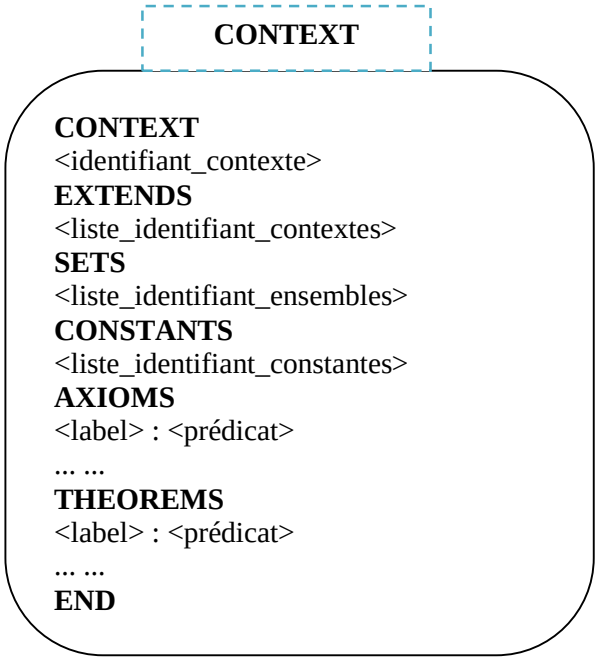


Figure 2.9. Structure du CONTEXT

3.6.1.2 La MACHINE

Le composant MACHINE ,illustré dans la Figure 2.10, formalise la partie dynamique, le comportement, du système modélisé.

Elle définit l'état du système, ses propriétés et les événements qui le font évoluer d'un état à un autre. La MACHINE est composée par les clauses suivantes :

– ***MACHINE*** représentant le nom du composant unique dans un modèle.

– ***REFINES*** déclarant le nom de la machine éventuellement raffinée par la machine décrite.

– ***SEES*** déclarant les contextes référencés explicitement par la machine. La machine peut utiliser les ensembles et variables définis dans les contextes référencés explicitement ou implicitement (contextes étendus par le contexte référencé).

– ***VARIABLES*** décrivant les noms des variables du modèle.

– ***INVARIANTS*** exprimant les propriétés invariantes du modèle à l'aide de prédicats de la logique du premier ordre, ces propriétés consistent en des propriétés de typage des variables ainsi que les propriétés de sûreté du modèle. Les invariants doivent être préservés par les événements de la clause EVENTS et vérifiés dans la machine et ses raffinements.

– ***THEOREMS*** exprimant un ensemble de propriétés pouvant être déduites à partir des invariants.

– ***VARIANT*** définissant l'expression du variant du modèle, une variables entière naturelle décroissante définissant l'ordre de prise de contrôle des évènements convergents.

– ***EVENTS*** déclarant les événements pouvant prendre le contrôle dans le modèle (transitions entre états). Chaque événement est décrit par une garde et par un corps constitué des substitutions traduisant le changement d'état du système. Un événement prend le contrôle lorsque sa garde est évaluée à vrai. Une machine possède un événement particulier correspondant à l'initialisation du système modélisé [64,69].

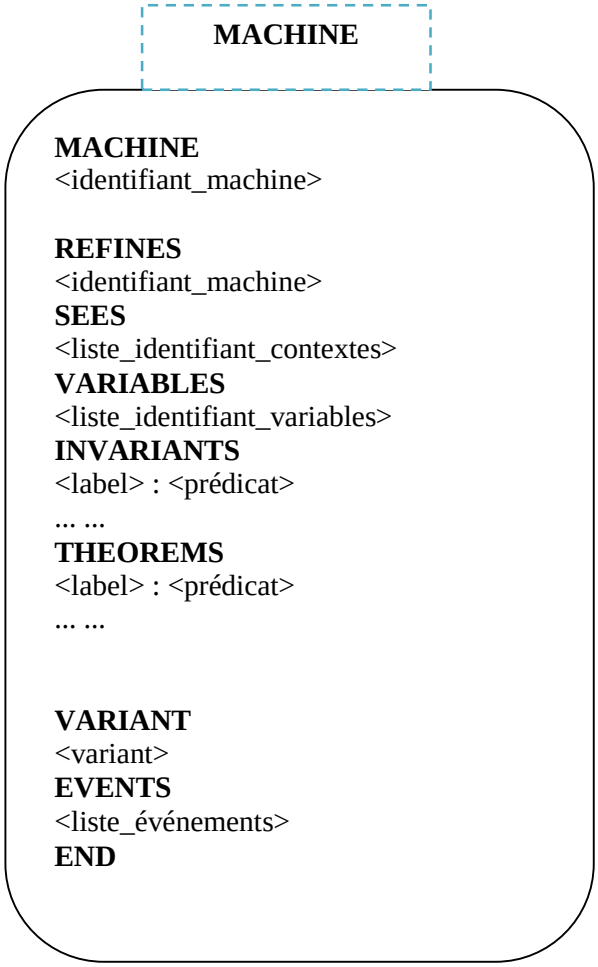


Figure 2.10. Structure de MACHINE

3.6.1.3 Evénement Event B

Un événement Event-B correspond à un changement d'état dénotant une transition dans le système modélisé. Un événement est essentiellement composé d'une garde (la clause WHEN) qui définit les conditions nécessaires au déclenchement de l'évènement et d'une action (la clause THEN) qui définit l'évolution des variables d'état. Notons que plusieurs gardes d'événements peuvent être vraies en même temps. Néanmoins, un seul événement peut se déclencher et le choix de cet événement est non déterministe. Un évènement peut posséder des paramètres, des variables locales, définis dans la clause ANY. En Event-B, on peut distinguer trois formes d'événements :

- ✓ *La **forme simple*** inclut seulement une action. Cela équivaut donc à une garde toujours vraie.

Event $nom \mathrel{\widehat{=}}$
begin
act : $x :\mid P(x,x')$
end

- ✓ *La **forme gardée*** inclut une garde et une action qui ne dépendent que des variables d'états du modèle.

Event $nom \mathrel{\widehat{=}}$
when
grd : $G(x)$
then
act : $x :\mid P(x,x')$
end

- ✓ *La **forme complète*** inclut des paramètres, une garde et une action.

Event $nom \mathrel{\widehat{=}}$
any
prm : t
where
grd : $G(x,t)$
then
act : $x :\mid P(x,x',t)$
end

Il existe aussi un évènement *skip* avec une action et une garde vide. Il existe un évènement obligatoire, nommé INITIALISATION est toujours de forme simple permettant d'initialiser le système en spécifiant les valeurs initiales possibles tout en respectant les invariants [70,71].

3.6.1.4 Raffinement

Le raffinement en B est une technique de développement incrémental permettant de préciser progressivement les données et les opérations de la spécification abstraite de départ. Le raffinement d'une machine abstraite est une transformation produisant une machine qui conserve la même interface (opération) et le même comportement que la machine abstraite initiale. Elle permet d'une part, de reformuler la machine en une expression de plus en plus concrète, et d'autre part, de l'enrichir en y ajoutant de nouvelles données et invariants. En B, le raffinement porte sur :

- **les données :** les données ou variables abstraites définies dans la clause VARIABLES peuvent être conservées, modifiées ou disparaître et de nouvelles variables peuvent être introduites. Dans ce dernier cas, un invariant, dit de liaison ou de collage, est défini en vue de spécifier la relation entre les données abstraites et les données concrètes.
- **les opérations :** chaque opération raffinée doit réaliser ce qui est spécifié dans l'abstraction, à l'aide des données du raffinement et de substitutions plus concrètes et plus déterministes [69,70,71].

3.6.1.5 Obligations de preuve

Afin de garantir la correction d'un modèle, il est indispensable de le prouver. Pour y parvenir, un outil de la plateforme RODIN appelé ***générateur d'obligations de preuve*** génère automatiquement des *obligations de preuve*. Une obligation de preuve définit ce que doit être prouvé pour un modèle. Il s'agit d'un prédicat dont on doit fournir une démonstration pour vérifier un critère de correction sur le modèle.

L'outil RODIN vérifie statiquement les contextes et les machines et génère des ***séquents***, un nom générique pour ce qu'on veut prouver. Il est de la forme $\boldsymbol{H} \vdash \boldsymbol{G}$: 'le but G est à démontrer en partant de l'ensemble H des hypothèses' [72,73,74].

Les règles des obligations de preuve sont au nombre de onze (11). Pour chaque règle, le générateur d'obligations de preuve génère une forme spécifique du séquent.
Etant donné un élément de modélisation ; un événement **evt**, un axiome **axm**, un théorème **thm**, un invariant **inv**, une garde **grd**, une action **act**, un variant ou témoin

(witness) **x**, les règles d'obligation de preuve pouvant être générées pour ces éléments sont:

- ***INV*** : règle de préservation de l'invariant assurant que chaque invariant dans une machine donnée est préservé par tous les événements. Elle est de la forme : *"evt/inv/INV."*
- ***FIS :*** rassurer qu'une action non déterministe est faisable. La forme est : *"evt/act/FIS."*
- ***GRD :*** renforcement des gardes abstraites ; les gardes des événements concrets sont plus fortes que celles des abstractions. Elle est de la forme : *"evt/grd/GRD."*
- ***MRG :*** la garde d'un événement concret fusionnant deux événements abstraits est plus forte que la disjonction (ou logique) des gardes de ces deux événements abstraits. Le nom de cette règle est : *"evt/MRG."*
- ***SIM :*** chaque action dans un événement abstrait est correctement simulée dans le raffinement correspondant. Autrement dit, l'exécution d'un événement concret n'est pas en contradiction avec son abstraction. Le nom de la règle est : *"evt/act/SIM."*
- ***NAT :*** sous une condition que les gardes d'un événement convergent ou anticipé sont vérifiées, le variant numérique proposé est un *entier naturel*. Son nom est : *"evt/NAT."*
- ***FIN :*** dans une condition où les gardes d'un événement convergent ou anticipé sont vérifiées, l'ensemble variant proposé est un ensemble *fini*. Son nom est : *"evt/FIN"*.
- ***VAR :*** chaque événement convergent diminue le variant numérique proposé ou l'ensemble variant proposé. De plus, chaque événement anticipé n'augmente pas le variant numérique proposé ou l'ensemble variant proposé. Son nom est : *"evt/VAR."*
- ***WFIS :*** chaque témoin proposé dans la clause *WITH* d'un événement concret existe vraiment. Son nom est : *"evt/x/WFIS."*
- ***THM :*** un théorème écrit dans une machine ou dans un contexte est vraiment prouvable. Le nom de telle règle est : *"thm/THM "*.
- ***WD :*** règle de bonne définition des axiomes, théorèmes, invariants, gardes, actions, variant et witness. Selon la nature de l'élément, les noms pour cette règle sont : *"axm/WD "," thm/WD ", " inv/WD ", " grd/WD ", " act/WD ", " VWD "* ou *"evt/x/WWD* [76,77,78].

3.6.2 RODIN

RODIN est une Plateforme open source basée sur Eclipse offrant un soutien efficace pour le raffinement et la preuve mathématique. La plate-forme RODIN est l'acronyme de *(Rigorous Open Development Environment for Complex Systems)[76]*. Il s'agit d'un outil de développement, permettant d'intégrer les outils supports à la méthode B. La Figure 2.11 montre l'interface de RODIN.

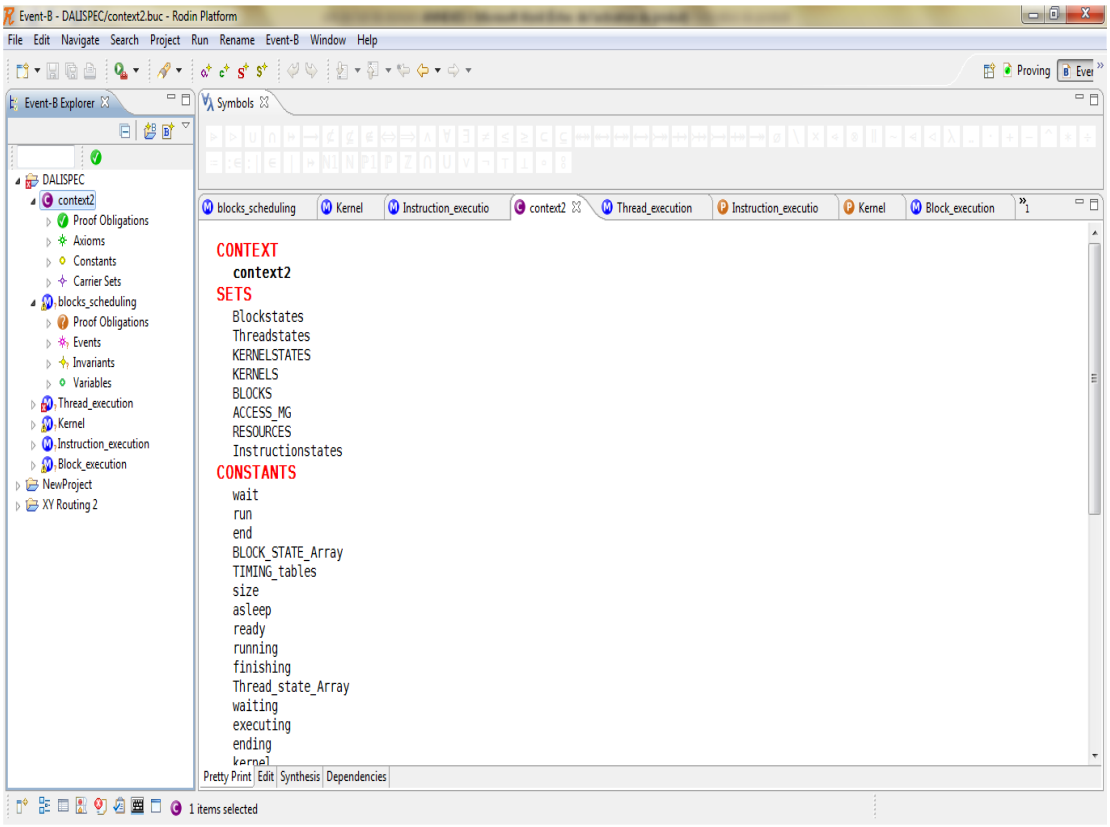


Figure 2.11. Interface de RODIN

La plate-forme RODIN a certaines caractéristiques qui en font d'elle un outil unique efficace, fiable et réutilisable pour le développement de modèles. Ainsi, elle aide à la compréhension du système dans son ensemble. Ces principales caractéristiques sont :

- ✓ La **vérification statique** confirmant si les propriétés du système sont en cours de validation. En cas de problème il faut signaler l'erreur puisque la plate-forme n'a pas de syntaxe fixe.
- ✓ Le **générateur** permettant de produire les preuves afin d'examiner si le modèle est valide ou non. Avec l'utilisation d'un démonstrateur permettant de vérifier automatiquement autant de preuves sans avoir besoin d'ajouter des prédicats sous forme des hypothèses.

- ✓ L’**interface graphique** facile à utiliser pour créer / modifier le modèle et raisonner sur le système par preuves interactives.
- ✓ La plateforme extensible permettant l'intégration de nouvelles fonctionnalités à l'outil RODIN tels que les vérificateurs modèle, les démonstrateurs, les animateurs, UML-B, latex, etc. à travers le développement de plug-ins [76,78,79].

Conclusion

Au cours de ce chapitre, nous avons présenté des méthodes de spécification. Nous avons choisi deux approches : semi-formelle et formelle. Concernant l’approche semi-formelle, nous avons choisi UML. Ensuite nous avons présenté le profil MARTE dédié à la modélisation des systèmes embraqués et temps réels. Cette modélisation est claire, facile à déployer et simplifiée à l‘utilisateur, toutefois moins sûre, et elle n’est pas formelle. Pour cela, nous sommes revenus aux spécifications formelles et plus particulièrement à la méthode Event B qui est retenue dans la suite de ce travail.

Chapitre 3 :

Etat de l’art de génération de code

Introduction

La spécification semi-formelle permet de présenter le système sous forme de diagrammes. En revanche, la spécification formelle d'Event B permet de présenter le système sous forme de contextes et de machines. Les contextes définissent le type de données du système. Nous proposons de faire un couplage entre la spécification formelle et semi-formelle des systèmes à base de GPU et générer un code valide.

Au cours de ce chapitre nous citons les approches de couplage de spécification formelle et semi-formelle et les approches de génération de code à partir une spécification.

La spécification de nombreuses applications et des applications parallèles en particulier nécessite des types spéciaux de langages parallèles tels qu'OpenCL et CUDA. Les types de ces langages n'existent pas dans le langage d'Event B. Plusieurs travaux ont étés proposés afin de définir de nouveaux types. Le plus efficace est le plug-in Theory. Cet outil permet de définir et de proposer des définitions de nouveaux types et de nouveaux opérateurs. Nous présentons le Theory plug-in et un état de l'art des travaux utilisant le plug in Theory.

1 Etat de l'art

1.1 Approches de Couplage de spécification UML/MARTE et Event B

Plusieurs travaux ont été réalisés dans le contexte de transformation de la spécification UML vers la méthode formelle B.

Laleau [80] proposait une approche basé sur les digrammes d'UML (classes, état-transition, collaboration) pour générer la spécification formelle B au contraire des travaux précédentes basés sur le diagramme de classes seulement. La méthodologie consiste à : 1) élaborer le digramme de classes qui représente la structure statique du système et, par la suite, la spécification B est générée. 2) Extraire les fonctions du système à partir des diagrammes d'état-transition décrivant le comportement des objets. 3) Collaborer entre les diagrammes pour déduire les annotations de la spécification formelle (machines B). 4) Réaliser les preuves globales du système en utilisant les preuves de B. Le prototype a été développé dans l'environnement de programmation Rose en utilisant le langage OCaml spécialisé dans le domaine de translation entre langages pour créer le translateur automatique. Une spécification B a pu être générée mais [80] a trouvé quelques limites telles que le manque de la sémantique et de la qualité de la spécification B généré dépendant de la façon dont les diagrammes d'UML sont présentés. En plus la spécification formelle doit être complété manuellement par le concepteur [80]. Un autre travail de formalisation du digramme de comportement en spécification B a été réalisé par LEDANG [81] qui a proposé une nouvelle approche d'utilisation de l'approche d'orientée objet pour le passage à la méthode formelle B. LEDANG note que l'incomplétude trouvée dans les autres travaux est dû à la mal distinction entre les concepts de machine abstraite B et de l'opération B. Une première phase qui appelée modélisation des opérations de classes a été proposée. Elle est constituée de (i) la dépendance de l'appelant-appelé qui existe entre une paire de classe (appelant-appelé) dans un diagramme d'activité ou de collaboration alors une opération B est créée reliant une classe d'opération à une classe de réalisation. (ii)Le groupement des informations et des opérations dans la même machine abstraite. (iii) La modélisation de la dépendance d'appelant-appelé dans la même classe d'opération ce qui demande de séparer l'opération appelant. Cette phase est réalisée à partir du diagramme de collaboration où [81] propose de le considérer comme des étages. Ce travail a utilisé les diagrammes de comportement en plus des digrammes de classes et

d'état-transition pour optimiser les approches de génération de spécification B à partir d'une conception UML[81] . Un autre travail [82] générant la spécification B à partir du digramme de cas d'utilisation a été réalisé. Il avait pour but de développer une spécification B à partir d'un modèle de cas d'utilisation d'un système à construire. Cette approche vise à fournir un cadre pour analyser formellement la cohérence du modèle des besoins produits dans un processus de développement par objets d'une part et d'intégrer la méthode B dès la phase de spécification des besoins dans un processus de développement de logiciel d'une autre part. L'approche se base sur le diagramme de cas d'utilisation en plus du diagramme de classe pour le passage à la spécification B. [82] a utilisé les schémas de dérivation objet-b de MEYER [83] relatifs aux modèles de classes pour traduire les classes du domaine d'application. Les cas d'utilisation ont été représentés textuellement et les notions des sous cas d'utilisation (use, include,…) ont été exploitées pour générer les machines[82]. Un outil de transformation systématique de digrammes UML en spécification appelé ArgoUML+B a été réalisé par [84] qui est une extension du prototype ArgoUML écrit en java permettant l'édition de la plupart des digrammes UML ainsi que des contraintes supplémentaires en OCL sur les diagrammes édités. Le prototype proposé est une extension d'ArgoUML implantant la procédure de dérivation d'une spécification B. ArgoUML+B est organisé comme suit :

- Premièrement, la création des diagrammes de classes et d'états transition par le biais de l'outil ArgoUML facilitant cette procédure. Par la suite ces diagrammes sont générés sous forme de codes source par une multitude de langages.
- La génération des données B à partir des diagrammes de classes en utilisant les éléments structurels au sein des diagrammes de classes et des diagrammes d'état-transition : classe, attribut, association et état.
- La génération des opérations B à partir des diagrammes état-transition. Une transition est dérivée formellement en B par une opération abstraite dont le rôle est de réaliser le changement d'état. Le nom de l'opération B modélisant une transition est préfixé par le nom de la classe en question ; cette opération est paramétrée par un argument modélisant l'objet cible de l'événement déclenchant la transition. L'opération est déclarée dans la machine B dérivée de la classe associée au diagramme d'état-transition en question. [84,85]

Les travaux étudiés utilisent les digrammes d'UML statiques et dynamiques vers la spécification formelle B. Nous proposons de générer une spécification B d'après une modélisation de système par UML en plus du profil MARTE qui spécifie la partie matérielle, logicielle et la notion de temps.

1.2 Approches de génération de code à partir d'une spécification semi-formelle

Dans le cadre de génération de code à partir d'une spécification MARTE, [32] a réalisé une multitude de travaux pour le passage d'une spécification MARTE à un code exécutable OpenCL sur GPU. L'objectif majeur de ce travail est de fournir des ressources pour les non spécialistes de la programmation parallèle pour développer leurs applications. L'ingénierie avancée et les communautés scientifiques ont utilisé la programmation parallèle pour résoudre leurs complexes problèmes. La programmation parallèle est difficile à implémenter en ce qui concerne les conditions du runtime, l'accès à la mémoire, etc. Pour faciliter la programmation parallèle, les développeurs ont spécifié des différentes approches de programmation ont été proposées. Les plus utilisé sont OpenMP[1] pour la mémoire partagé et le Message Interface (MPI) pour la programmation distribuée de la mémoire et OpenCL qui reste le premier standard pour la programmation parallèle à usage générale des systèmes hétérogènes. L'approche de [32] est basée sur MDE pour spécifier, modéliser et générer les applications OpenCL. [32] L'approche testée sur un simple algorithme du produit matriciel pour le paralléliser avec le GPU et génère un code exécutable OpenCL[86,87]. Ensuite [32] a passé au traitement d'algorithmes plus complexe tels que Sparse Matrix Solver. L'approche a aussi été examinée sur le parallélisme des solveurs distribués de matrices sur les processeurs graphiques. Les techniques de conjugaison de gradient sont implémentées sur des entrés sauvegardées en formats CSR, Symetric CSR et CSC. L'approche s'est montrée très efficace pour résoudre les fonctions basées sur les éléments finis de l'équation de Maxwell. L'algorithme itératif a été implémenté avec OpenCL et a été exécuté sur deux plateformes de GPU, la première a sur Nvidia Tesla C870 avec une précision simple et la deuxième sur Nvidia GeForce GTX 280 avec une double précision. Une comparaison de temps d'exécution sur CPU, sur C870, sur GTX280 a montré l'efficacité d'utilisation de GPU pour chaque itération. Cette approche a apporté

[1]**OpenMP** (Open Multi-Processing) est une interface de programmation pour le calcul parallèle sur architecture à mémoire partagée.

deux avantages dans la méthode de Maxwell : premièrement, le gain d'accélération de CG grâce à l'architecture parallèle de GPU, deuxièmement, la sauvegarde des matrices symétriques dans la mémoire[86]. Par la suite un aspect spécifique de compression de vidéo basé sur H.263 : Eclairage (scaling) a été traité sur GPU. L'éclairage consiste à transformer une vidéo représenté par CIF d'une grande taille à une petite taille (mise à l'échelle). Dans ce cas, le Downscaler est composé de deux composants : Un filtreur horizontal qui réduit le nombre de pixels de 352 lignes à 132. Un filtreur vertical qui réduit le nombre de pixels de 288 lignes à 128 avec une interpolation de 8 pixels (ligne/colonne). L'interpolation est répétée à chaque frame pour chaque pixel, couleur et canal. Ce qui montre la nécessité du parallélisme et de la répétition. Le modèle de Downscaler a été modélisé par MARTE et ARRAYOL [32,86,87]. Le Downscaler a été testé en quatre versions : la version séquentielle utilisant la même structure défini dans le modèle. Les deux autres versions d'OpenCL généré automatiquement. La quatrième version c'est un code OpenCL écrit manuellement. Des bons résultats ont été obtenus au niveau de performance, de temps d'exécution d'OpenCL et de temps de transferts de données. [88] a fait la comparaison de l'approche de génération de code OpenCL d'après MARTE avec une translation de code Single Assaignement C (SAC) en code CUDA pour voir l'efficacité des codes exécutables généré. L'application a été compilée et exécutée sur un frame d'image de 1080 × 1920. Le système de test contient un GPU Nvidia fermi GTX 480 [constitué de 15 SMs et chaque SM dispose de 32 Sps de 1.4 GHz] et un CPU Intel 2.8 GHz i7-930 avec 8MB de mémoire cache L2. Une comparaison entre les deux approches montre que les kernels CUDA transformé à partir de SAC prennent beaucoup plus de temps que les kernels OpenCL pour exécuter les filtres horizontale et vertical ce qui montre l'efficacité de l'approche basée sur Gaspard2. En ce qui concerne les transferts de données (host2device et device2host) sont identiques approximativement. Les deux approches nous conduisent à une performance significative d'accélération jusqu'à 11× sur GPU par rapport aux plateformes séquentiels [86,89,90,91].

Les travaux de WENDELL traités [86,87,89,90,91] visent de faire la transformation de conception MARTE avec ARRAYOL au langage OpenCL. Ils arrivent à générer un code OpenCL valide qui améliore l'exécution sur GPU. Mais il reste toujours des limites en ce qui concerne l'optimisation du code OpenCL et le temps d'exécution. On a vu que les transferts de données (host/device) prends 50% du temps

d’exécution alors ce temps doit être minimisé. Une autre perspective est de générer un code CUDA à partir d’une conception MARTE.

1.3 Approches de génération de code à partir une spécification formelle Event B

Event-B est devenu très utile pour modéliser et spécifier des systèmes dans de nombreux systèmes embarqués. Il existe de nombreux travaux sur la génération de code valide à partir d'Event-B. Nous décrivons certains d'entre eux. Cependant, contrairement aux travaux des auteurs, aucun de ces travaux n'avait un langage de programmation parallèle OpenCL ou Cuda comme langage cible.

E2ALL [92] est un outil de génération automatique de code. Il est dédié à la traduction des spécifications Event-B en plusieurs langages de programmation (C, C++, Java, C#). Cet outil est une collection de plugins RODIN (E2C, E2C++, E2Java, E2C#). L'E2ALL est développé en utilisant les propriétés de sécurité (Safety properties) et prenant en charge l'étape de mise en œuvre du code. Les spécifications formelles sont traduites en codes de source dans un langage de programmation donné. Le code source est compilé pour réécrire les notations partiellement formelles d'Event-B [92]. La traduction se base sur les modèles de contexte définissant les nouveaux types de données en plus des types existants d'Event-B. De plus, [92] a également défini les fonctions parallèles des langues cibles dans le même contexte. Les limites de l'outil E2ALL sont que toutes les notations ne sont pas directement traçables dans un langage de programmation et qu'il nécessite l'interaction de l'utilisateur pour corriger le pré-code généré final.

Un autre travail de génération de code pour Event-B a été proposé par [93], cette approche est basée sur le principe de correcte par construction (correct by construction) et repose sur un raisonnement de restrictions de bonne définition (well-definedness), les assertions et le raffinement. L'exactitude est assurée par l'utilisation combinée de restrictions de définition, de raffinement et d'assertions.

Dans un autre travail, [94] a proposé de générer du code Java à partir d'Event-B. L'outil est nommé EventB2Java. La principale contribution de cette approche est la définition d'une traduction avec indicateur complet d'Event-B vers JML (Java Modeling Language): programme Java annoté et l'implémentation de ces traductions en tant qu'outil EventB2Java. L'outil permet de générer un pré-code Java à partir de la spécification Event-B. Ce pré-code est exécuté à l'aide d'une classe Java spécifique. Le

code est généré après une séquence de raffinements pour écrire un pré-code Java. Ensuite, ils ont créé le plug-in RODIN EventB2Java pour générer automatiquement du code Java. Dans une dernière étape, le développeur peut corriger ou injecter un code externe dans celui généré.

En appliquant l'approche de bonne définition du modèle (well-definedeness), les erreurs d'exécution qui proviennent de différences sémantiques entre la langue cible et Event-B sont évitées. Bien que [95] n'aient présenté que la traduction en code source C et aient déclaré que leur approche est également applicable à d'autres langages en adaptant l'étape de bonne définition et de restriction au langage cible correspondant.

[96] se concentre sur la difficulté de mappage un modèle formel à un langage de programmation spécifique. Afin de proposer une solution, il présente une approche de mappage source-à-source entre les modèles Event-B et les programmes Eiffel, permettant ainsi la preuve de l'exactitude de certaines propriétés système via l'approche Design-by-Contract (nativement supporté par Eiffel), tout en utilisant toutes les fonctionnalités de la programmation Orienté Objet. [96] présente une série de règles pour transformer un modèle Event-B en programme Eiffel. La traduction tire pleinement parti de tous les éléments de la source en les traduisant sous forme de contrats dans la langue cible. Ainsi, aucune information sur le comportement du système n'est perdue. Ces règles montrent une méthodologie pour les logiciels construction qui utilise deux approches différentes.

L'objectif de [97] est de développer une transformation automatique des algorithmes distribués d'Event-B en programmes DistAlgo. Le langage Event-B combine des techniques de raffinement et de modélisation par état et est adapté à la vérification des systèmes distribués. Le langage DistAlgo est un langage de programmation de haut niveau pour les algorithmes distribués. Son haut niveau rend DistAlgo plus proche des notations mathématiques de l'événement B et améliore la clarté des programmes DistAlgo. Une transformation automatique vérifiée garantit que les propriétés prouvées dans le modèle tiennent toujours dans le programme et facilite le processus de développement.

[98] a proposé un système de de traduction d'Event-B en code Python. Le système proposé améliore les spécifications des exigences, ce qui aboutirait à une conception uniforme du système. Cette approche est très utile pour l'automatisation industrielle.

Event-B a le RODIN pour vérifier le modèle et générer des obligations de preuve. Ce travail propose de faire une traduction de spécifications Event-B en code Python à travers des règles de traduction mappant chaque composant d'un modèle Event-B en classe Python.

2 Plug-in Theory Event B

2.1 Présentation du Plug-in Theory

Event-B est un langage formel pour la modélisation des systèmes, basé sur la théorie des ensembles et la logique des prédicats[58,68]. Il permet de spécifier les systèmes, les raffiner et les prouver. La preuve est réalisée à l'aide de l'outil RODIN [99,100]. (Voir Section 3.4 chap2)

Dans les versions récentes de la plateforme RODIN, il n'y avait pas d'outils disponibles pour définir de nouveaux opérateurs ou pour étendre les prouveurs standards avec de nouvelles règles de preuve. Pour surmonter cette limitation, un travail important a été développé pour modifier la plateforme RODIN: **le composant Theory** est disponible dans RODIN en tant que **plug in** . La Figure 3.1 montre le plug-in Theory dans RODIN.

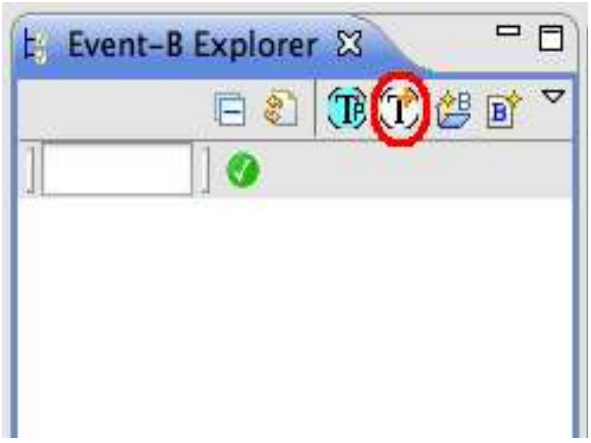


Figure 3.1. Notation du composant Theory dans RODIN

2.1.1 Composant Theory

Une composante théorie est un nouveau type de composant Event-B permettant de définir des théories qui peuvent être indépendantes de tout modèle particulier. Une théorie est le moyen par lequel le langage mathématique et les prouveurs automatiques peuvent être étendus[101]. Comme illustré dans la Figure 3.2, un composant de théorie

a un nom, une liste de paramètres de type globaux (globaux à la théorie), et un nombre arbitraire de définitions et de règles .

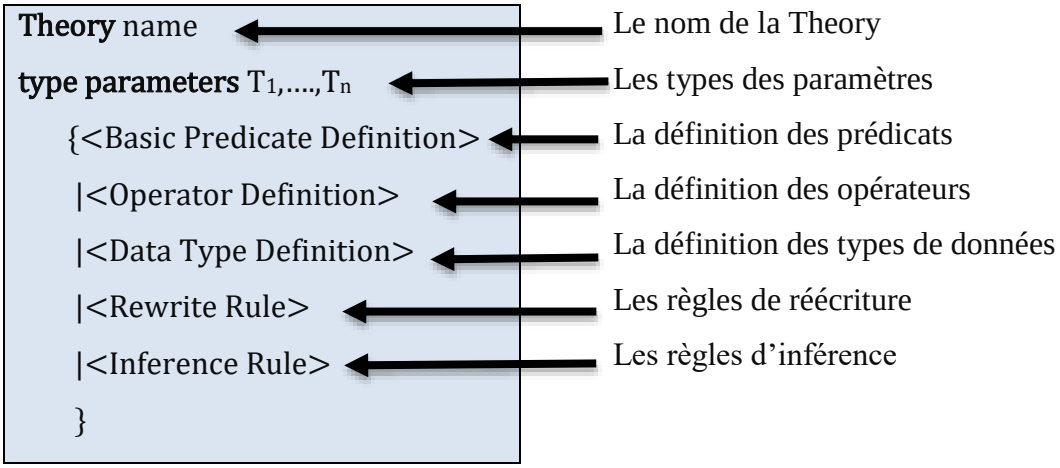


Figure 3.2. Composant Theory

Une théorie Event-B est identifiée par un nom dans l'espace de travail de RODIN. La hiérarchie des théories peut être créée au moyen de la technique d'importation.

"Theory A imports Theory B" ; indique que toutes les définitions et les règles de la théorie B peuvent être utilisées dans la théorie A. Une théorie peut avoir un nombre arbitraire de paramètres de type qui sont des ensembles supposés être non-vide. Dans ce cas, la théorie est dite polymorphe sur ses paramètres de type [101].

2.1.2 Relation du composant Theory avec les notations d'Event B

Pour rappel, une Machine est composé d'une partie statique contenant les états, les invariants et les propriétés, et une partie dynamique contenant des transitions. [102,103, 104] .

La Figure 3.3 montre l'utilisation de la notation de Theory permettant de définir de nouveaux opérateurs et de nouveaux types pour pouvoir spécifier le système car Event B contient un nombre limité de types de données.

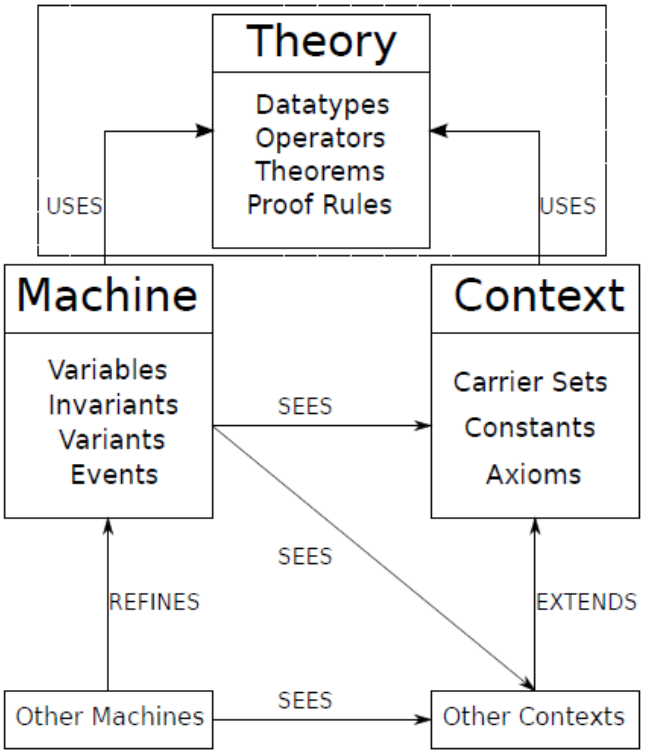


Figure 3.3. Anatomie élargi des modèles Event B

2.1.3 Le déploiement d'une Theory

L'utilisation des théories dans Event B mène à définir de nouveaux formalismes pour enrichir une spécification formelle. Il faut bien différencier entre le développement d'une **T*heory*** et le déploiement d'une **T*heory.***

Le ***développement*** de la ***Theory*** fait référence à l'activité de définition et de validation de ***Theory***. A ce stade, les extensions sont définies et les obligations de preuve sont automatiquement générées pour chaque extension si nécessaire. Cette activité peut suivre un modèle itératif puisque l'inspection de tentatives de preuves automatiques en échec peut révéler des informations importantes sur la solidité ou non des extensions. L'exécution de preuves interactives fournit la rétroaction et guide le modélisateur pour modifier les définitions, le cas échéant. Par conséquent, le développement de ***Theory*** profite grandement de la nature réactive de la plateforme RODIN [101,105].

Le ***déploiement de la théorie*** se réfère à l'activité de rendre les théories développées disponibles pour leur utilisation en modélisation. Une théorie est réutilisable par de nombreux modèles en même temps. Le déploiement de la théorie garantit que les obligations de preuve sont au moins inspectées par l'utilisateur, et une fois déployées,

toutes les extensions mathématiques et les règles de preuve peuvent être utilisées pour spécifier les contextes et les machines Event-B [101,107].

2.2 Travaux utilisant le plugin Theory

Le plugin Theory a permis d'améliorer plusieurs travaux de la spécification formelle en utilisant Event B. Dans ce qui suit nous présentons quelques exemples qui ont aboutis à des résultats importants.

2.2.1 Opérateur Booléen

Dans [101,108], les auteurs proposent un nouvel opérateur pour exprimer les relations booléennes. Les expressions et les prédicats sont des catégories différentes syntaxiquement dans le langage mathématique Event-B. Contrairement aux expressions, les prédicats n'ont pas de type. Cependant, Event-B fournit un Type BOOL a deux valeurs : TRUE et FALSE.

$$BOOL=\{TRUE,FALSE\}$$

BOOL, TRUE et False sont des expressions. La Theory proposée nommée *BooleanOps* a pour but d'exprimer les relations logiques $(\neg, \wedge, \vee)$ sur des variables booléennes.

Les définitions d'opérateurs sont toutes directes à l'aide de l'opérateur bool. Les opérateurs AND et OR sont des opérations commutatives. La Figure 3.4 montre la syntaxe de BooleanOps Theory [101,108].

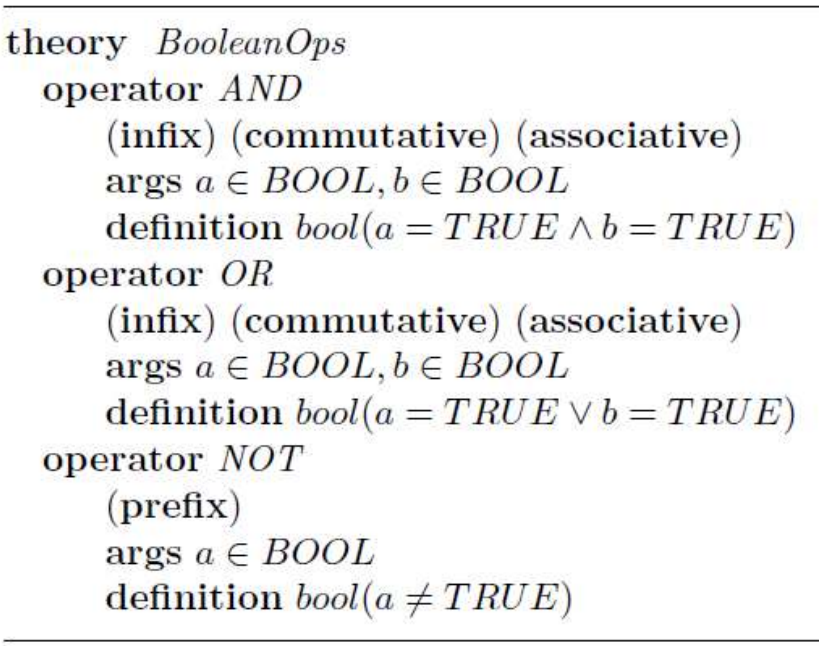


Figure 3.4. BooleanOps Theory

2.2.2 Listes inductives

[101] propose une nouvelle Theory définissant le type de donnée Liste dans le langage Event B. Des opérations pouvant être appliquées sur une liste telles que le calcul de sa taille, l'ajout d'éléments, la concaténation de deux listes, etc.

2.2.2.1 La Theory Lists : Elle permet de définir le type de liste ou chaque liste est définie par deux paramètres clefs : la tête et la queue. La Theory Lists est illustré dans la Figure 3.5.

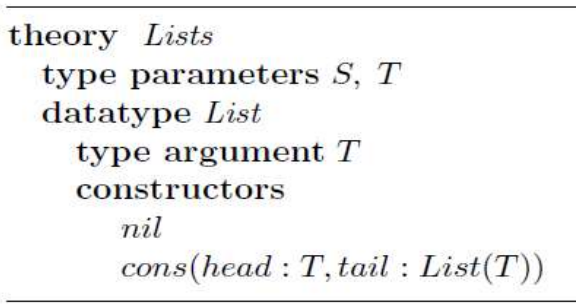


Figure 3.5. Lists Theory

2.2.2.2 Les opérateurs sur le type liste

Opérateur listsize : permet de calculer le nombre d'éléments dans une liste, Comme illustré dans la Figure 3.6.

operator *listSize*
(prefix)
args $l \in List(T)$
definition
case l
$listSize(nil) = 0$
$listSize(cons(x_0, l_0)) = 1 + listSize(l_0)$

Figure 3.6. Opérateur listSize

- **Opérateur append :** permet l'ajout d'un nouvel élément à une liste. La Figure 3.7 montre l'opérateur append.

operator *append*
(prefix)
args $l \in List(T)$, $e \in T$
definition
case l
$append(nil, e) = cons(e, nil)$
$append(cons(x_0, l_0), e) = cons(x_0, append(l_0, e))$

Figure 3.7. Opérateur append

- **Opérateur conc :** permet de concaténer deux listes pour produire une liste unique, comme illustré dans la Figure 3.8.

operator *conc*
(infix) (associative)
args $l_1 \in List(T),\ l_2 \in List(T)$
definition
case l_1
$nil\ conc\ l_2 = l_2$
$cons(x_0, l_0)\ conc\ l_2 = cons(x_0, conc(l_0, l_2))$

Figure 3.8. Opérateur conc

2.2.3 Array Theory

[109] propose une nouvelle Theory array pour générer du pré-code. Une Theory Array présente la structure des tableaux. [109] a d'abord crée une nouvelle Theory des tableaux de Type T. le tableau est un opérateur prenant comme argument un power set de type T. La Figure 3.9 montre la Theory Array.

theory *Array*
type parameters T
operator *array*
(prefix)
args $s \in \mathbb{P}(T)$
definition $\{n, f \cdot n \in \mathbb{N} \land f \in 0..(n-1) \rightarrow s \mid f\}$

Figure 3.9. Theory array

Des opérations nécessaires de traitement de tableaux sont proposées telles que la création d'un tableau, le calcul de la taille d'un tableau, la mise à jour et la recherche d'un élément. La Figure 3.10 présente les opérateurs sur un tableau.

operator *arrayN*
(prefix)
args $n \in \mathbb{Z}, s \in \mathbb{P}(T)$
condition $n \in \mathbb{N} \land finite(s)$
definition $\{a \mid a \in array(s) \land card(s) = n\}$
operator *lookup*
(prefix)
args $a \in \mathbb{Z} \leftrightarrow T, i \in \mathbb{Z}$
condition $a \in array(T) \land i \in 0..(card(a) - 1)$
definition $a(i)$
operator *update*
(prefix)
args $a \in \mathbb{Z} \leftrightarrow T, i \in \mathbb{Z}, x \in T$
condition $a \in array(T) \land i \in 0..(card(a) - 1)$
definition $a \Leftarrow \{i \mapsto x\}$
operator *newArray*
(prefix)
args $n \in \mathbb{Z}, x \in T$
condition $n \in \mathbb{N}$
definition $(0..(n-1)) \times \{x\}$

Figure 3.10. Opérateur des opérations sur les tableaux

2.2.4 NoC Theory

Le NoC Theory [110] est une Theory utilisée pour définir les relations sur une architecture Network On Chip. Le NoC est présenté sous forme d'un réseau entre les sources et les destinations des paquets. Ce graphe est non vide, non transitif et symétrique.

L'architecture NoC est généralement une topologie maillée : les commutateurs de périmètre sont connectés à deux, trois ou quatre voisins. L'algorithme de routage XY est définit par les coordonnées 2D: (xs, ys) pour une source (s) et (xd, yd) pour une destination (d) d'un paquet P.

Le réseau NoC est un ensemble de nœuds pouvant avoir plusieurs rôles, par exemple être une source src de transition de paquet ou un receveur de transition ou un dst intermédiaire pendant la transition de phase. Un nœud peut ne pas avoir de rôle nop. les data types : nop, src, rep et dst sont définis.

Plusieurs opérateurs sont définis pour assurer la transmission entre les nœuds voisins [110]. Les opérateurs sont présentés dans le tableau 5.

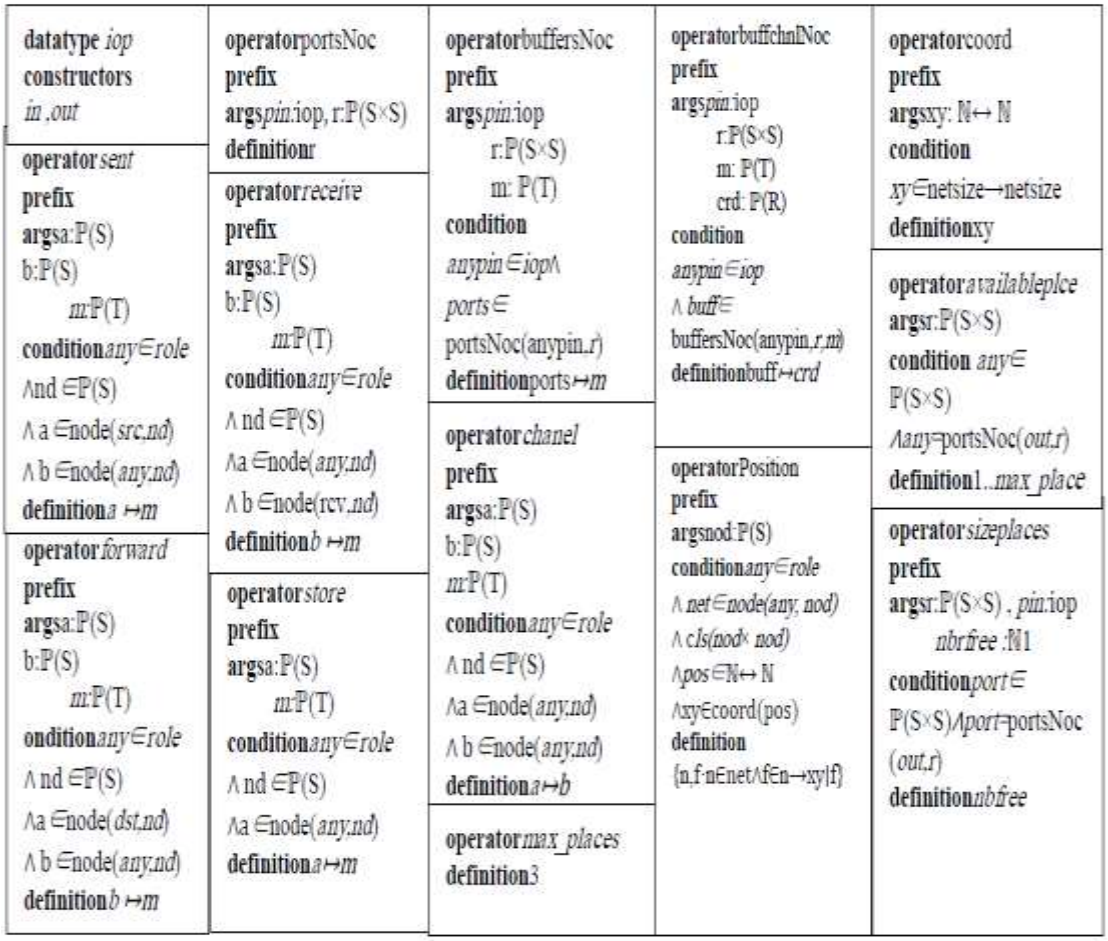


Tableau 5. Opérateurs NoC

Conclusion

Le chapitre présente les travaux connexes de couplage de spécifications semi-formelle et formelle et les approches de passage d'une spécification à l'autre. Nous avons aussi cité les travaux de génération de code à partir une spécification formelle Event B. Une Theory d'Event B est utilisée pour définir des extensions mathématiques : types de données, définitions directes, récursives et axiomatiques des opérateurs, des preuves des théorèmes polymorphes, des règles de réécriture et d'inférence. Les théories constituent une base utile pour la génération de vérification statique et d'obligations de preuve garantissant qu'aucun compromis ne sera fait pour l'infrastructure existante de modélisation, de preuve et de toute extension apportée. En substance, le module théorique fournit une plate-forme systématique pour définir et valider la bonne création de systèmes embarqués. Nous avons présenté la structure de Theory ainsi que quelques exemples de son utilisation pour la définition de nouveaux types et de nouveaux opérateurs. Le Chapitre 4 explique l'utilisation du plug in Theory et d'autres outils formelle et semi-formelle pour spécifier un système et générer un pré-code parallèle.

Chapitre 4 :

Approche proposée de spécification multiforme et de génération de code appliquée sur GPU

Introduction

La problématique de ce travail se situe dans le cadre de couplage de spécifications semi-formelle et formelle générant le code exécutable d'une application parallèle qui peut s'exécuter sur des architectures GPU. Comme noté dans l'introduction générale, le but est de spécifier un système sur puce d'une part avec une méthode semi-formelle pour qu'elle soit accessible et compréhensible en utilisant l'UML et les différents profils de MARTE permettant de traiter les propriétés des systèmes embarqués, et d'autre part, de valider la spécification MARTE et s'assurer qu'elle est sûre par le couplage de cette dernière avec une spécification formelle Event B. Dans ce chapitre nous présentons l'approche générale proposée, ainsi que les résultats obtenus.

1 Approche de spécification proposée

L'objectif est de partir d'une spécification formelle en Event B des tâches sur GPU et d'en valider leur ordonnancement à l'aide de RODIN. A cela ajouter un modèle temporelle permettant de montrer les traces d'exécution et l'évolution du temps pendant l'exécution des tâches sur GPU.

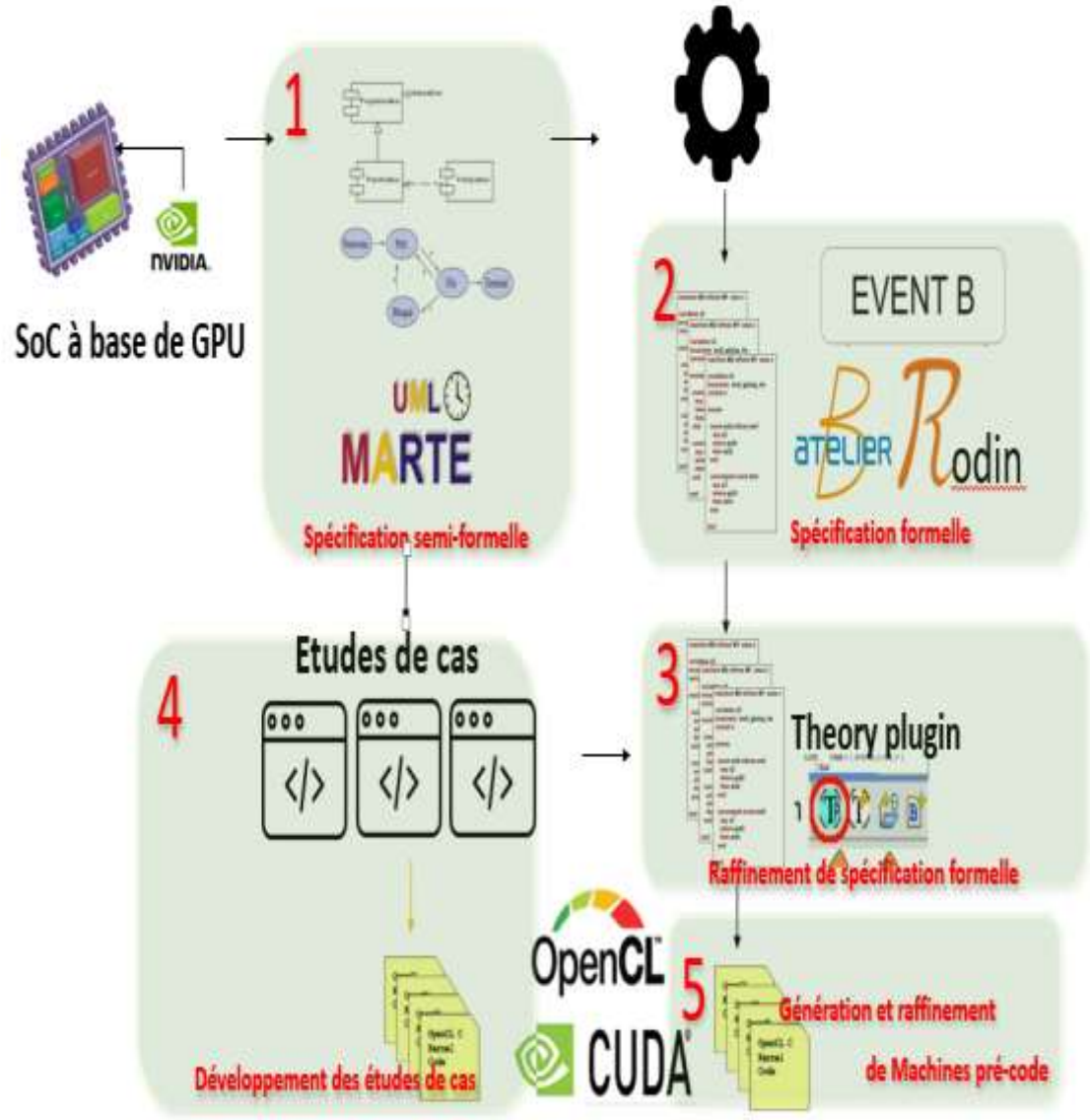


Figure 4.1. Approche proposée de spécification d'un système embarqué

Comme illustré dans la Figure 4.1, après la spécification d'un système, la tâche consiste à transformer la spécification en code exécutable et parallèle sur les unités de calcul du SOC d'une façon appropriée. Le code qui va être généré doit être s'exécuter sur des architectures GPUs (en CUDA, en OpenCL).

2 Architecture choisi (GPU GF210 Nvidia)

2.1 Structure de l'architecture GPU GF 210

L'architecture GPU GF 210 ,illustré dans la Figure 4.2, est une architecture à base de multiprocesseurs utilisée pour renforcer le calcul intensif et pour décharger le CPU. Les composants d'architectures GPU NVIDIA ont été expliqués dans le premier chapitre (Voir Section 2.4).

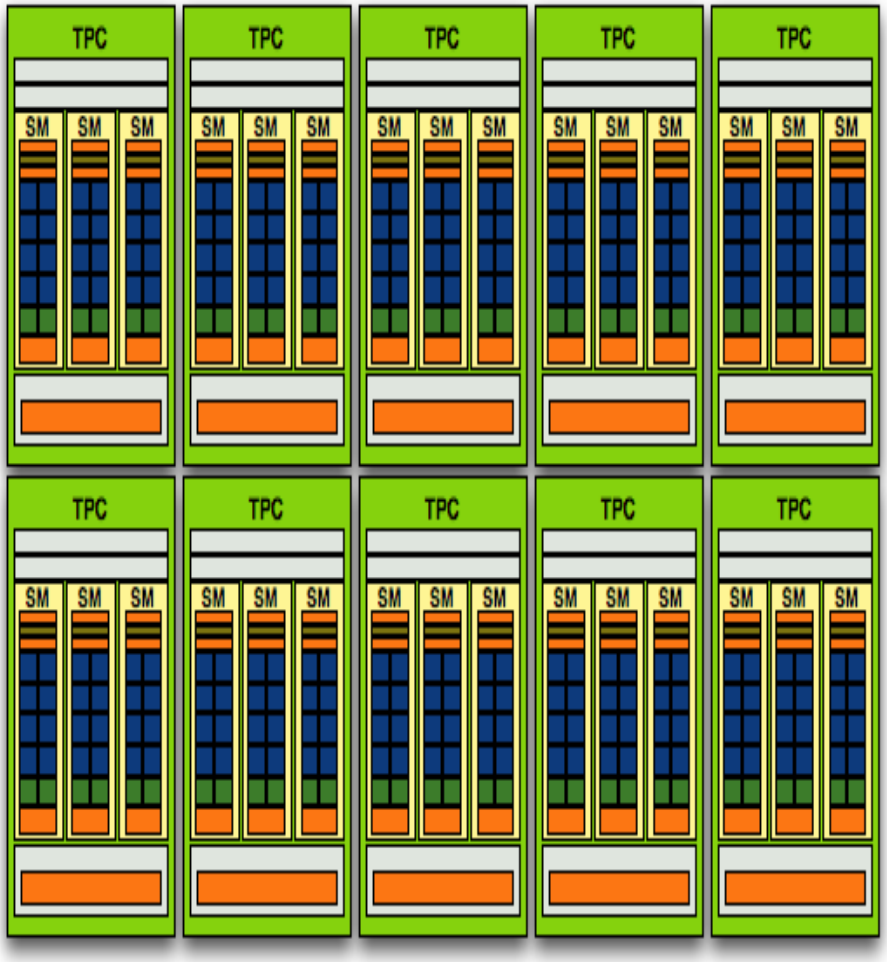


Figure 4.2. Architecture de GPU GF210

2.2 Ordonnancement sur GPU

Lorsqu'un kernel est lancé sur GPU il est transformé en grille. Chaque grille est composée d'un ensemble de blocs, composé de warp correspondant à un regroupement de threads. La Figure 4.3 montre la correspondance entre les composants matériels et logiciels sur GPU.

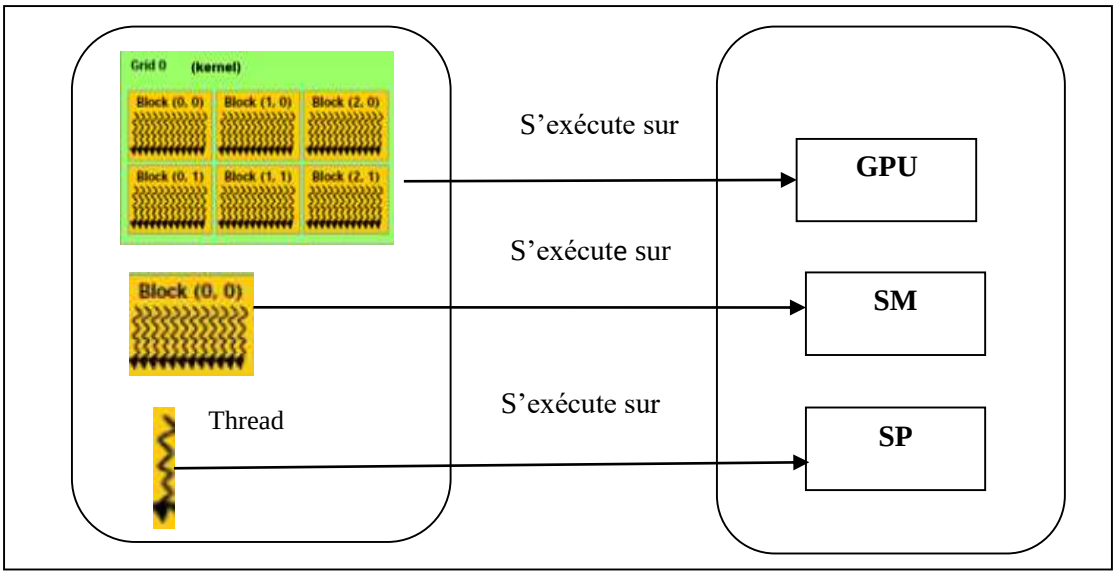


Figure 4.3. Correspondance matériel-logiciel sur GPU

La politique d'ordonnancement d'un kernel sur GPU est présentée dans la Figure 4.4.

- Lorsqu'un kernel est lancé, il sera affecté à un GPU sous forme d'une grille. Cette grille est composée d'un ensemble de blocs de threads. Chaque bloc de threads est affecté à un Streaming Multiprocessor (SM) du GPU. Si le nombre des SMs est insuffisant pour exécuter tous les blocs en parallèle alors les blocs seront ajoutés à une liste d'attente FIFO. Si un SM se libère, le premier élément dans la liste d'attente lui sera affecté.
- Au niveau d'un bloc, il y a entre 1 et 1024 threads lancés potentiellement en parallèle. Les threads sont exécutés par groupe de 32 correspondant à la taille d'un warp (granularité SIMT). La concurrence d'exécution entre les threads d'un bloc impacte la cohérence mémoire (mémoire partagée et globale).
- Les threads d'un bloc communiquent entre eux via la mémoire et l'utilisation de barrière de synchronisation.
- Les threads de blocs différents ne peuvent se synchroniser entre eux.
- Lorsque l'exécution de la grille est finie, le résultat d'exécution est transféré au GPU.

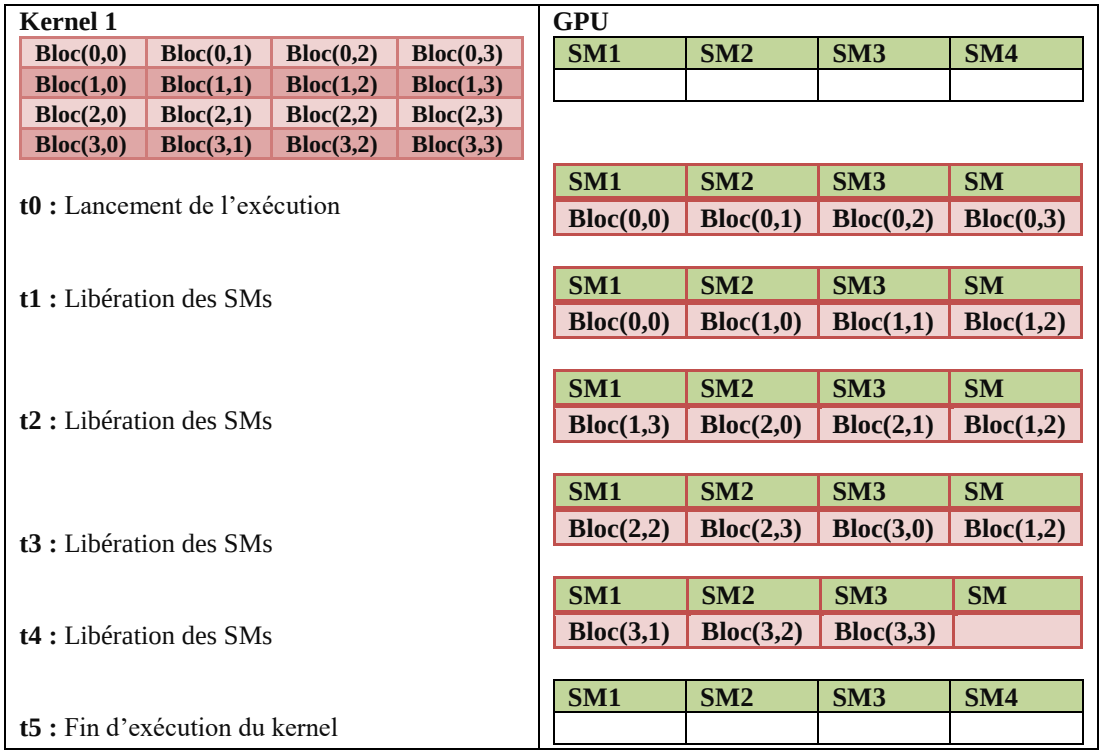


Figure 4.4. Exécution d'un kernel sur un GPU contenant quatre SMs

3 Spécification UML et MARTE du GPU

Afin de spécifier le GPU**,** nous utilisons les diagrammes de composants, de classes et de séquences en plus de quelques profils de MARTE :

- ✓ HRM profile {composants matériels}
- ✓ SRM profile {architecture de l'application}
- ✓ Allocation profile pour faire la relation entre les architectures matérielles et logicielles.

Pour réaliser la spécification, Nous utilisons l'outil Gaspard2 et l'outil Modelio.

a) **Les composants élémentaires utilisés pour la spécification du GPU :** les composants de calcul, de stockage, d'interconnexion et de synchronisation utilisés sont montrés dans la Figure 4.5.

Figure 4.5. Composants matériels élémentaires de GPU

b) Les composants élémentaires logiciels : sont illustrés dans la Figure 4.6.

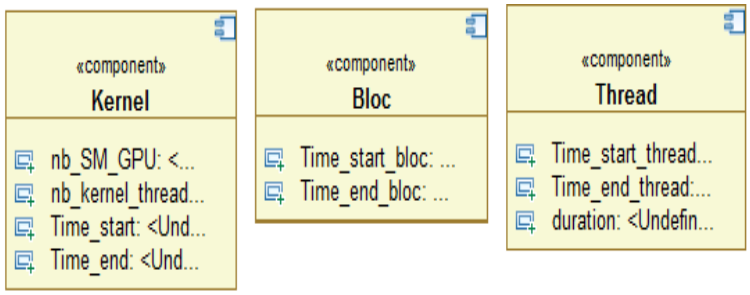


Figure 4.6. Composants logiciels élémentaires de GPU

c) Le diagramme de composant de GPU : Comme illustré dans La figure 4.7, il montre les composants matériels du GPU et les relations qui existent entre eux. Comme il est cité dans le chapitre 1 (Voir Section 2.7), le GPU est composé d'un ensemble de streaming multiprocessor et une mémoire globale. Chaque streaming multiprocessor est composé d'un ensemble de cœurs (streaming processor) lié à une mémoire partagée par les cœurs du même streaming

multiprocessor. Chaque cœur contient une mémoire locale qui lui permet de faire les traitements et les calculs.

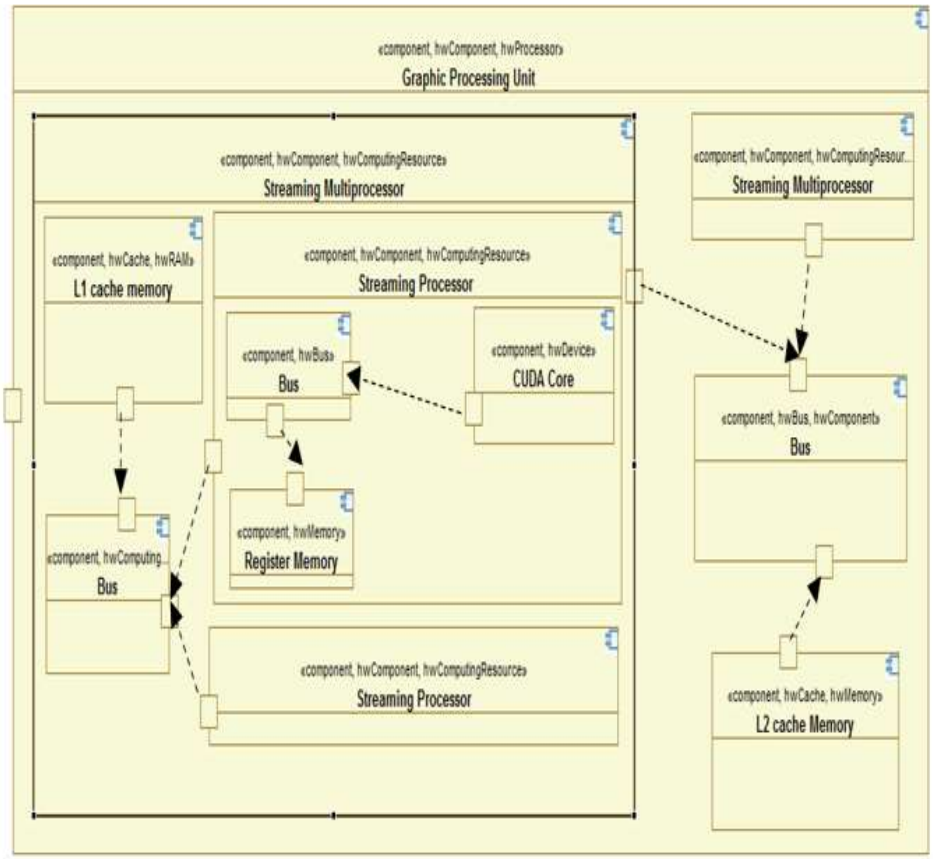


Figure 4.7. Diagramme de composant du GPU

d) La représentation de la classe GPU : est illustré dans la Figure 4.8.

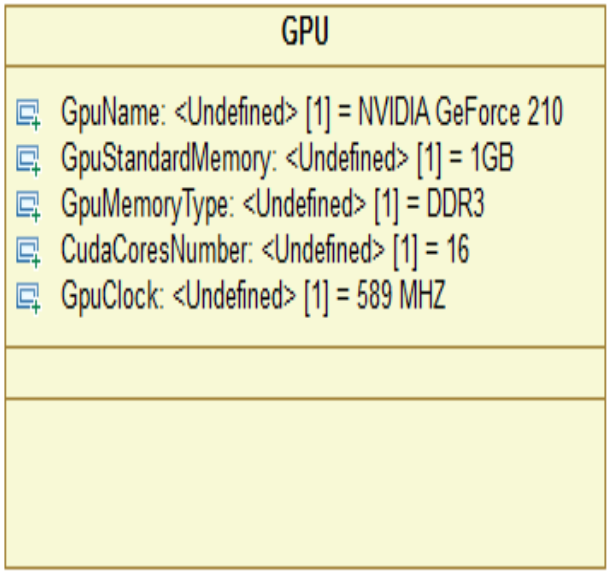


Figure 4.8. Classe GPU

e) **Le diagramme de séquences d'exécution du kernel :** les étapes de communication entre CPU et GPU sont illustrées dans le diagramme de la Figure 4.9.

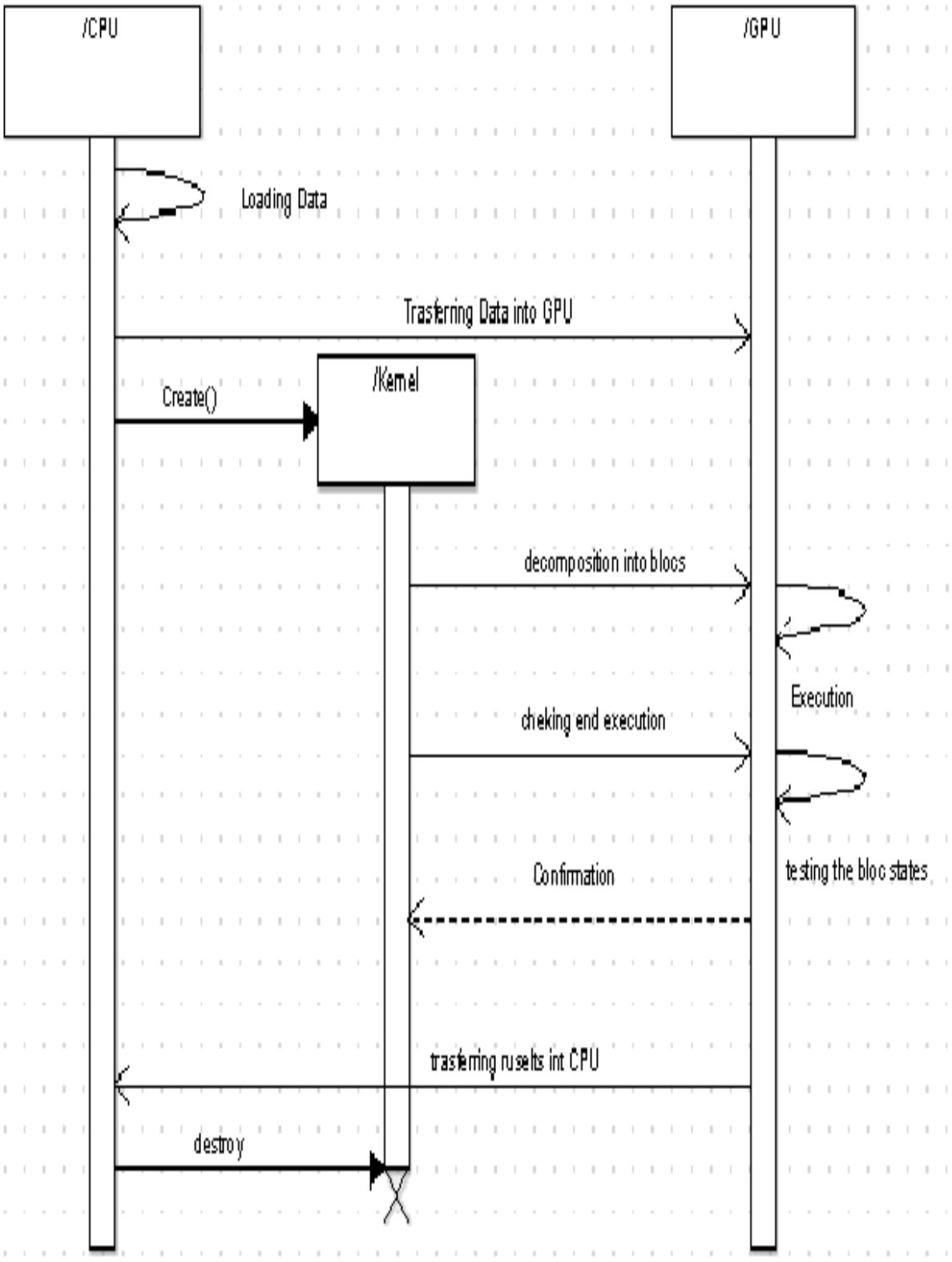


Figure 4.9. Diagramme de séquence

f) **Modélisation d'une application :** pour illustrer la modélisation d'une application sur GPU, Nous avons choisi l'addition de deux vecteurs (C=A+B, C, A, B : tableaux de N éléments). Un exemple est montré dans la Figure 4.10. L'exécution de l'addition sur GPU optimise le temps d'exécution.

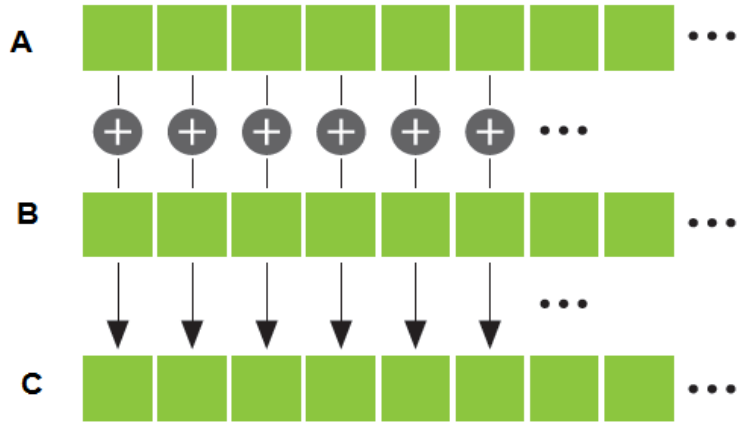


Figure 4.10. Addition vectoriel

g) **Allocation de l'application d'addition vectorielle sur l'architecture matérielle :** l'allocation des ressources logicielles sur les ressources matérielles est illustrée dans la Figure 4.11.

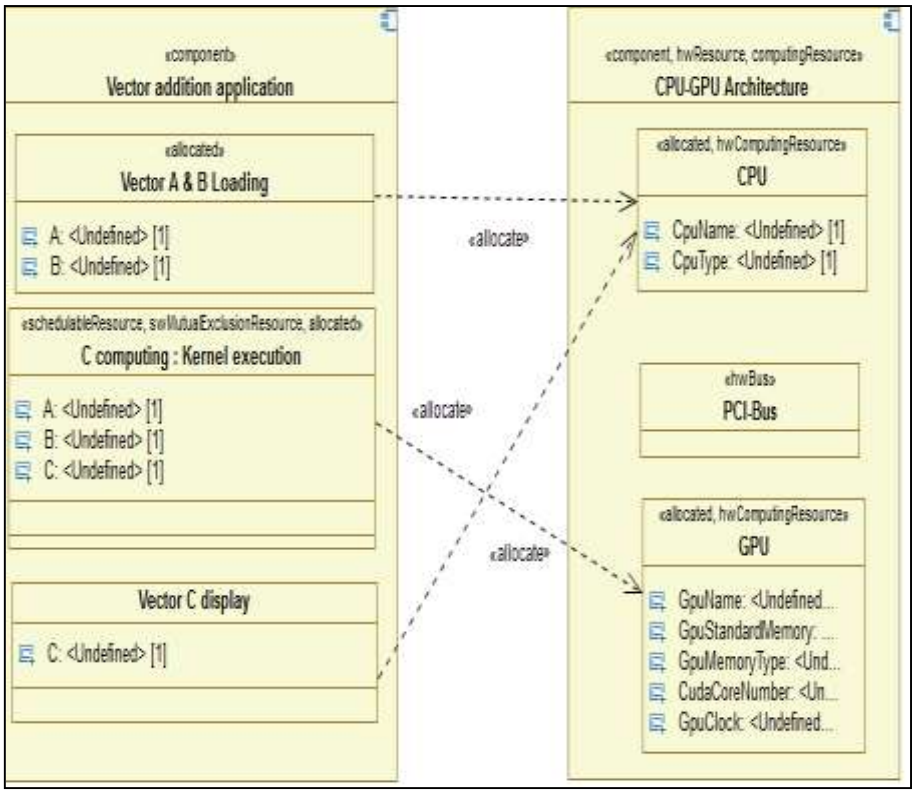


Figure 4.11. Allocation de l'application d'addition vectorielle sur l'architecture matérielle

h) **Timing diagram :** utilisé pour modéliser le temps, lorsque l'exécution d'un kernel se lance, les données seront transférées de la mémoire CPU vers la mémoire GPU. Ensuite le traitement parallèle s'exécute sur les différents cœurs disponibles. Puis les résultats sont transférés de la mémoire GPU à la mémoire CPU. La Figure 4.12 montre le Timing diagramme de l'exécution d'un kernel.

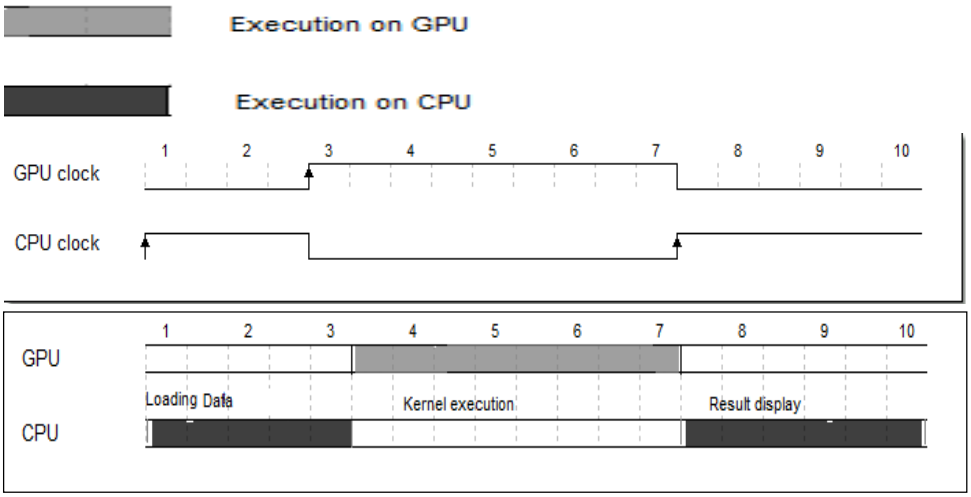


Figure 4.12. Timing diagramme de l'exécution d'un kernel

4 Couplage de spécification UML/MARTE et Event B

Comme illustré dans la Figure 4.13, le couplage des modèles MARTE avec Event B est une technique consistant à représenter les connaissances d'un domaine d'application à l'aide de diagrammes UML et à traduire ces diagrammes en langage B, puis de mettre en œuvre des raisonnements sur ces représentations. Ce couplage a pour avantage d'utiliser UML comme point de départ de la modélisation B des modèles orienté objet, valider les modèles UML à l'aide d'outils de preuve (atelier B, BToolKit) de la méthode B Event et de donner une sémantique aux modèles UML/MARTE à l'aide de Event B.

Les règles proposées au cours de ce travail consistent en la transformation des deux diagrammes (composant et état-transition) pour passer à Event B.

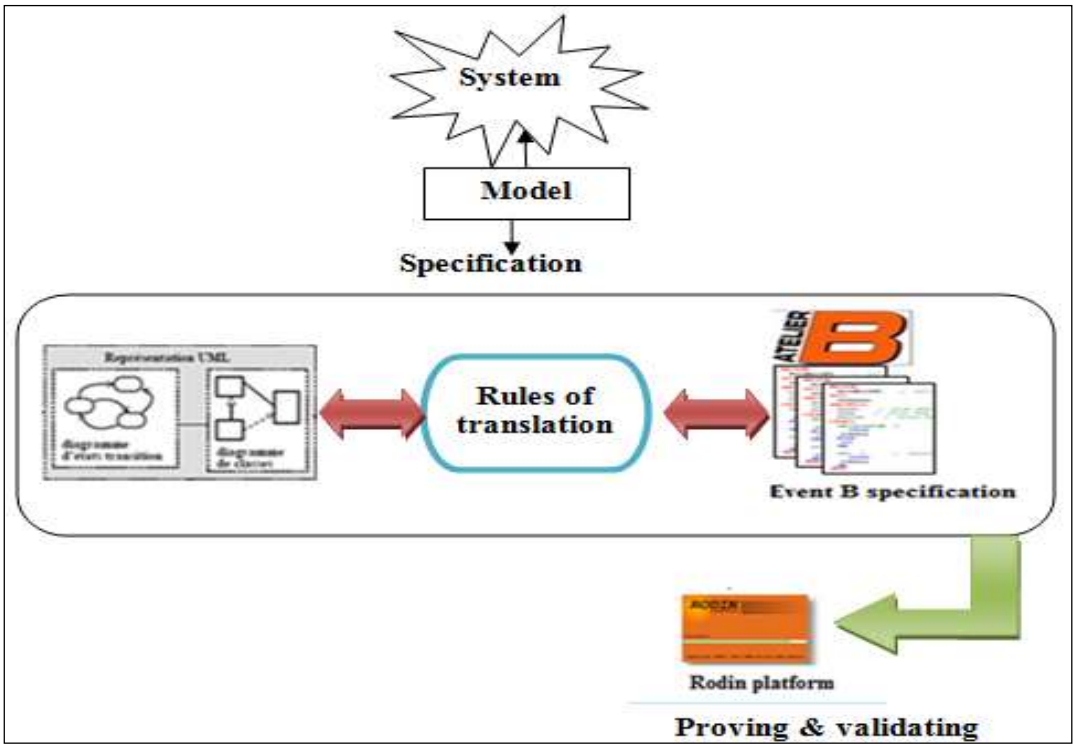


Figure 4.13. Architecture du processus de transformation

La modélisation de GPU a été spécifiée avec Gaspard 2 en premier lieu qui contient les notations d'UML/MARTE permettant la réalisation des différents diagrammes. Ensuite nous avanos essayé de générer la spécification formelle d'Event B.

La génération de la spécification du digramme d'état-transition est illustré dans la Figure 4.14.

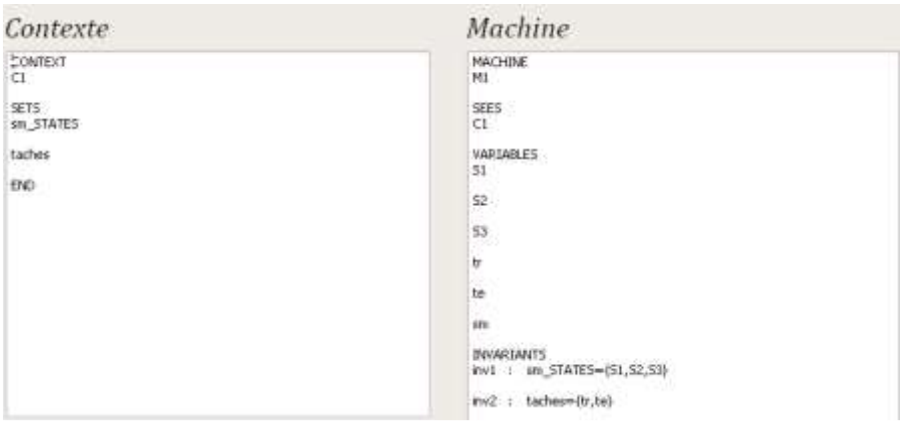


Figure 4.14. Le code B-event du diagramme d'état-transition

Le code Event-B crée a été prouvé par les preuves formelles de la plateforme RODIN. Nous avons trouvé quelques limites de l'approche de transformation d'UML/MARTE vers B tels que :

- Après la transformation, une grande partie du travail consiste à compléter et à adapter la spécification B obtenue.
- Le manque de liens dynamiques entre la représentation UML et B conduit à ce que les modifications opérées sur la spécification B générée ne soient pas prises en compte au niveau UML, ce qui pose le problème de la cohérence entre les deux représentations.

5 Spécification Event B détaillée d'exécution des tâches sur GPU

Nous proposons une spécification Event B du modèle de programmation des GPUs Nvidia en prenant en considération le temps de calcul et d'exécution pour valider l'ordonnancement sur GPU. Les éléments de base de la spécification formelle en Event B sont illustrés dans la Figure 4.15.

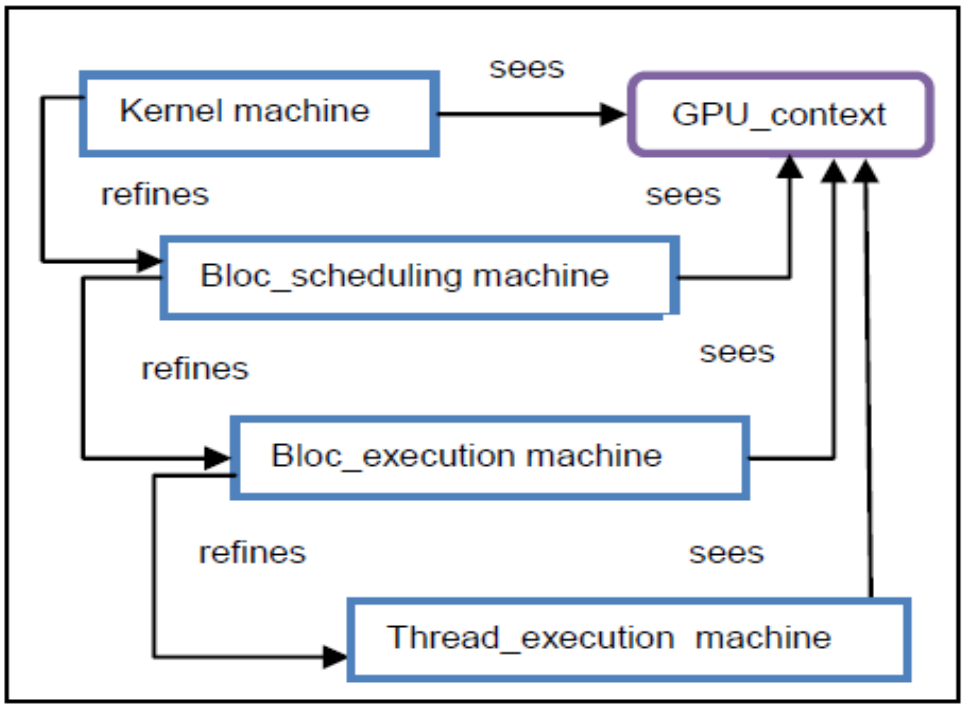


Figure 4.15. Eléments de Spécification Event B du GPU

5.1Context GPU_Context

Ce contexte permet de définir les structures de données et les nouveaux ensembles utilisés dans toutes les machines. Pour couvrir les états de chaque composant logiciel, nous proposons les ensembles suivants :

- **KENELSTATES** qui définit les états du kernel pendant l'exécution. Un kernel est soit en état **waiting** quand le GPU est utilisé par un autre kernel, soit en état **executing** pendant son exécution et en état **ending** quand il se termine.
- **Blocstates** qui définit les états du bloc depuis leur création jusqu'à la fin de leur exécution. Un bloc est en état **wait** jusqu' la libération d'un SM, **run** pendant son exécution et **end** à la fin de son exécution.
- **Threadstates** qui représente les états du thread. Les états sont : asleep, ready, running ou finishing.
- **Instructionstates** qui représente les états de l'instruction pendant l'exécution. Une instruction est soit **blocked** ou **inexecution**.

Pour la représentation d'état de la mémoire nous avons défini un ensemble MG_ACCESS qui est soit accessible soit inaccessible.

Ces ensembles sont utilisés pour déclarer les variables des machines Kernel, Bloc_scheduling, Bloc_execution et Thread_execution .

CONTEXT
GPU_Context
SETS
Blocstates
Threadstates
KERNELSTATES
KERNELS
BLOCS
ACCESS_MG
RESOURCES
Instructionstates
CONSTANTS
wait
run
end
BLOC_STATE_Array
TIMING_tables
size
asleep
ready
running
finishing
Thread_state_Array
waiting
executing
Ending
Kernel

Bloc
accessible
notaccessible
r
verif_dispo
inexecution
blocked

AXIOMS

axm4 : size∈ℕ1
axm1 : Blocstates={wait,run, end}
axm2 : BLOC_STATE_Array∈1‥size→Blocstates
axm3 : TIMING_tables∈1‥size→ℕ
axm5 : Threadstates={asleep,ready,running,finishing}
axm6 : Thread_state_Array∈1‥32→Threadstates
axm7 : KERNELSTATES={waiting,executing,ending}
axm8 : kernel∈KERNELS
axm9 : bloc∈BLOCS
axm10 : ACCESS_MG={accessible,notaccessible}
axm11 : r∈RESOURCES
axm12 : verif_dispo∈RESOURCES↔BOOL
axm13 : Instructionstates={inexecution,blocked}

END

5.2 La machine Kernel

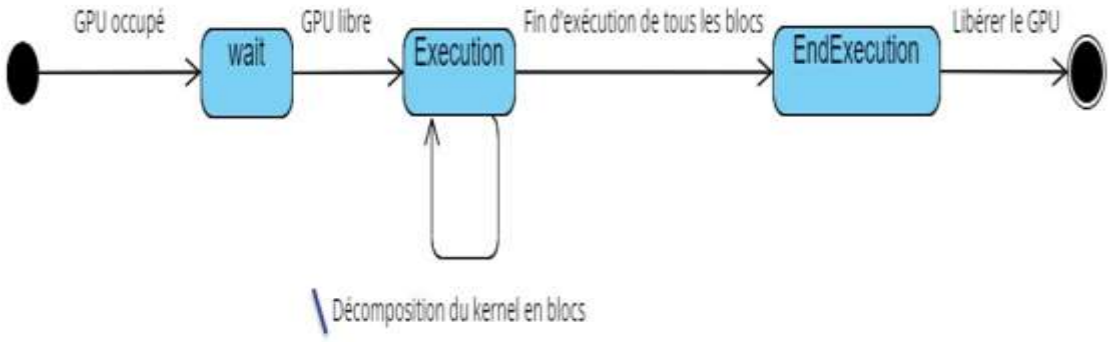


Figure 4.16. Les évènements de la machine kernel

La machine Kernel contient plusieurs variables : nombre des SMs du GPU d'exécution, le nombre total des threads, temps de début, temps de fin d'exécution et une variable T_ev pour représenter l'évolution du temps à chaque étape. Comme illustré dans la Figure 4.16, le GPU a plusieurs états : wait, Execution et EndExecution. Tant que le GPU est occupé, le kernel est en état **waiting** jusqu'à la libération du GPU. Lorsque le kernel accède en état **exécution,** il décompose le kernel en N blocs.

***N=(nombre de threads du kernel/32)/nombre des SMs* dans le GPU d'exécution**)

Pour finir son exécution tous les blocs doivent être exécutés. Comme illustré dans La figure 4.17, pour contrôler l'exécution des états des blocs nous proposons d'utiliser trois tableaux à N dimension. Un tableau représentant l'état d'exécution des blocs, un tableau de début d'exécution des blocs et un tableau des valeurs de temps de fin d'exécution de ces blocs. Ces tableaux sont initialisés au niveau de la machine Kernel et mis à jour au niveau de la machine d'exécution de chaque bloc. Initialement le tableau des états de bloc est initialisé par **wai**t pour tous les blocs.

Tableau d'état de blocs

1	2	3		N
run	**run**	**Run**	……..	**wait**

Tableau de temps de début d'exécution

1	2	3		N
T_ev	**T_ev**	**T_ev**		**T_ev**

Tableau de temps de fin d'exécution

1	2	3		N

Figure 4.17. Les tableaux de contrôle d'exécution des blocs

Au niveau de la machine Bloc_scheduling, les nb_SM_GPU premiers éléments du tableau état de bloc sont modifiés de **wait** en **run**. A chaque fois qu'un bloc finit son exécution, l'état du bloc suivant doit être changé en état **run**. A la fin de l'exécution tous les éléments du tableau sont remis à **end.** Le Code de la machine Kernel est illustré ci-dessous.

MACHINE
    Kernel
SEES
    GPU_context
VARIABLES
    nb_SM_GPU
    nb_kernel_threads
    kernel1
    affect
    Nbreiter
    GPU_OCC
    k_state
    Time_start
    Time_end
    T_ev
    Rest
    C
    blocsArray
    blocs_start_time
    blocs_end_time
INVARIANTS

inv11 : kernel1∈KERNELS
inv1 : nb_SM_GPU∈ℕ1
inv2 : nb_kernel_threads∈ℕ1
inv3 : affect∈ℕ1
inv4 : Rest∈ℕ
inv5 : nbreiter∈ℕ1
inv6 : GPU_OCC∈BOOL
inv7 : Time_start∈ℕ
inv8 : Time_end∈ℕ
inv9 : T_ev∈ℕ
inv10 : k_state∈KERNELSTATES
inv12 : c∈ℕ1
inv13 : blocsArray∈1‥nb_kernel_threads→Blocstates
inv14 : ran(blocsArray)=ran(BLOC_STATE_Array)
inv15 : blocsArray∈ℕ→Blocstates
inv16 : blocs_start_time∈1‥nb_kernel_threads→ℕ
inv17 : ran(blocs_start_time)=ran(TIMING_tables)
inv18 : blocs_start_time∈ℕ→ℕ
inv19 : blocs_end_time∈1‥nb_kernel_threads→ℕ
inv20 : ran(blocs_end_time)=ran(TIMING_tables)
inv21 : blocs_end_time∈ℕ→ℕ

EVENTS

INITIALISATION ≙
STATUS
ordinary

BEGIN
act1 : nb_kernel_threads≔1024
act2 : nb_SM_GPU≔16
act3 : affect≔1
act4 : Rest≔0
act5 : nbreiter≔1
act6 : GPU_OCC≔FALSE
act7 : Time_start≔0
act8 : k_state≔waiting

END

EXECUTION ≙
STATUS
ordinary

WHEN
grd1 : k_state=waiting
grd2 : GPU_OCC=FALSE

THEN
act1 : GPU_OCC≔TRUE
act2 : k_state≔executing
act3 : affect≔nb_kernel_threads÷32
act4 : nbreiter≔affect÷nb_SM_GPU
act5 : T_ev≔Time_start+c // *//temps d'affectation et de calcul*

END

ENDEXECUTION ≙
STATUS
ordinary

```
BEGIN
        act1  :  GPU_OCC:=FALSE
        act2  :  Time_end:=T_ev
        act3  :  k_state:=ending
END

      WAIT  ≙
      STATUS
        ordinary
WHEN
        grd1  :  GPU_OCC=TRUE
THEN
        act1  :  k_state:=waiting
END

END
```

5.3 La machine Bloc_execution

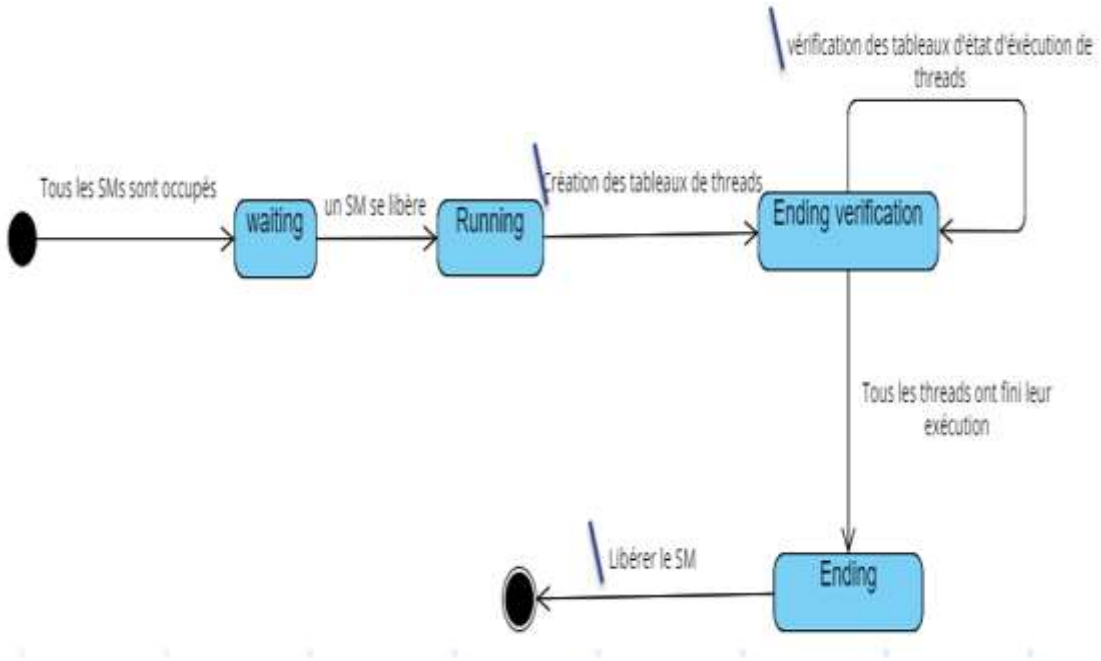


Figure 4.18. Les évènements de la machine bloc_executing

Les états des SMs sont illustrés dans la Figure 4.18. Quand un état de bloc est mis à jour par l'état **run.** Comme illustré la Figure 4.19, trois tableaux sont créés pour contrôler l'exécution des 32 threads du bloc : un tableau d'état de threads, un tableau des valeurs de temps de début d'exécution et un tableau des valeurs de temps de fin d'exécution. Les threads sont initialisés à l'état **ready**. Pour modifier l'état de bloc en **end** tous les éléments du tableau de threads doivent être égaux à **finishing**. Le temps d'exécution d'un bloc est le max des temps de fin d'exécution des threads.

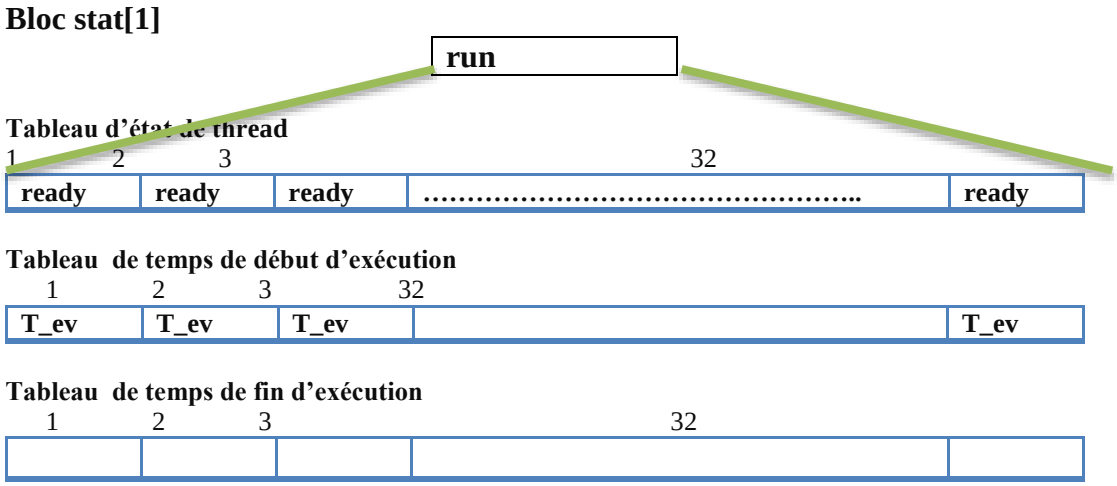


Figure 4.19. Tableaux de contrôle d'exécution des 32 threads d'un bloc

La machine Bloc_execution est présentée comme suit :

MACHINE
Bloc_execution
REFINES
blocs_scheduling
SEES
GPU_context
VARIABLES
i
j
k
increm
End_iteration
current_bloc
threadsArray
threads_start_time
threads_end_time
pointeur
pointeur1
TimeSet
pointeur2
INVARIANTS
inv1 : threadsArray∈1‥32→Threadstates
inv2 : ran(threadsArray)=ran(Thread_state_Array)
inv3 : threadsArray∈ℕ→Threadstates
inv4 : threads_start_time∈1‥32→ℕ
inv5 : ran(threads_start_time)=ran(TIMING_tables)
inv6 : threads_start_time∈ℕ→ℕ
inv7 : threads_end_time∈1‥32→ℕ
inv8 : ran(threads_end_time)=ran(TIMING_tables)
inv9 : threads_end_time∈ℕ→ℕ
inv10 : pointeur∈ℕ
inv11 : pointeur1∈ℕ
inv12 : TimeSet⊂ℕ
inv13 : pointeur2∈ℕ
EVENTS

INITIALISATION ≙
extended
STATUS
ordinary
REFINES
Affectation
BEGIN
act1 : nb_kernel_threads≔1024
act2 : nb_SM_GPU≔16
act3 : affect≔1
act4 : Rest≔0
act5 : nbreiter≔1
act6 : GPU_OCC≔FALSE
act7 : Time_start≔0
act8 : k_state≔waiting
act9 : i≔0
act10 : j≔1
act11 : k≔33
act12 : increm≔1
act13 : End_iteration≔FALSE
act14 : current_bloc≔1
act15 : pointeur≔1
act16 : pointeur1≔1
act17 : TimeSet≔∅
act18 : pointeur2≔1
END
bloc_waiting ≙
STATUS
ordinary
ANY
M
WHERE
grd3 : m∈ℕ
grd2 : blocsArray(m)=wait
grd4 : pointeur≤32
THEN
act1 : threadsArray(pointeur)≔asleep
act2 : threads_start_time(pointeur)≔T_ev
act3 : pointeur≔pointeur+1
END
bloc_executing ≙
STATUS
ordinary
ANY
M
WHERE
grd1 : m∈ℕ
grd2 : blocsArray(m)=run
grd3 : pointeur1≤32
THEN
act1 : threadsArray(pointeur1)≔ready
act2 : pointeur1≔pointeur1+1
END

```
    bloc_ending_verification  ≙
    STATUS
      ordinary
ANY
      M
WHERE
      grd1  :  m∈ℕ
      grd2  :  blocsArray(m)=run
      grd3  :  pointeur2≤32
      grd4  :  threadsArray(pointeur2)=finishing
THEN
      act2  :  TimeSet≔TimeSet∪{threads_end_time(pointeur2)}      // Max
      act3  :  pointeur2≔pointeur2+1      // retour au scheduler
END

    bloc_ending  ≙
    STATUS
      ordinary
REFINES
      verifying_execution_end
ANY
      M
WHERE
      grd2  :  m∈ℕ
      grd1  :  blocsArray(m)=run
THEN
      act1  :  blocs_end_time(m)≔max(TimeSet)
      act2  :  T_ev≔max(TimeSet)
      act3  :  blocsArray(m)≔end
END

END
```

5.4 Machine Thread_execution

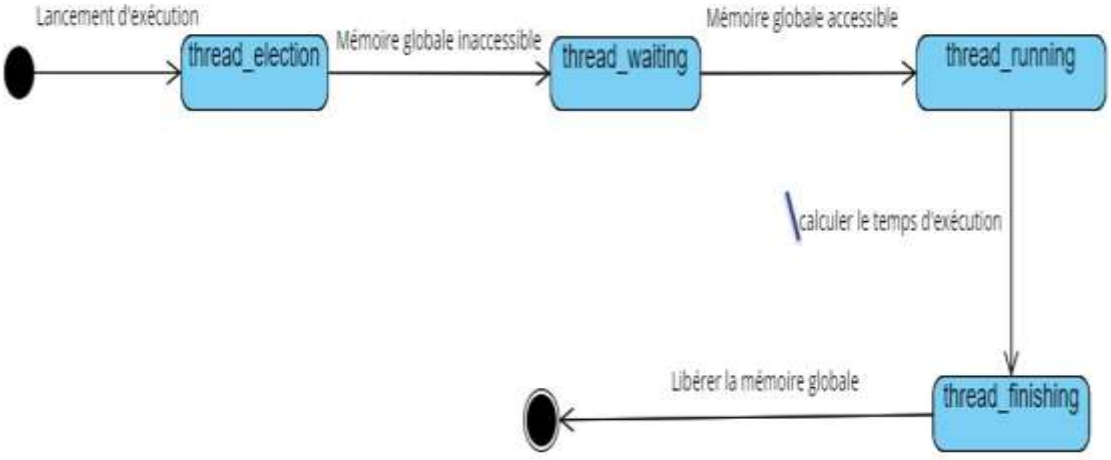


Figure 4.20. Les évènements de la machine Thread_execution

Au niveau d'exécution de thread, il y a la notion de concurrence sur l'accès à la mémoire globale. Pour cela on propose d'utiliser une variable de contrôle d'accès à la mémoire globale. Comme illustré dans la Figure 4.20, lorsque le thread est élu son état sera mis en **ready**. Tant que la mémoire globale est inaccessible il restera dans cet état et le temps d'attente va être ajouté au temps d'exécution. Si la mémoire est accessible, l'état du thread passe à l'état **running** et l'état de la mémoire deviendra inaccessible. Quand un thread finit son exécution son état est **finishing**, il libère la mémoire globale et met à jour son temps de fin d'exécution. Le Code de la machine Thread_execution est illustré dans ce qui cuit.

MACHINE
Thread_execution
REFINES
Bloc_execution
SEES
GPU_context
VARIABLES
MGA // *accès à la mémoire global*
duration
pos // *position du thread dans le bloc (indice)*
INVARIANTS
inv1 : MGA∈ACCESS_MG
inv2 : duration∈ℕ
inv3 : pos∈ℕ
EVENTS
INITIALISATION ≙
extended
STATUS
ordinary
REFINES
bloc_executing
BEGIN
act19 : MGA≔accessible
act20 : duration≔14
act21 : pos≔1
act22 : threadsArray(pos)≔asleep
END
Thread_election ≙
STATUS ordinary
ANY
M
WHERE
grd2 : blocsArray(m)=run
grd1 : threadsArray(pos)=asleep
THEN
act1 : threadsArray(pos)≔ready
act2 : threads_start_time(pos)≔T_ev
act3 : threads_end_time(pos)≔threads_start_time(pos)

```
END
    Thread_waiting  ≙
    STATUS ordinary
WHEN
      grd1  :  threadsArray(pos)=ready
      grd2  :  MGA=notaccessible
THEN
      act1  :  threads_end_time(pos):=threads_start_time(pos)+1
END
    Thread_running  ≙
    STATUS
      ordinary
WHEN
      grd1  :  threadsArray(pos)=ready
      grd2  :  MGA=accessible
THEN
      act3  :  MGA:=notaccessible
      act1  :  threadsArray(pos):=running
      act2  :  threads_end_time(pos):=threads_end_time(pos)+duration
END
    Thread_finishing  ≙
    STATUS ordinary
WHEN
      grd1  :  threadsArray(pos)=running
THEN
      act1  :  threadsArray(pos):=finishing
      act2  :  MGA:=accessible
END
END
```

5.5 Preuves de RODIN

Afin de valider la spécification proposée, nous appliquons les preuves RODIN sur la spécification Event B. Le tableau 6 suivant représente les statistiques de preuve obtenu par RODIN. La colonne Auto dans le tableau 6 correspond aux obligations de preuves générées automatiquement par RODIN. Des preuves manuelles sont utilisés pour éliminer les invariants non prouvés.

Modèle	Total	Auto
Kernel	25	15
Bloc_scheduling	22	12
Bloc_execution	38	18
Thread_execution	17	10
Total	102	55

Tableau 6. Statistiques d'obligations de preuves

6 Génération de code CUDA/OpenCL à partir d'une spécification UML/MARTE de GPU

Nous proposons de raffiner la spécification Event B en un pré-code d'Event B du code comme une étape préliminaire avant de passer au code exécutable.

6.1 Etude de cas de l'algorithme d'addition vectorielle

Afin de montrer l'approche de raffinement d'une spécification Event B d'une application à un pré-code (CUDA et OpenCL), nous reprenons l'algorithme d'addition vectoriel. Le raffinement des machines est illustré dans la Figure 4.21.

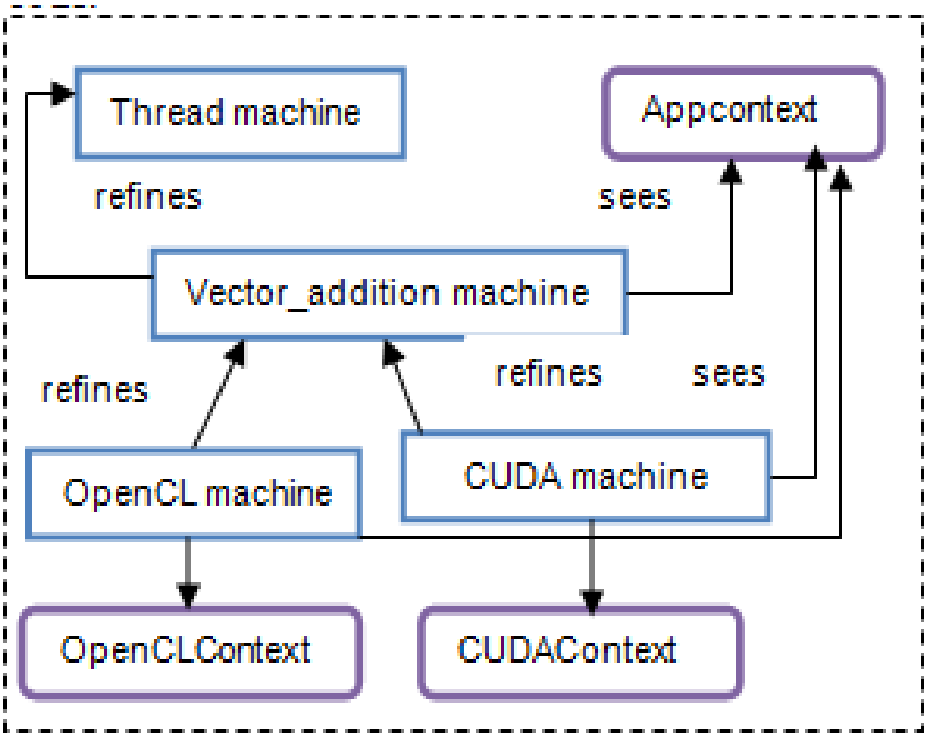


Figure 4.21. Raffinement de la machine code

6.2 Pré-code CUDA

Dans un programme CUDA typique, les données sont envoyées de la mémoire centrale vers la mémoire GPU, le CPU envoie des commandes au GPU, ensuite le GPU exécute les noyaux de calcul en ordonnançant le travail sur le matériel disponible pour enfin copier le résultat de la mémoire GPU vers la mémoire CPU.

Pour représenter le code CUDA, on a définit la structure du code dans un contexte CUDA et le code a été écrit dans une machine appelé machine CUDA. La machine et le context CUDA sont illustrés dans la Figure 4.22.

Contexte

CONTEXT
CudaContext

SETS
TRANSFERS

FREES

MALLOCS

CONSTANTS
HostToDevice

DeviceToHost

free

nonfree

Malloc

nonalloc

AXIOMS
axm1 : TRANSFERS={HostToDevice,DeviceToHost}

axm2 : FREES={free,nonfree}

axm3 : MALLOCS={malloc,nonalloc}

END

Machine CUDA

MACHINE
CudaMachine

REFINES
vectorAddition

SEES
AppContext

CudaContext

VARIABLES
DeviceA

DeviceB

DeviceC

EVENTS

INITIALISATION ≙

CudaAllocation ≙

CUDA_kernel_thread ≙

CUDA_Results_disply ≙

END

Figure 4.22. Pré-code Cuda

6.3 pré-code OpenCL

OpenCL est basé sur une plateforme qui divise le système en un hôte (host) et une ou plusieurs dispositifs (devices) de calcul. Ces derniers agissent en tant que coprocesseur pour l'hôte. Ils sont subdivisés en plusieurs unités de calcul (CUs) qui sont aussi divisées en multiple élément de traitement (PEs). Une application OpenCL est exécutée

sur un hôte qui envoi des instructions défini sous forme de fonctions appelées kernels aux dispositifs. Dans le cas du pré-code OpenCL, nous avons défini la structure dans un contexte et nous avons écrit le code dans la machine OpenCL. La machine et le context OpenCL sont illustrés dans la Figure 4.23.

Contexte OpenCL

OpenCLContext

SETS
Memory_access

transfertype

CONSTANTS
CL_MEM_COPY_HOST_PTR

CL_MEM_READ_ONLY

CL_MEM_WRITE_ONLY

AXIOMS
axm1 : Memory_access={CL_MEM_READ_ONLY, CL_MEM_WRITE_ONLY}

axm2 : transfertype={CL_MEM_COPY_HOST_PTR}

END

Machine OpenCL

MACHINE
OpenCL_context_machine

REFINES
vectorAddition

SEES
AppContext

OpenCLContext

VARIABLES
DeviceA

DeviceB

DeviceC

EVENTS
INITIALISATION ≙

OpenCL_allocation ≙

OpenCL_kernel_thread ≙

OpenCL_EnqueuReadBuffer ≙

END

Figure 4.23. Pré-code OpenCL

7 Etude de cas : Détection parallèle de pics des images de spectre Raman en utilisant OpenCL

La spectrométrie Raman est une technique qui permet la détection des produits chimiques à travers un certain nombre de pics trouvés dans un spectre d'images ou dans une série numérique de données[112]. La spectrométrie Raman est une technique nécessaire dans de nombreux domaines tels que : la physique, la chimie et la biologie. La machine de Raman spectrophotomètre analyse un produit et génère des images sous forme de courbe ou sous forme de fichier texte ou de fichier CSV (Comma Separated values) [112,113]. L'interprétation des pics de courbes permet de détecter l'origine du produit chimique analysé en utilisant des bases de données spécialisées [114,115]. Les biologistes font cette opération manuellement in vitro, ce qui la rend dur et longue en termes de temps. Notre but est d'automatiser la détection des molécules en utilisant des techniques de traitement d'image. Nous proposons une solution parallèle pour détecter les pics en utilisant OpenCL sur GPU. La Figure 4.24 montre l'approche de détection de pics.

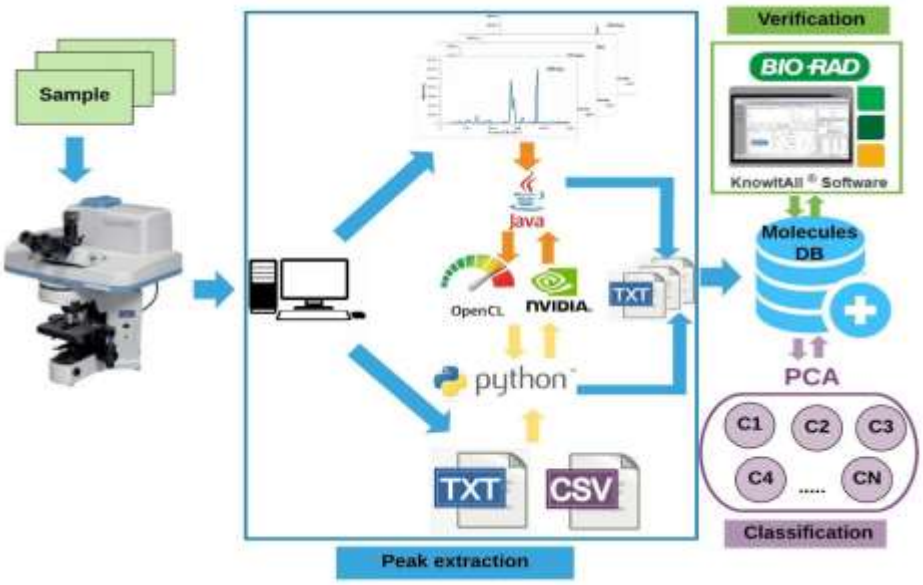


Figure 4.24. Approche de détection de pics

7.1 Travaux connexes

De nombreux travaux ont été proposés pour automatiser la technique de détection des pics. [115] présente une comparaison entre les approches de classification supervisées et non supervisées pour analyser les images Raman spectrales. Au début, l'ensemble de

données résultant est un tableau de trois dimensions de valeurs avec la distance spatiale dans les dimensions de x et y et la longueur d'onde ou la fréquence pour la dimension z. Ils ont supposé que l'analyse de regroupement a prouvé qu'elle est extrêmement utile pour l'investigation de l'image spectrale des données. C'est une solution robuste, rapide et non-subjective. Un autre travail [116] présente une nouvelle méthode pour l'extraction des sommets (extrêmes) non supervisé à partir de données hyper spectrales nommée analyse de composants (VCA). Les résultats VCA ont montré une énorme performance en termes de temps et d'exactitude. [117] présente une comparaison détaillée de six algorithmes multi variés de l'analyse d'images Raman par spectroscopie microscopique. L'objectif de ce travail est de montrer l'impact de la nouvelle proposition implémentation de l'algorithme N-FINDR. Le facteur commun entre les approches choisies est que les données hyper spectrales ont été considérées comme une matrice bidimensionnelle. [118] a développé une nouvelle approche en mettant en œuvre une méthode d'imagerie chimique pour détecter plusieurs adultérant dans la poudre de lait écrémé en utilisant une classification basée sur des images de Raman . [1] a proposé une approche appliquée en biologie établissant un flux de traitement de données pour discrétiser le spectre Raman continu à spectre discret, nommé rDisc (discrétisation du spectre Raman). Le rDisc comprend deux parties principales : la procédure de contrôle de qualité Raman et la discrétisation Raman par des pics représentatifs.

La plupart des travaux notés précédemment dans la détection des pics représentatifs sont basés sur des techniques de classification contrairement à l'approche proposée où nous utilisons uniquement les techniques de traitement d'image. En ce qui concerne le bruitage et le prétraitement des images spectrales qui sont nécessaires, nous proposons une nouvelle approche pour extraire les pics représentatifs en utilisant le traitement d'image sur GPU.

7.2 Raman spectroscopie

La spectroscopie Raman est une technique de spectroscopie vibrationnelle fournissant un outil sensible, relativement rapide, non destructif des moyens de sondage de la structure moléculaire en phase solide et liquide. Il est également pratique pour l'analyse de la dépendance de conformation et des interactions intra- et inter-chaînes [115,120] .

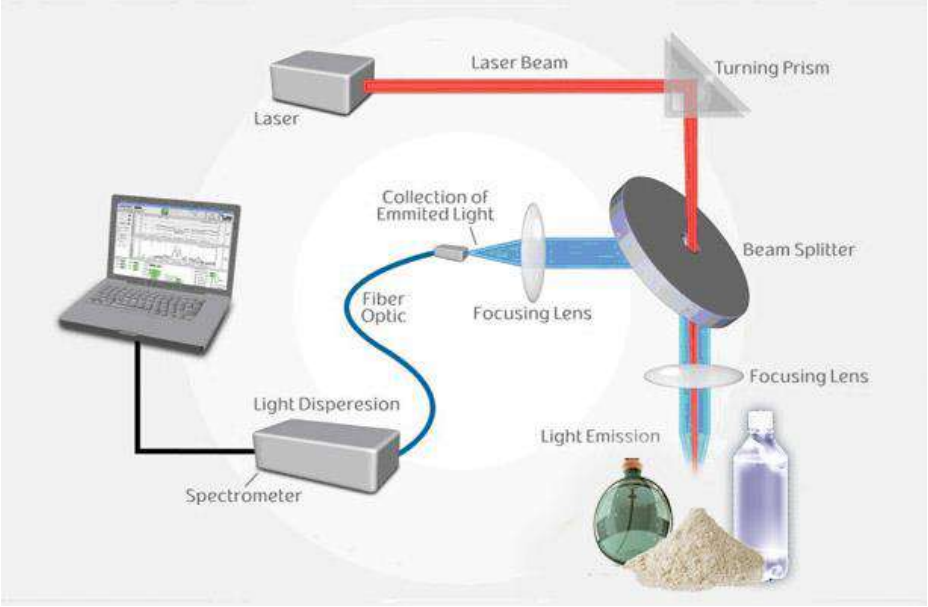


Figure 4.25. Un système typique de détection de pics de spectrométrie Raman

Quand la lumière interagit avec un échantillon, une partie de l'énergie est absorbée, l'autre est dispersée. De la dernière partie, la plupart des photons sont dispersés élastiquement avec la même énergie et la même longueur d'onde de la lumière incidente (diffusion de Rayleigh). Une petite fraction, environ 1 sur 10 millions de photons, est inélastiquement dispersée, avec les photons dispersés ayant une fréquence différente de la lumière incidente.

Comme illustré dans la Figure 4.25, un Raman spectroscopie basique contient quatre parties principales :

- une source laser fournissant une excitation par faisceau laser;
- un microscope optique capable de focaliser le laser sur une cellule et de collecter simultanément le Raman dispersé des photons;
- un filtre qui rejette les photons élastiques de Rayleigh;
- un spectromètre à réseau équipé d'un dispositif chargeur couplé (CCD) détecteur de tableau pour acquérir les fréquences spécifiques du Raman dispersées poutres [120,121,122,123].

7.3 Approche proposée de détection des pics à partir des images

Afin d'automatiser l'opération de détection de la molécule la spectrométrie Raman, nous proposons de développer une solution parallèle en utilisant OpenCL. La Figure 4.26

montre les phases de l'approche proposée de détection de pics. L'approche peut être divisée en trois phases : 1)Analyse de la molécule à l'aide d'un spectromètre Raman, 2)Réception et Traitement des images pour extraire les pics représentatifs, 3) Recherche des pics à partir d'un corpus de molécules.

Nous visons à optimiser la deuxième phase primordiale. L'approche utilise les techniques de traitement d'image pour extraire les pics représentatifs des images spectrales. Ce traitement est gourmand en temps pour trouver les pics représentatifs. Alors, nous proposons un algorithme parallèle sur GPU pour optimiser le temps estimé en utilisant les techniques de GPGPU. En commençant par un prétraitement pour d'ébruiter l'image. Ensuite, les pics représentatifs sont extraits. Par la suite, nous lancerons une recherche dans le corpus de molécules utilisant l'intervalle comme requête. Le résultat de cette recherche est l'identificateur, le type et la famille chimique de la molécule.

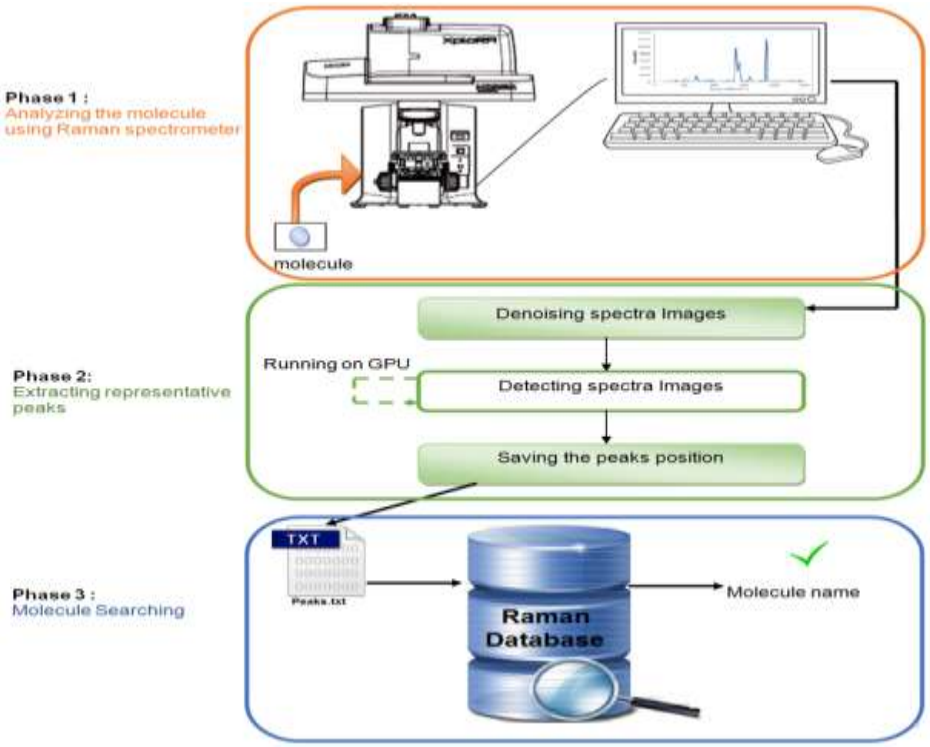


Figure 4.26. Phases de détection de pics

a) Prétraitement des images spectrales

Les images générées par le spectromètre Raman sont généralement bruitées ce qui rend le processus d'extraction des pics représentatifs plus difficile. Pour résoudre ce

problème, nous proposons de convertir l'image en une image binaire. Il sera plus facile de traiter l'image binaire sans bruit. Pour convertir nos images en images en noir et blanc, nous avons utilisé une fonction simple et efficace de Java à partir des outils d'image[1]. Un exemple du processus de conversion est démontré sur la Figure 4.27.

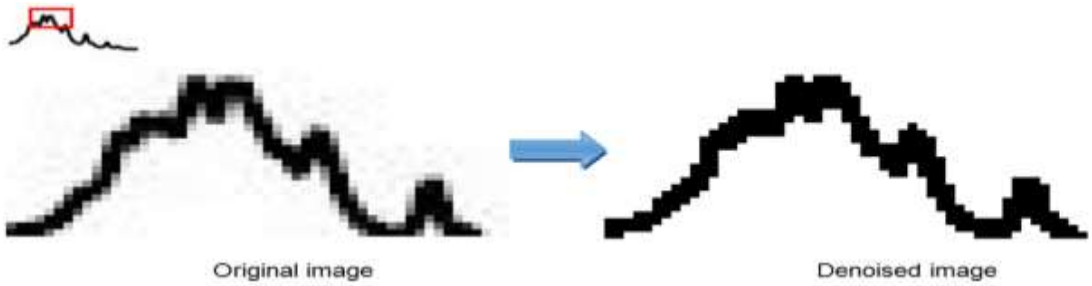


Figure 4.27. Conversion d'image en format binaire

b) Algorithme de détection de pics

Afin d'extraire des données utiles de l'image du spectre, nous proposons de mettre en œuvre une méthode de détection de pic en utilisant des techniques de traitement d'image. Sachant que nous avons un défis associés à la définition correcte des coordonnées (x, y) en utilisant la spectroscopie Raman, la question qui se pose est de savoir comment les données sont traitées et analysées pour être significatives [125,126].L'approche proposée traite l'image pixel par pixel en utilisant uniquement des images binaires. En observant les spectres des images Raman, nous trouvons que le pixel peut prendre l'une des cinq formes comme indiqué dans la Figure 4.28. Après la définition des formes de pixels, le reste du travail est de parcourir l'image pixel par pixel afin de trouver des pics représentatifs.

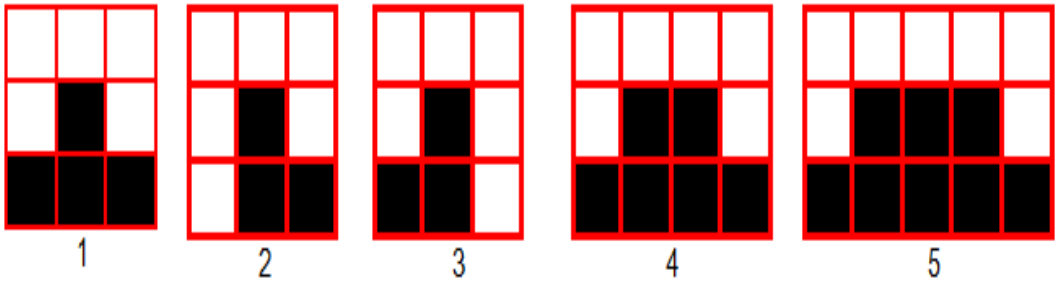


Figure 4.28. Les cinq formats de pics

Un certain nombre de pixels est traités en même temps en utilisant des cœurs du GPU Nvidia. L'algorithme est converti en un kernel, qui sera exécuté pour chaque élément

[1] Java Image tools: http://ddsdx.uthscsa.edu/dig/itdesc.html

de travail. Les éléments de travail sont regroupés en groupes. Pour exécuter le noyau sur GPU, le traitement est structuré en cinq étapes:

- Transférer l'image dans la mémoire du GPU.
- Diviser les pixels de l'image sur les éléments de traitement GPU en fonction de l'image de taille (N) pour créer des éléments de travail.
- Créer des groupes de travail en fonction des multiprocesseursde GPU.
- Lancer le kernel de détection des pics.
- Transférer le résultat de la mémoire du GPU dans la Mémoire du CPU.

La Figure 4.29 illustre un exemple d'extraction de pics à partir une image spectrale de liquide ionique .

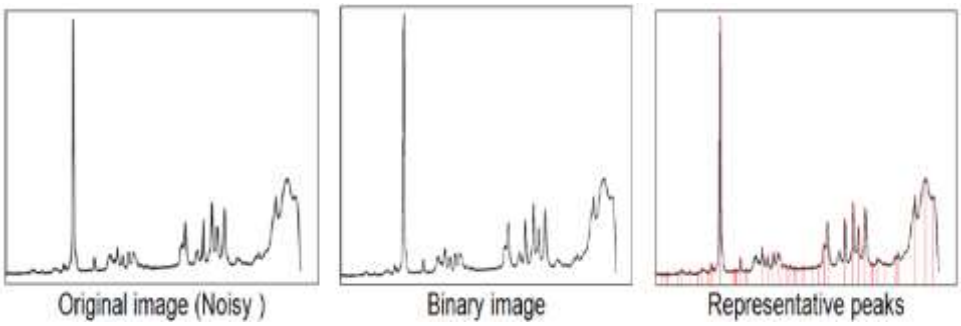


Figure 4.29. Extraction de pics à partir d'une image spectrale

7.4 Détection des pics à partir des fichiers texte et CSV

Afin de générer des images spectrales à partir de données numériques pour la détection d'intervalles ciblés avant le calcul automatique des pics, nous proposons un algorithme parallèle en utilisant OpenCL et des fonctions mathématiques de PYTHON. Ce développement peut être divisé en trois phases :1) Importation des données brutes, 2) Génération instantanée du spectre, 3) Détection des Pics tel que présentées dans la Figure 4.30.

Figure 4.30. Phases de détection de pics à partir des fichiers CSV

Le Raman génère généralement des données numériques sous forme de fichiers txt ou CSV. Un fichier txt est un fichier contenant une série de caractères ; il utilise un format de codage de caractères spécifique. CSV est un fichier txt qui représente un tableau, où les lignes sont séparées par des virgules. Les valeurs séparées par des virgules sont également le contenu de la cellule. Une ligne CSV est une séquence de caractères se terminant par un saut de ligne (-CRLF). Afin d'utiliser des données numériques, nous proposons un algorithme qui trace une courbe reliant l'abscisse et l'ordonnée indiquées dans les tableaux.

Nous avons implémenté un algorithme pour détecter un pic dans un intervalle donné en utilisant la méthode locale maximale après avoir généré une courbe à l'aide des données numériques obtenues à partir des fichiers txt et CSV. La méthode du maximum local consiste à délimiter les valeurs à l'intervalle donné avec une valeur maximum et une valeur minimum [127,128,129]. Dans un intervalle donné [a, b], on peut supposer que M est un maximum local où f en M est supérieur ou égal à la hauteur à tout autre intervalle, plus précisément :

$$f(M) \geq f(x) \ . \ \forall x \in [a, b]$$

Nous avons implémenté un algorithme basé sur la méthode de la valeur maximale en cliquant sur un intervalle dans l'image. La valeur maximale est déterminée en

comparant chaque valeur à ses valeurs adjacentes à un intervalle donné et est ensuite définie comme un pic.

Le langage Python et la bibliothèque PyOpenCL ont été utilisés pour extraire les pics représentatifs des fichiers txt et CSV permettant le calcul parallèle OpenCL. Le fichier CSV est exploité pour tracer un spectre qui relie les points et déterminer la valeur locale maximale dans chaque intervalle jusqu'à ce que tous les pics soient extraits, la Figure 4.31 exprime cette détermination de valeurs locales..

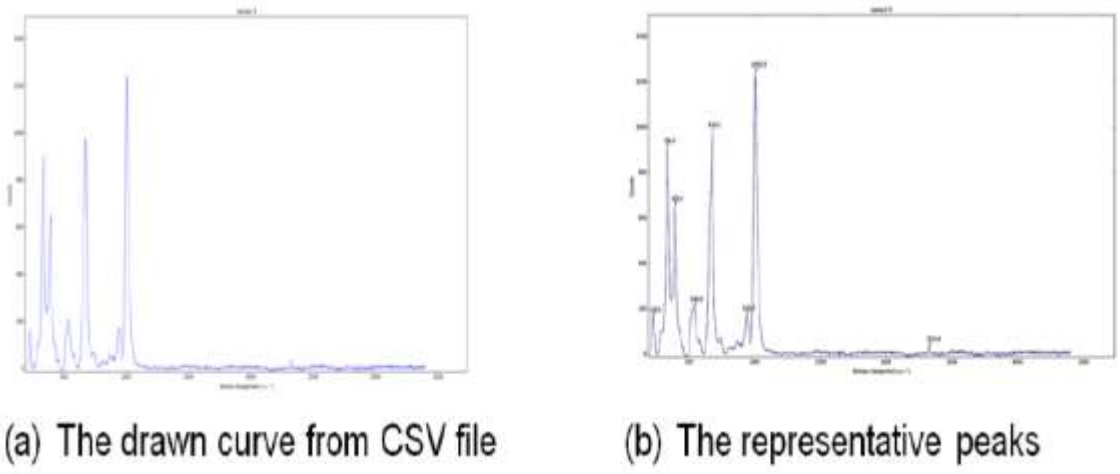


Figure 4.31. Exemple d'extraction de pics représentative à l'aide d'analyse de données CSV parallèle

7.5 Analyse par Composante Principale

L'ACP est utilisé comme méthode de modélisation des relations entre différentes variables décrivant le comportement de tout système [130]. L'estimation des paramètres du modèle ACP est accomplie en déterminant les valeurs propres des données et les vecteurs propres de la matrice de corrélation. Le nombre de composants maintenus dans le modèle est toujours calculé afin de déterminer la structure du modèle. De plus, les critères adoptés pour choisir le nombre de composantes dans ce travail reposent sur le principe de reconstitution de la variance, pour sélectionner le nombre de composantes à retenir dans le modèle ACP car il exploite la redondance entre les variables. Le modèle ACP est identifié par le nombre de composants [131].

Le travail porte sur les cas de systèmes de synthèse (dihydroxyadénine, allopurinol, urate d'ammonium, etc.) et des valeurs (Pic1, Pic2, Pic3, Pic4) qui sont traitées séparément. Nous avons opté pour la caractérisation du plus grand maximum du spectre

Raman parmi les paramètres que nous avons fixés pour construire la base de données indiquée dans le tableau 5.

Désignation du biomatériel	Pic 1	Pic 2	Pic 3	Pic 4
Dihdroxyadenine	102.39	381.80	886,00	1469,3
Dihdroxyadenine	102.19	492.88	617,32	980,03
Amorphous carbonated calcium	424.59	604,00	961,00	1439,80
Amorphous carbonated calcium	558,00	607,00	962,00	1452,00
Allopurinol	105.20	717,00	947.62	1393.60
Allopurinol	107.14	717.87	1394,00	1576,00
Ammonium urate	629.89	999,00	1215,00	1373,00
Ammonium urate	486,00	995,00	1208,00	1356,00
Amoxicillin, trihydrate	633,00	835,00	1774,00	2990,00
Amoxicillin, trihydrate	502,00	744,00	870,00	1051,00

Tableau 7. Extrait de la base de données des biomatériaux

Dans la Figure 4.32, les composants f1 et f2 représentent 82 % des informations, les données de trous sont regroupées en sous-classes basées sur les résultats de l'ACP appliqués aux pics de biomatériaux du tableau 7.

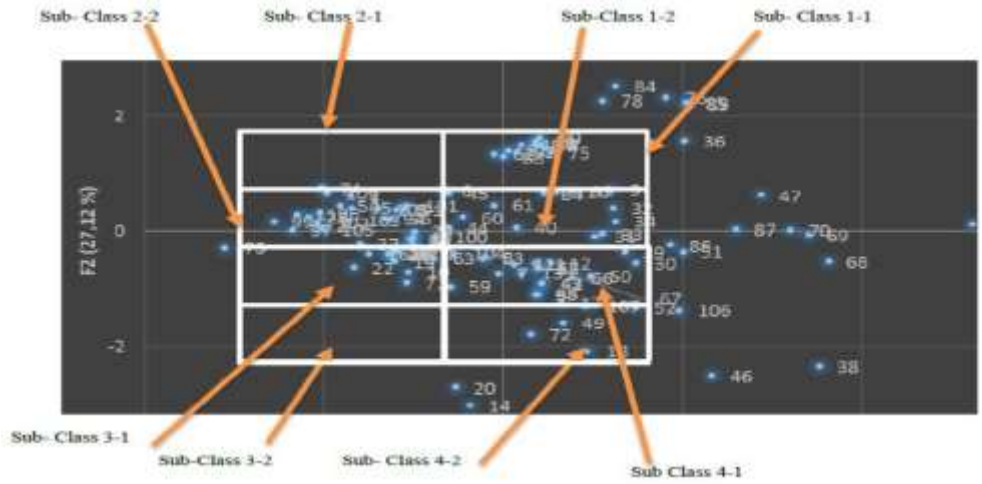


Figure 4.32. Classification des biomatériaux basée sur les résultats de l'ACP

Le tableau 11 des classes générées est présenté dans l'Annexe 1.

7.6 Résultats et simulation

7.6.1 Résultats de l'approche de détection de pics

Pour expérimenter l'approche proposée nous avons préparé une base d'images de 100 images de spectres de liquides ioniques et de fichiers CSV. Les images sont converties

en images binaires. Nous avons exécuté l'algorithme sur une GPU Nvidia Ge Force 630M. C'est une architecture de Fermi composée de 96 cœurs [132].

Dans l'approche de détection de pics par traitement d'image, nous avons trouvé que le temps moyen d'exécution d'une image dans l'approche séquentielle appartient à l'intervalle [0.18, 6.862] alors que dans l'approche parallèle, la mise en œuvre de la durée moyenne d'une image appartient à l'intervalle [0.1, 0.92] incluant le temps de transfert de la mémoire CPU / GPU. Sachant que la durée moyenne d'exécution d'un kernel est entre 8 et 31 millisecondes, nous concluons que le temps est gaspillé dans le transfert de la mémoire entre CPU / GPU.

Le tableau 8 montre la comparaison du temps d'exécution de 100images successivement dans l'approche parallèle et dans l'approche séquentielle. Notant que nous avons lancé les testes des centaines de fois et nous avons pris la valeur moyenne.

L'approche de détection de pics	Type de données	Temps d'exécution sur CPU (ms)	Temps d'exécution sur GPU (ms)
Par traitement d'image	Image de courbe (1)	0.18	0.1
	Image de courbe (100)	18	9.166
Par analyse de données	Fichier CSV (1)	0.06200	0.008902
	Fichier CSV (100)	6.2	0.89

Tableau 8. Comparaison du temps d'exécution d'algorithme de détection des pics CPU vs GPU

Dans l'approche de détection de pics par analyse de données numériques, lors de l'utilisation d'OpenCL sur le processeur, le temps d'exécution a été réduit par rapport à l'implémentation séquentielle utilisant Python. Nous avons également noté que l'implémentation parallèle utilisant PyOpenCL sur GPU Nvidia est plus rapide que l'implémentation séquentielle utilisant PyOpenCL sur CPU.

Les résultats montrent qu'en utilisant le GPU, l'implémentation parallèle a amélioré le processus de détection des pics. Cette amélioration était due au fait que les cœurs du GPU étaient utilisés pour déterminer les valeurs maximales dans chaque intervalle. Cette méthode est efficace, mais il deviendra difficile de définir les intervalles lorsque le nombre de pics est important.

Afin de s'assurer que les résultats trouvés sont corrects et valide, nous avons utilisé la base de données KnowItAll comme benchmark pour l'enseignement de la chimie dans la bibliothèque spectrale. Elle permet un accès illimité à plus de 2 millions des spectres, y compris infrarouge (IR), Raman, RMN (CNMR, HNMR, XNMR), US-Vis et les Spectra de masse (MS) [133]. Elle permet à l'utilisateur de rechercher des collections spectrales en ligne ou en utilisant KnowItAll application. Le progiciel KnowItAll

comprend de nombreux outils tels que la recherche spectrale, l'identification spectrale, spectre IR, la reformulation, l'analyse de mélange et la gestion de données spectrales [134,135].

Nous avons utilisé cette base de données pour valider les résultats. La Figure 4.33 montre un exemple KnowItAll par rapport à l'approche développée de détection des pics.

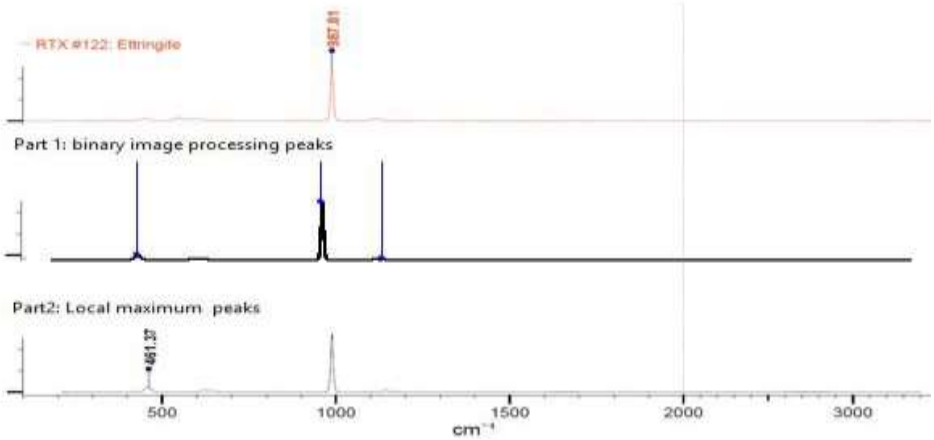


Figure 4.33. Comparaison d'un spectre de KnowItAll par rapport aux résultats de l'approche proposée

7.6.2 Résultats de classification par ACP

L'approche développée permet de comparer les éléments biomatériaux ; un corpus complet est la principale difficulté pour conduire de tels travaux. De nombreux biomatériaux peuvent être regroupés par cluster après traitement car ils ont des relations spécifiques.

Nous avons choisi l'ACP en raison du fait qu'elle permet d'identifier facilement la corrélation entre les points (pics). Nous avons utilisé quatre pics, ce qui limite notre interprétation nécessitant des approches de calcul en temps réel pour le traitement parallèle d'un maximum de spectres. Le spectre Raman est une structure appropriée pour le développement de biocapteurs car il représente une identité réelle d'un matériau donné associé à une plate-forme de calcul.

Nous avons déduit quatre clusters d'un nombre moyen de matériaux approchant 22, ce qui donne un total de 88 éléments sur 112 éléments, soit 78% de la population choisie. Sur chaque cluster, nous avons vérifié la corrélation structurelle jusqu'à 60% du nombre d'éléments du cluster. Le choix des valeurs de pic est donc de déterminer la classification de l'ensemble.

L’approche ACP développée a été comparée aux approches de l'état de l'art qui utilisaient uniquement l’ACP. Le tableau 9 résume la comparaison des différentes approches en termes de type de jeu de données, de nombre de population, de nombre de composantes de l'ACP et de corrélation.

Approches	Base de données	Population	ACPs(clusters)	Corrélation
[136]	Microplastics	50	5	53%
[114]	pathogenic bacteria	50	20	57.5%
[137]	nanoscale extracellular vesicles	20	2	40.5%
[138]	Gram-positive bacteria species	33	3	30%
Our approach	Biomaterial	112	4	60%

Tableau 9. Comparaison des résultats de la classification ACP avec les résultats des travaux connexes

Les résultats montrent que la corrélation de notre approche est légèrement meilleure que celle des autres approches. [114,136,137 ;138] ont considéré l'ACP comme une première méthode de discrimination. Pour obtenir une corrélation allant jusqu'à 90 %, ils ont proposé de combiner l'ACP avec d'autres méthodes de classification telles que le classificateur de forêt aléatoire (RF), l'analyse discriminante linéaire (LDA) et l'apprentissage des réseaux de neurones.

La méthode ACP à travers cette étude de cas a démontré son utilité dans la classification des éléments selon les structures et les fonctions. Cependant, nous pensons que ces résultats pourraient être améliorés dans des travaux futurs. Nos résultats sont prometteurs et devraient être validés par un ensemble de données plus important (base de données de spectrométrie Raman).

8 Utilisation du plug-in Theory Event B

Dans les versions récentes de la plateforme RODIN, il n'y avait pas d'outils disponibles pour définir de nouveaux opérateurs ou pour étendre les prouveurs standard avec de nouvelles règles de preuve. Pour surmonter cette limitation, Nous utilisons les Theory (Voir Chapitre 2).

Notre but dans ce qui suit est d’utiliser la Theory Event B pour créer une théorie qui permet de générer un code OpenCL valide pour les cas d’utilisations présentés précédemment d’addition vectoriel et de détection des pics.

8.1 Theories Data_List et Program_List

Pour présenter l'accès et l'utilisation des données dans OpenCL, nous proposons d'utiliser la théorie de la notion de liste. La proposition est basée sur la liste FIFO, qui charge et décharge les entrées et les sorties dans deux listes dépendantes :

La première liste contient les données et la deuxième liste contient les instructions du programme. La Figure 4.34 ci-dessous montre l'empilement. Lorsqu'un registre est nécessaire dans l'application, il sera empilé dans la liste de données, ainsi que son adresse en mémoire sera empilée dans la liste de programmes pour s'assurer que le programme peut utiliser les données.

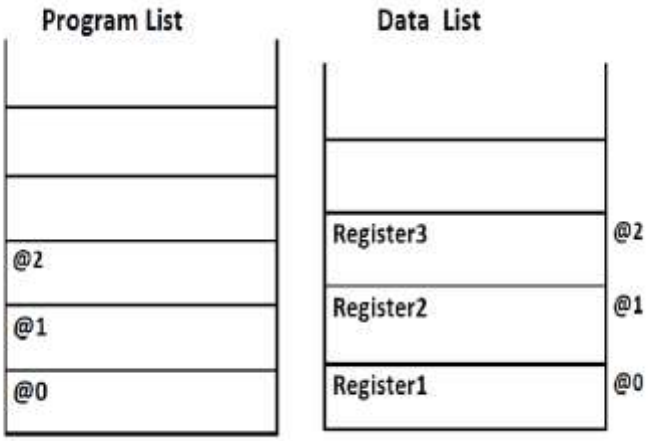


Figure 4.34. Listes Data et Program

- **Data_List Operator** contient la liste des registres dont la taille ne peut pas dépasser maxx et que l'utilisateur peut indiquer. L'opérateur Data_List est illustré dans La Figure 4.35.

```
OPERATORS
Data_List:  Data_List(Register : Data, l : List(Data), maxx : 1‥10, Regsiter
ADR : ℙ(Data))
well-definedness condition
       ListSize(l) ≤ maxx
       RegisterADR ∈ ARRAY(Register)
recursive definition
       case l
          Data_List(Register, nil, maxx, RegsiterADR) ≙ ListSize(l) = 0
          Data_List(Register, cons(Register, l0), maxx, RegsiterADR) ≙
           1+listSize(l0)
```

Figure 4.35. Data_List operator

- **PGM_List Operator :** contient la liste d'adresses des registres. La Figure 4.36 montre l'opérateur PGM_List.

```
PGM_List : PGM_List(PL: List(RegisterADR))
recursive definition
        case PL
          PGM_List(nil) ≙ ListSize(PL) = 0
          PGM_List (cons(RegisterADR, PL0)) ≙ 1+ListSize(PL0)
END
```

Figure 4.36. PGM_List Operator

8.2 Machines d'OpenCL

Pour écrire une machine OpenCL, nous devons avoir une application spécifique. Pour une application générale, Nous proposons d'utiliser une machine programme qui montre le programme global OpenCL et d'écrire la machine noyau.

Nous utilisons une opération nécessitant deux variables d'entrée et une valeur de sortie. L'opération peut être définie ultérieurement en fonction de l'application choisie.

Le programme OpenCL est divisé en deux parties : Kernel et Programme globale.

8.2.1 Machine OpenCL_Kernel

Les invariants de la machine OpenCL_kernel sont :

```
inv1 : TaskNBRegister ∈ℕ
       inv2 : TaskStack ∈ Data_List
       inv3 : TaskProgramStack ∈ PGM_List
       inv4 : R1 ∈  Data
       inv5 : R2 ∈  Data
       inv6 : Op ∈   Operand
       inv7 : state ∈ℕ
       inv8 : RegADR1 ∈ ℙ(Data)

inv9 : RegADR2 ∈ ℙ(Data)
```

Nous avons utilisé PGM_List, Data_List et une variable nommée state pour changer l'état du noyau OpenCL. Qu'il soit en attente (Waiting), élu (Elected) ou en cours d'exécution (Running).

Evènement Waiting : lorsque la file d'attente OpenCL est vide, la TaskStack est initialisée à nil et son état est égal à 1.

```
Waiting ≙
       STATUS ordinary
       WHEN
           grd1 : TaskStack = Nil
```

```
        THEN
            act1 :  state := 1
        END
```

Evènement Running: les registres (opérandes) nécessaires à l'opération sont empilés dans les données TaskStack. Dans la liste des programmes, leurs adresses sont également ajoutées.

```
Running ≙
        STATUS ordinary
        WHEN
            grd1 : TaskStack ≠ Nil
        THEN
            act1 : Data_ListAppend(R1, TaskStack)
            act2 : Data_ListAppend(R2, TaskStack)
            act3 : Data_ListAppend(RegADR1, TaskProgramStack)
            act4 : Data_ListAppend(RegADR2, TaskProgramStack)
END
```

8.2.2 Machine OpenCL_Program

Le programme OpenCL est la partie du logiciel qui calcule les cœurs GPU et qui lance l'exécution sur le périphérique GPU. Pour assurer la bonne manipulation il y a plusieurs phases à utiliser.

Allocation de ressources : création d'instances de données (registres ou vecteurs,…)

Création de contexte OpenCL : il s'agit d'un événement créé avec GPU à l'aide de la plate-forme OpenCL et de l'identifiant de périphérique.

```
Context_creation ≙
        STATUS ordinary
        WHEN
            grd1 : CL_Platform  = 2
        grd2 : CL_DeviceType = CL_DEVICE_TYPE_GPU
        grd3 : CL_Device_id = 0
            grd4 : CL_OpenCL_Context = Nil
        THEN
            act1 :  CL_OpenCL_Context := 1
        END
```

Création de file d'attente de commandes OpenCL : assure la planification et l'exécution des noyaux parallèles à l'aide du contexte OpenCL et d'autres paramètres.

```
Command_queue_creation ≙
        STATUS ordinary
        WHEN
            grd1 : CL_OpenCL_Context = 1
            grd2 : CL_Device_id = 0
        THEN
        act1 :  CL_CommandQueue := 1
        END
```

Lancement du noyau(Kernel) : à l'aide des théories du noyau et des périphériques, le nombre d'exécutions du noyau peut être calculé en fonction du GPU.

```
Kernel_lauching ≙
        STATUS ordinary
        WHEN
            grd1 : GPUstate = 0
            grd2 : CL_OpenCL_Context = 1
            grd3 : CL_CommandQueue = 1
```

```
        THEN
            act1 :  GPUstate := 1
            act2 : NB_Kernel := Totalcoresnumber ÷Datasize
        act3 : CL_Program := 1
        END
```

Libération des ressources : Après avoir terminé l'exécution, les ressources utilisées sont libérées telles que les données et le périphérique GPU.

```
Data_Release ≙
        STATUS ordinary
        BEGIN
            act1 : Output := Nil
            act2 : Input := Nil
            act3 : GPUstate := 0
        END
```

8.3 Raffinement du modèle de base pour la détection de pics à partir d'images spectrales

On a choisi l'approche de détection de pics développée et présentée dans la section 7 comme application pour raffiner le modèle de base de machine Event B.

8.3.1 Raffinement de la machine OpenC_ Kernel

Le pic représentatif est défini par les pixels voisins, nous avons défini de nombreux types de pixels (Voir Section 7.3). Dans ce travail, nous avons développé noyau pour trouver un type de pic où tous les pixels voisins devraient être blancs sauf le pixel représentatif du fond. Seul l'événement en cours d'exécution dans la machine OpenCL Kernel sera modifié. Il est remplacé par la machine de recherche appelée searching .

Event-B ne prend pas en charge la définition matricielle, nous avons donc utilisé deux tableaux pour parcourir l'image verticalement et horizontalement.

```
    searching   ≙
STATUS
        Ordinary
    ANY
        pixel
    WHERE
        grd1   :   pixel ∈Arr(i)
        grd2   :   Arr(i) =1        //   black
        grd3   :   Arr(i-1)=0        //   white
        grd4   :   Arr(i+1)=0
        grd5   :   Arr(j+1)=0
    THEN
        act1   :   pos1≔i
        act2   :   pos2≔j
    END
```

8.3.2 Raffinement de la machine OpenCL_Program

Comme les données d'entrée sont une image, la machine raffinée aura besoin de deux autres variables : la largeur et la hauteur.

```
    inv12   :   height ∈ ℕ
```

```
inv13   :   width ∈ ℕ
```

Par conséquent, la structure de l'événement d'initialisation comprendra :

```
act8   :   height ≔ 754
act9   :   width ≔ 2000
act10   :   Datasize≔width*height
```

Dans la machine OpenCL program, le processus est le même pour toutes les applications : création de contexte, création de file d'attente de commandes, lancement du noyau et libération des ressources. Par conséquent, les autres événements de la machine programme OpenCL ne seront pas modifiés. En revanche, la machine Kernel sera améliorée. Le raffinement de la machine à noyau permet de redéfinir les instructions parallèles en fonction de l'application .

8.4 La validation de la spécification proposée par référence au langage OpenCL

Nous avons choisi OpenCL comme langage cible pour générer un pré-code formel à l'aide d'une application de détections de pics. Puisqu'on a déjà développé cette application en utilisant OpenCL précédemment, nous pouvons faire une comparaison avec le pré-code OpenCL généré. Les programmes OpenCL sont structurés dans le code à l'aide de la fonction d'application choisie ; les plus importantes fonctions sont présentées dans le tableau 8 ci-dessous qui illustre l'utilisation des fonctions de base d'OpenCL pour exécuter un programme. Dans la première spécification, en utilisant uniquement les raffinements Event-B, nous avons pu créer un système organisé en couches allant du modèle abstrait vers le modèle minimal du système qui est le noyau. Alternativement, nous avons trouvé un problème majeur pour définir et utiliser les fonctions OpenCL car cela nécessite de nombreux paramètres et de nombreux nouveaux types. Ces structures spécifiques n'ont pu être déclarées et acceptées dans la plateforme RODIN. Il y avait une différence positive significative de l'utilisation du plug-in Theory sur la spécification Event-B. Grâce au plug-in Theory permettant de définir de nouveaux types et de paramétrer les machines de manière similaire aux fonctions OpenCL, il nous a permis de générer un pré-code également proche du code OpenCL.

OpenCL functions	OpenCL Event-B specification	OpenCL Event-B specification using Theory plug-in
get_global_id		✓
Malloc	✓	✓
clGetPlatformIDs		✓
clGetDeviceIDs		✓
clCreateContext	✓	✓
clCreateCommandQueue		✓
clCreateProgramWithSource	✓	✓
clBuildProgram		
clCreateKernel		✓
clCreateBuffer		
clReleaseMemObject	✓	✓
clReleaseKernel		
clReleaseCommandQueue		✓

Tableau 10. Comparaison du code OpenCL avec les deux spécifications OpenCL Event-B

Pour résumer, la deuxième machine OpenCL avec le plug-in Theory est une structure acceptée faisant référence au code du langage OpenCL. Selon le tableau 8, les fonctions les plus importantes se trouvent dans la spécification utilisant Theory plug in d'OpenCL. De plus, la première machine OpenCL utilise trop de variables, invariants, événements et raffinements, ce qui rend les preuves et la validation plus difficiles. Contrairement à la deuxième machine OpenCL utilisant un nombre de variables limité, une structure correcte et des bonnes relations entre les variables.

Conclusion

Dans ce chapitre nous avons essayé de résumer nos travaux réalisés. Nous avons fait la synthèse des travaux connexes au couplage de spécification semi-formelle avec une spécification formelle. En plus nous avons étudiés des travaux qui font la génération de code exécutable parallèle à partir d'une spécification MARTE dans le domaine des applications parallèles sur GPU. De plus, nous avons présenté une spécification proposée avec MARTE et en Event B. Ensuite, Nous avons proposé une étape intermédiaire avant de générer le code consistant à raffiner une spécification formelle Event B du code exécutable de la spécification formelle Event B et on a appelé ce niveau de raffinage un pré-code. A partir de cette étape nous avons générer un pré-code en CUDA et OpenCL. Nous avons développé une approche de détection des pics sur GPU pour montrer l'efficacité de calcul du GPU en utilisant OpenCL. Puis, nous avons utilisé le plug-in Theory pour écrire un squelette du programme OpenCL en utilisant l'application de détection de pics.

Conclusion générale

Conclusion générale

Au cours de ce travail, la notion de système sur puce et l'utilisation de l'architecture multi-calcul pour paralléliser les algorithmes nécessitant un traitement parallèle. Des outils de spécification formelle et semi-formelle d'un système ont été mentionnés appuyés par quelques exemples de méthodes.

L'utilisation et la maîtrise d'UML/MARTE était un objectif pédagogique fixé à la thématique. UML/MARTE contient une multitude de profils permettant de spécifier un système embarqué. Event B est utilisé pour la validation et la preuve de la spécification UML/MARTE. L'approche d'IDM est employée pour la génération de code exécutable sur GPU suivant le couplage d'une spécification UML/MARTE bien détaillée et d'une spécification formelle Event B.

1 Contributions

En référence à la problématique, nous avons finalisé l'étape concernant la spécification semi-formelle en MARTE, la spécification formelle en Event B de l'architecture GPU et l'exécution d'une application parallèle sur un GPU. Pour ce faire, nous avons:

- ✓ modélisé l'architecture GPU en utilisant UML en plus de quelques profils de MARTE permettant de représenter les aspects du système projeté en montrant les composants matériels et l'allocation de ressources logicielles sur les composants matériels.
- ✓ traité l'ordonnancement des tâches sur GPU et la gestion du temps des exécutions des tâches sur GPU.
- ✓ couplé la spécification semi-formelle UML/MARTE et formelle Event B du GPU.
- ✓ affiné la spécification formelle pour écrire un pré-code exécutable CUDA/OpenCL.
- ✓ Utilisé une méthode pratique pour la génération de code à travers le plugin Theory.
- ✓ choisi une étude de cas pour montrer la validité de l'approche de détection de pics.

- ✓ optimisé le temps d'exécution de l'application de détection de pics à l'aide de l'implémentation parallèle sur GPU. L'application utilise deux algorithmes différents : la détection de pics à partir des images spectrales et à partir des fichiers CSV.
- ✓ proposé d'une classification par l'ACP afin de classifier les données exploitées dans la détection de pics pour pouvoir faciliter le processus de recherche de pics.

2 Perspectives

Les résultats théoriques et pratiques obtenus dans le cadre de cette thèse, permettent d'envisager un certain nombre de perspectives de recherche intéressantes. Quatre perspectives semblent pertinentes :(1) la génération automatique de code parallèle exécutable de la spécification semi-formelle en utilisant les techniques MDA . (2) La génération automatique du code exécutable CUDA/OpenCL à partir d'une spécification Event B d'une application parallèle sur une architecture GPU à l'aide des outils de Rodin. (3) L'optimisation de cas d'utilisation de détection de pics par des techniques d'ordonnancement pour rendre l'exécution plus rapide en réduisant le temps de transfert de données entre CPU et GPU. (4) le développement d'autres applications parallèles pour tester davantage l'approche de génération de code.

3 Publications de l'auteur

3.1 Publications internationales

Imane Zouaneb, Mostefa Belarbi, Abdallah Chouarfia (2014) 'Validating timing and scheduling MARTE profiles using event B, case study of a GPU architecture', in Models & Optimization and Mathematical Analysis *MOMA Journal*, No. 2, pp.30–43.

Imane Zouaneb, Mostefa Belarbi, Abdallah Chouarfia (2016) 'Multi approach for real-time systems specification: case study of GPU parallel systems', *International Journal of Big Data Intelligence IJBDI*, Vol. 3, No. 2, pp.122–141.

Imane Zouaneb, Mostefa Belarbi and Abdallah Chouarfia (2022) 'Converging Image Processing and Data Mining for Raman Spectroscopy Analysis', *International Journal of* Communication Networks and Distributed Systems, Vol.28,No.3,pp.287-311.

3.2 Conférences internationales

Imane Zouaneb, Mostefa Belarbi, Abdallah Chouarfia (2014) 'Formal approach for GPU architecture schedulability', in *The 2nd International Workshop on Mathematics and Computer Science*, University of Ibn Khaldoun, Tiaret.

Imane Zouaneb, Mostefa Belarbi, Abdallah Chouarfia (2017) 'A novel parallel approach to analyze spectroscopy Raman images using OpenCL', in *The 5th International Conference on Electrical Engineering – ICEE'2017*, Boumerdes, Algeria.

Imane Zouaneb, Mostefa Belarbi, Abdallah Chouarfia (2018) 'Based GPU approach to accelerate spectroscopy Raman spectrum processing', in *2018 5th International Conference on Electrical and Electronic Engineering (ICEEE)*, pp.360–365, Istanbul, Turkey.

Bibliographie

Bibliographie

[1] Sami BOUKHECHEM. Contribution à la mise en place d'une plateforme open-source MPSoC sous SystemC pour la Co-simulation d'architectures hétérogènes, Thèse de Doctorat, Université de Bourgogne, 2008.

[2] Youssef ATAT. Conception de haut niveau des MPSoCs à partir d'une spécification Simulink : Passerelle entre la conception au niveau Système et la génération d'architecture, Thèse de Doctorat, institut national polytechnique de Grenoble, 2007.

[3] Belkacemi DIHIA, Mapping d'applications parallèles sur des architectures embarquées multiprocesseurs à base de réseaux sur puce. Thèse de Doctorat. Université Mouloud Mammeri de Tizi Ouzou, 2020.

[4] Nicolas NGAN, Etude et conception d'un réseau sur puce dynamiquement adaptable pour la vision embarquée, Thèse de Doctorat, Université Paris-EST, 2011.

[5] Guérard HUBERT. Intégration d'un modèle de réseau sur puce dans un flôt de conception de niveau système, Support de cours, école polytechnique de Montréal, 2011.

[6] Julien DELORME. Méthodologie de modélisation et d'exploration d'architecture de réseaux sur puce appliquée aux télécommunications. Thèse de Doctorat. L'Institut National des Sciences Appliquées de Rennes, 2007.

[7] Matthieu BRIERE. Flot de conception hiérarchique d'un système hétérogène. Prototypage virtuel d'un réseau d'interconnexion optique intégré. Thèse de Doctorat. L'ECOLE CENTRALE DE LYON, 2005.

[8] Junyan TAN. Exploration d'architectures génériques sur FPGA pour des algorithmes d'imagerie multispectrale. Thèse de Doctorat. Université de Jean Monnet, 2012.

[9] Awange J.L., Paláncz B., Lewis R.H., Völgyesi L. Parallel Computations. In: Mathematical Geosciences. Springer, Cham, pp.559-596, 2018.

[10] LoïcRamu HELHO, Steve Destrebecq. Les cartes graphiques : http://transitkorichikhaled.files.wordpress.com/2013/02/cartesgraphique.pdf, 2004

[11] Maxime MARTELLI, Approche haut niveau pour l'accélération d'algorithmes sur des architectures hétérogènes CPU/GPU/FPGA, Application à la qualification des radars et des systèmes d'écoute électromagnétique, Université de Paris Saclay, 2019.

[12] Nicolas SOUCIES, Prédiction de performance d'algorithmes de traitement d'images sur différentes architectures hardwares, Signal and Image Processing. Université Pierre et Marie Curie - Paris VI, 2015.

[13] Yomna Ben Jmaa Chtourou, Implémentation temps réel des algorithmes de tri dans les applications de transports intelligents en se basant sur l'outil de synthèse haut niveau HLS, Université de Valenciennes et du Hainaut-Cambresis; École nationale d'ingénieurs de Sfax (Tunisie), 2019.

[14] Driss EN-NEJJARY. Spatial data Parallel Processing on GPGPU. Thèse de doctorat. Université Clermont Auvergne, 2021.

[15] Sylvain COLLANGe, Marc DAUMAS, David DEFOUR & David PARELLO. Étude comparée et simulation d'algorithmes de branchement pour le GPGPU. RenPar'19 / SympA'13 / CFSE'7, Toulouse, France, septembre 2009.

[16] Thomas IZARD. Opérateurs Arithmétiques Parallèles pour la Cryptographie Asymétrique. Cryptography and Security. Thèse de Doctorat. Université Montpellier II - Sciences et Techniques du Languedoc, 2011.

[17] Richard VUDUC and Jee CHOI. A brief history and introduction to GPGPU, In: X. Shi et al. (eds) Modern Accelerator Technologies for Geographic Information Science. Boston, MA: Springer US, pp.9-23, 2013.

[18] Wang Guohui and Xiong, Yingen and Yun Jayand Cavallaro Joseph R, "Computer vision accelerators for mobile systems based on OpenCL GPGPU co-processing," in Journal of Signal Processing Systems, vol.74, pp.283-299, 2014.

[19] Christophe Rubeck. Calcul hautes performances pour les formulations intégrales en électromagnétisme basses fréquences. Intégration, compression matricielle par ondelettes et résolution sur architecture GPGPU. Thèse de Doctorat. Université de Grenoble, 2012.

[20] Dongxu LIANG, Nong ZHANG, Hongyuan LIU, Daisuke FUKUDA, Haoyu RONG. Hybrid finite-discrete element simulator based on GPGPU-parallelized computation for modelling crack initiation and coalescence in sandy mudstone with prefabricated cross-flaws under uniaxial compression, Engineering Fracture Mechanics, vol.247, pp,1-24, 2021.

[21] Quentin Avril. Détection de Collision pour Environnements Large Echelle : Modèle Unifié et Adaptatif pour Architectures Multi-coeur et Multi-GPU. Graphics. Thèse de Doctorat. INSA de Rennes, 2011.

[22] Mehdi ROUAN SERIK. Implémentation des méthodes de recherche locale sur les architectures multi et many-cœurs : Application au problème d'affectation quadratique à 3 dimensions. Mémoire de magistère. Université d'Oran, 2013.

[23] Hocine Saadi, Metaheuristics on GPU graphics processor; Application to the Molecular Docking problem, Thèse de doctorat, University DJILLALI Liabès of Sidi Bel-Abbès, 2021.

[24] Peter N. Glaskowsky. NVIDIA's Fermi: The First Complete GPU Computing Architecture. Nvidia Corporation, http://www.nvidia.com/content/PDF/fermi_white_papers/P.Glaskowsky_NVIDIAFermi-TheFirstCompleteGPUComputingArchitecture.pdf, September 2009.

[25] Craig M. Wittenbrink, Emmett Kilgariff, ArjunPrabhu. FERMI GF100 GPU ARCHITECTURE. IEEE Micro, http://www.cs.cmu.edu/afs/cs/academic/class/15869-f11/www/readings/wittenbrink11_fermi.pdf. 2011.

[26] Jean-Michel Richer. Cours CUDA. Université d'Anger: http://www.info.univ-angers.fr/~richer/cuda_crs1.php, 2013.

[27] Wen-mei, W.Hwu. Programming Massively Parallel Processors: A Hands on Approach. Nvidia Corporation, 2010.

[28] Whitepaper NVIDIA's Next Generation CUDA™ Compute Architecture: Fermi™. Nvidia Corporation, 2009.

[29] AMD RADEON HD7950, HD7770 ET 7750 : GRAPHICS CORE NEXT POUR TOUS LES BUDGETS. Magazine PC UPDADTE, vol. 58, pp12-19, Avril 2012.

[30] Alexandre Chagoya-Garzon. Synthèse des communications dans un environnement de génération de logiciel embarqué pour des plateformes multi-tuiles hétérogènes. Thèse de Doctorat. Université de Grenoble, 2010.

[31] Alexandre Chariot. Quelques Applications de la Programmation des Processeurs Graphiques à la Simulation Neuronale et à la Vision par Ordinateur. Thèse de doctorat. Ecole des Ponts Certis Paris Tech, Novembre 2008.

[32] Antonio Wendell de O. Rodrigues, Frédéric Guyomarc'h, Jean-Luc Dekeyser. An MDE Approach for Automatic Code Generation from MARTE to OpenCL. INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE, 2011.

[33] Chitty, D.M. Faster GPU-based genetic programming using a two-dimensional stack. Soft Computing Journal, vol.21,n°14, 3859–3878, 2017.

[34] S. XIao et W. chun Feng, Inter-block GPU Communication via Fast Barrier Synchronisation. In IEEE International Symposium on Parallel & Distributed Processing IPDPS, pp.1-12, 2010.

[35] A. Betts, N. Chong, A. Donaldson, S. Qadeer et P.Thomson. GPUVerify: a verifier for GPU kernels. ACM SIGPLAN Notices, vol. 47, n10, pp. 113–132, octobre 2012.

[36] M. Bond. GPUDET: A Deterministic GPU Architecture. ACM SIGPLAN Notices, vol.48, n°4, pp. 1-12, avril 2013.

[37] W. Chun Feng et S. Xiao. To GPU Synchronize or not GPU Synchronize? In Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on, pp. 3801-3804, 2010.

[38] D. Defour, Impact des schedulers sur la prédictibilité dans les GPU, In Conférence en Parallélisme, Architecture et Système ComPAS"2014, Avril 2014.

[39] Lee, S., Seo, H., Kwon, H. et al. Hybrid approach of parallel implementation on CPU–GPU for high-speed ECDSA verification. Journal of Supercomput, v.75, no.8, pp.4329 – 4349, 2019.

[40] Poyraz Kocak, Y., Sevgen, S. Detecting and counting people using real-time directional algorithms implemented by compute unified device architecture. Neurocomputing journal, vol.248, 105-111, 2017.

[42] Amine Lajmi. Usine logicielle de composants de simulation de procédés CAPE-OPEN. Thèse de Doctorat. Université Pierre & Marie Curie, 2010.

[43] Olivier Sigaud. Introduction à la modélisation orientée objets avec UML. ENSTA ParisTech, http://wwwdfr.ensta.fr/Cours/docs/IN204/uml.pdf, 2004.

[44] Olivier Guibert. Analyse et Conception des Systèmes d'Information – Méthodes Objet : Le langage de modélisation objet UML. Support de cours. Université Bordeaux 1, 2010.

[45] Mohamed-Lamine Boukhanoufa. Adaptabilité et reconfiguration des systèmes temps-réel embarqués. Thèse de Doctorat. Université Paris Sud - Paris XI, 2012.

[46] Fateh Boutekkouk. Aide à la Conception des Systèmes Multiprocesseurs Mono-puce à partir de Spécification de haut niveau. Thèse de Doctorat. Université de Constantine, 2010.

[47] Jean-François Le Tallec. Extraction de modèles pour la conception de systèmes sur puce. Embedded Systems. Thèse de Doctorat. Université Nice Sophia Antipolis, 2012.

[48] Ali Koudri, Denis Aulagnier, Didier Vojtisek, Philippe Soulard, Christophe Moy, Joël Champeau, Jorgiano Vidal and Jean-Christophe Le Lann. Using MARTE in a Co-Design Methodology. Workshop MARTE, 2008.

[49] IMRAN RAFIQ QUADRI. Une méthodologie de conception dirigée par les modèles en MARTE pour cibler les systèmes sur puce basés sur FPGA dynamiquement reconfigurables. Thèse de Doctorat. Université des Siences et Technologies de LILLE-France, 2010.

[50] Rabie Ben Atitallah, Pierre Boulet, Arnaud Cuccuru, Jean-Luc Dekeyser, Antoine Honoré, et al. Gaspard2 UML profile documentation. Technical Report, RT-0342, 2007, pp.45.

[51] Imran Rafiq Quadri, Abdoulaye Gamatié, Pierre Boulet, Jean-Luc Dekeyser. Modeling of Configurations for Embedded System Implementations in MARTE. 1st workshop on Model Based Engineering for Embedded Systems Design - Design, Automation and Test in Europe (DATE 2010), Mar 2010, Dresden, Germany. 2010.

[52] Benabidallah Rymel, Approche de simulation de système de systèmes pour l'identification de comportements émergent, Thèse de Doctorat, Université des sciences et de la technologie Houari Boumedienne, 2021.

[53] Antonio PAOLILLO. *INRIA :* Model-driven engineering, Gaspard2 et optimisation multicritère. Université libre de Bruxelle. Rapport du séjour scientifique en centre de recherche. 2011.

[54] Guillet Sébastien. Modélisation et contrôle formel de la reconfiguration : Application aux systèmes embarques dynamiquement reconfigurables. Software Engineering. Thèse de Doctorat. Université de Bretagne Sud, 2012.

[55] Sana Cherif. Approche basée sur les modèles pour la conception des systèmes dynamiquement reconfigurables : de MARTE vers RecoMARTE. Modeling and Simulation. Thèse de Doctorat. Université des Sciences et Technologie de Lille - Lille I, 2013.

[56] Eric Jaeger. Remarques relatives à l'emploi des méthodes formelles (déductives) en sécurité des systèmes d'information. Secrétariat général de la défense nationale, 2008.

[57] Xavier Renault. Mise en œuvre de notations standardisées, formelles et semi-formelles dans un processus de développement de systèmes embarqués, temps-réel répartis. Thèse de Doctorat. Université Pierre et Marie Curie, 2009.

[58] Dairi Abdelkader, Spécification Formelle des systèmes d'information, Thèse de Doctorat. Université des Sciences et de la Technologie d'Oran Mohamed-Boudiaf USTOMB, 2021.

[59] Colette Johnen. Méthode Formelle –Java Modeling Language. Université de Bordeaux : http://www.labri.fr/perso/johnen/pdf/IUT-Bordeaux/MF/MF-cours.pdf, 2011.

[60] Yann Morère. Cours de réseau de Petri,: http://www.morere.eu/IMG/pdf/cours_petri2.pdf, Avril 2002 .

[61] G. Scorletti et G. Binet. Réseaux de Petri. UNIVERSITE de CAEN/BASSE–NORMANDIE, http://www.metz.supelec.fr/metz/personnel/vialle/course/CNAM-ACCOV-NFP103/extern-doc/RdP/Cours_Petri_etudiant_GS_2006.pdf, 2006.

[62] Emmanuel Fleury, Automates temporisés avec mises à jour. Thèse de Doctorat. ECOLE NORMALE SUPERIEURE DE CACHAN, 2002.

[63] Patricia BOUYER ,François LAROUSSINIE. Vérification par automates temporisés. In Nicolas Navet, editor, Systèmes temps-réel 1: techniques de description et de vérification , pages 121--150. Hermès, June 2006.

[64] Patrick BELLOT, Jean-Philippe COTTIN, Jean-François MONIN. Développement et validation de logiciels. Méthodes formelles. Centre National d'Études des Télécommunications, Lannio, 1996.

[65] J.R Abrial, "The B-book: Assigning programs to meanings", 1996.

[66] C. Métayer, J.-R. Abrial, L. Voisin. "Event-B Language", May 2005.

[67] Didier Bert, Marie-Laure Potet. Spécification en B. Support de Cours. Ecole des Jeunes Chercheurs en Programmation de Grenoble, 2007.

[68] Jens BENDISPOSTO · Michael LEUSCHEL · o. LIGOT · m. SAMIA. La validation de modèles Event-B avec le plug-in ProB pour RODIN. Revue des sciences et technologies de l'information. vol.27, no. 8, pp. 1065-1084, 2008.

[69] Linda Mohand Oussaid. Conception et vérification formelles des interfaces homme-machine multimodales : applications à la multi-modalité en sortie. Other. ISAE-ENSMA Ecole Nationale Supérieure de Mécanique et d'Aérotechique - Poitiers, 2014.

[70] Abderrahman Matoussi. Construction de spécifications formelles abstraites dirigée par les buts. Computers and Society Thèse de Doctorat. Université Paris-Est, 2011.

[71] Eman H. Alkhammash, Trustworthy smart city systems using refinement and Event-B Theories,Multimedia Tools and Applications vol.81,no.1, pp,615–636, 2021.

[72] Karmakar, Rahul. Formal Verification of a Medical Insurance System Prototype: The Event-B Modeling Approach. vol.17. pp.25-34.2022.

[73] Siala Badr. Décomposition formelle des spécifications centralisées Event-B: application aux systèmes distribués BIP,Performance et fiabilité. Thèse de Doctorat. Université Paul Sabatier- Toulouse III, 2017.

[74] Siala, B., Bodeveix, JP., Filali, M., Bhiri, M.T. An Event-B Development Process for the Distributed BIP Framework. In: Ait-Ameur, Y., Nakajima, S., Méry, D. (eds) Implicit and Explicit Semantics Integration in Proof-Based Developments of Discrete Systems. Springer, Singapore, pp 273-307, 2021.

[75] Thai Son Hoang. Proof Hints for Event-B. In Proceedings of DS-Event-B 2012: Workshop on the experience of and advances in developing dependable systems in Event-B, in conjunction with ICFEM 2012 - Kyoto, Japan, November 13, 2012.

[76] C. Métayer J.-R. Abrial, L. Voisin. Event B Language : RODIN "Rigorous Open Development Environment for Complex Systems". April 2005.

[77] Rupert Schlick and Thorsten Tarrach. Mutation-based test-case generation for Event-B, A new plug-in for Rodin. Rodin Workshop 2018. AIT Austrian Institute of Technology. 2018.

[78] Gerard O'Regan. Formal methods, Chapter in Mathematics in Computing. Springer-Verlag London, pp. 89-108, 2013.

[79] Stefan Hallerstede. On the purpose of Event-B proof obligations. Formal Aspects of Computing, Springer Verlag, 2009, vol.23,no.1, pp.133-150, 2009.

[80] R. Laleau and A. Mammar. An Overview of a Method and its support Tool for Generating B Specifications from UML Notations. In The 15st IEEE Int. Conf. on Automated Software Engineering, Grenoble (F), September 11-15, 2000.

[81] H. Ledang and J. Souquières. Formalizing UML Behavioral Diagrams with B. In the Tenth OOPSLA Workshop on Behavioral Semantics: Back to Basics, Tampa Bay, Florida (USA), pp.1-12, October 15, 2001.

[82] H. Ledang. Des cas d'utilisation à une spécification B. In Journées AFADL'2001 : Approches Formelles dans l'Assistance au Développement de Logiciels, Nancy (F), 11-13 juin, 2001.

[83] E. Meyer. Développements formels par objets: utilisation conjointe de B et d'UML. Thèse de Doctorat, LORIA – Université Nancy 2, Nancy (F), mars 2001.

[84] H. Ledang, J. Souquières et S. Charles. ArgoUML+B : un outil de transformation systématique de spécifications UML en B. LORIA - Université Nancy 2. 2003.

[85] Imen Sayar. Articulation entre activités formelles et activités semi-formelles dans le développement de logiciels. Génie logiciel. Thèse de Doctorat. Université de Lorraine, 2019.

[86] Antonio Wendell De Oliveira Rodrigues. UNE méthodologie pour le développement d'applications hautes performances sur des architectures gpgpu: application à la simulation des machines électriques. Thèse de Doctorat. Université des Sciences et Technologies de Lille, 2012.

[87] Wendell Rodrigues, Frédéric Guyomarc'h, Jean-Luc Dekeyser. Using ArrayOL to Identify Potentially Shareable Data in Thread Work-Groups of GPUs. Institut national de recherche en informatique et en automatique Lille, 2011.

[88] Antonio Wendell de Oliveira Rodrigues, Frédéric Guyomarch, Yvonnick Le Menach, Jean-Luc Dekeyser. Parallel Sparse Matrix Solver on the GPU Applied to Simulation of Electrical Machines. *Compumag 2009*, Nov 2009, Florianopolis, Brazil.

[89] Antonio Wendell de O. Rodrigues, Frédéric Guyomarc'h, Jean-Luc Dekeyser. Programming Massively Parallel Architectures using MARTE: a Case Study. Publiédans 2nd Workshop on Model Based Engineering for Embedded Systems Design (M-BED 2011), 2011.

[90] Jing Guo, Wendell Rodrigues & all. Harnessing the Power of GPUs without Losing Abstractions in SAC and ARRAYOL: A Comparative Study. Publiédans HIPS 2011, 16th International Workshop on High-Level Parallel Programming Models and Supportive Environments, 2011.

[91] Antonio Wendell de Oliveira Rodrigues, Frédéric Guyomarc'H, Jean-Luc Dekeyser, Yvonnick Le Menach. Automatic Multi-GPU Code Generation applied to Simulation of Electrical Machines. *Compumag 2011*, Jul 2011, Sydney, Australia.

[92] N. K. Singh, "EB2ALL: An automatic code generation tool," in *Using Event-B for Critical Device Software Systems*, Springer, pp. 105-141, 2013.

[93] Andreas Furst, Thai Son Hoang, David Basin, Krishnaji Desai, Naoto Sato, and Kunihiko Miyazaki. "Code Generation for Event-B", In Integrated Formal Methods book, Springer, pp.323-338, 2014.

[94] Cataño, N., Rivera, V. EventB2Java: A Code Generator for Event-B. In: Rayadurgam, S., Tkachuk, O. (eds) NASA Formal Methods. NFM 2016.

Lecture Notes in Computer Science, vol 9690. Springer, Cham, pp.166–171, 2016.

[95] Víctor Rivera, Néstor Cataño, Tim Wahls, Camilo Rueda. “Code generation for Event-B”, In International Journal on Software Tools for Technology Transfer, vol.19,pp.31-52, May 2015.

[96] Rivera, V., Lee, J., Mazzara, M. Mapping Event-B Machines into Eiffel Programming Language. In: Ciancarini, P., Mazzara, M., Messina, A., Sillitti, A., Succi, G. (eds) Proceedings of 6th International Conference in Software Engineering for Defence Applications. SEDA 2018. Advances in Intelligent Systems and Computing, vol.925. Springer, Cham.2020.

[97] Grall, A. Automatic Generation of DistAlgo Programs from Event-B Models. In: Raschke, A., Méry, D., Houdek, F. (eds) Rigorous State-Based Methods. ABZ 2020. Lecture Notes in Computer Science, vol.12071.pp.414-417.2020.

[98] Karmakar, R. A Framework for Component Mapping Between Event-B and Python. In: Hu, YC., Tiwari, S., Trivedi, M.C., Mishra, K.K. (eds) Ambient Communications and Computer Systems. Lecture Notes in Networks and Systems, Springer, Singapore, vol 356. pp.129–139, 2022.

[99] BOSTRÖM, Pontus, DEGERLUND, Fredrik, SERE, Kaisa, et al. Concurrent scheduling of Event-B models. Formal Aspects of Computing, vol.26,no.2,pp.166-182, 2011.

[100] Joey Coleman, Cliff Jones, Ian Oliver, Alexander Romanovsky, and Elena Troubitsyna, RODIN (Rigorous Open Development Environment for Complex Systems), Support de Cours, 2004.

[101] Issam Maamria. Towards a practically extensible Event-B methodology**.** Thèse de Doctorat. University of Southampton, UK 2013.

[102] Yamine AIT-Ait-Ameur & all. “Vérification et validation formelles de systèmes interactifs fondées sur la preuve : application aux systèmes multi-modaux,” IN Journal d’Interaction Personne-Système, Vol. 1, No. 1, Art. 3, Septembre 2010.

[103] Mendil, I., Aït-Ameur, Y., Singh, N.K., Méry, D., Palanque, P. Leveraging Event-B Theories for Handling Domain Knowledge in Design Models. In: Qin, S., Woodcock, J., Zhang, W. (eds) Dependable Software Engineering. Theories, Tools, and Applications. SETTA 2021. Lecture Notes in Computer Science, Springer, Cham, vol.13071. pp 40–58, 2021.

[104] Lorina Negreanu and Matei Popovici, Modeling and proof of event-driven interaction in multi agent systems in Event-B, 19th International Conference on Control Systems and Computer Science, Bucharest: Romania, pp.180 - 183, 2013.

[105] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, and Laurent Voisin. An open extensible tool environment for Event-B. In ICFEM 2006, LNCS, pp.588-605. Springer, 2006.

[106] Farhad Mehta. Supporting Proof in a Reactive Development Environment. International Conference on Software Engineering and Formal Methods, pp.103-112, 2007.

[107] Maamria, Issam, Butler, Michael, Edmunds, Andrew and Rezazadeh, Abdolbaghi. On an extensible rule-based prover for event-B. In *Proceedings of ABZ 010.* Springer. 3 pp .407, 2010.

[108] Butler, Michael & Maamria, Issam. . Practical Theory Extension in Event-B. Theories of Programming and Formal Methods. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, vol.8051,pp67-81.2013.

[109] Edmunds, Andrew & Butler, Michael & Maamria, Issam & Silva, Renato & Lovell, Chris.. Event-B Code Generation: Type Extension with Theories. 7316. 365-368. 2012.

[110] Abdelhamid, Hariche, Chouarfia, Abdallah, BELARBI, Mostefa. Embedded Systems Design Using Event-B Theories. International Journal of Computing and Digital Systems. vol.5.no(1),pp.173-187, 2016.

[111] George Socrates, "Infrared and raman characteristic group frequencies" New York: Wiley 2001.

[112] Jones, R.R., Hooper, D.C., Zhang, L. et al. Raman techniques: fundamentals and frontiers, *Nanoscale Research Letter*, vol. 14, no. 231.2019.

[113] Dieing, T. and Ibach, W. 'Software requirements and data analysis in confocal Raman microscopy', in Dieing T., Hollricher O., Toporski J. (Eds.): *Confocal Raman Microscopy*, Springer Series in Optical Sciences, vol. 158, Springer, Berlin, Heidelberg, 2018.

[114] Ho, C.S., Jean, N., Hogan, C.A. et al. Rapid identification of pathogenic bacteria using Raman spectroscopy and deep learning, *Nature Communications*, vol. 10, no. 1, pp.492:1–8, 2019.

[115] R.Salzer , G.Steiner , H.Mantsch , J.Mansfield ,N. E.Lewis, "Infrared and Raman imaging of biological and biomimetic samples, " Fresenius' Journal of Analytical Chemistry, vol.63, pp.712–726, 2000.

[116] J. M. P. Nascimento ,J. M. B.Dias, "Vertex component analysis: a fast algorithm to unmix hyperspectral data, " in Journal of IEEE Transactions on Geoscience and Remote Sensing, vol.43, pp.898– 910, 2005.

[117] Martin Hedegaard, Christian Matthaus , Soren Hassing , Christoph Krafft , Max Diem and Jurgen Popp, "Spectral unmixing and clustering algorithms

for assessment of single cells by raman microscopic imaging, ” in Theoretical Chemistry Accounts journal, vol.130, pp.1249–1260, 2011.

[118] Jianwei Qin , Kuanglin Chao , Moon S. Kim , Hoyoung Lee , Yankun Peng, “Development of a raman chemical imaging detection method for authenticating skim milk powder, ” in Journal of Food Measurement and Characterizatio, vol.8, pp.122–131,2014.

[119] Shiwei Sun , Xuetao Wang , Xin Gao , Lihui Ren , Xiaoquan Su , Dongbo Bu et al, “Condensing raman spectrum for single-cell phenotype analysis, ” in BMC Bioinformatics journal, vol.16, , pp. 1–7, 2015.

[120] T. Moumene ,E.H. Belarbi , B. Haddad and D. Villemin and O. Abbas et al, “Vibrational spectroscopic study of ionic liquids: Comparison between monocationic and dicationicimidazolium ionic liquids, ” in Journal of Molecular Structure, vol. 1065–1066, pp. 86 – 92, 2014.

[121] Petersen M, Yu Z, Lu X. Application of Raman Spectroscopic Methods in Food Safety: A Review. Biosensors (Basel). 2021.

[122] Hess C. New advances in using Raman spectroscopy for the characterization of catalysts and catalytic reactions. Chemical Society Reviews. vol.50,no.5,pp.3519-3564, 2021.

[123] Sieg Anke, Raman Spectroscopy, In: E. Berardesca et al. (eds.) Non Invasive Diagnostic Techniques in Clinical Dermatology, Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 217-223, 2014.

[124] Mosca, S., Conti, C., Stone, N. et al. Spatially offset Raman spectroscopy. Nature reviews methods primers vol.1, no.21 (2021).

[125] Geoffrey P.S. Smith ,Cushla M. McGoverin ,Sara J. Fraser and Keith C. Gordon, “Raman imaging of drug delivery systems,” in Advanced Drug Delivery Reviews journal, vol.89, pp.21–41,2015.

[126] Zibang Zhang , Xiao Ma , Jingang Zhong, “ Single-pixel imaging by means of Fourier spectrum acquisition, ” in Nature communications journal. Macmillan Publishers,vol.5, pp.1-6, 2015.

[127] Santos J. A., and Neves M. M. A local maximum likelihood estimator for Poisson regression, Metrica Journal , 68(3), pp. 257-270, 2008.

[128] Obuchowski J., Wyłomańska A., and Zimroz R. The local maxima method for enhancement of time–frequency map’, Advances in Condition Monitoring of Machinery in Non-Stationary Operations, Proceedings of the Third International Conference Condition Monitoring of Machinery in Non-Stationary Operations CMMNO 2013, Lecture Notes in Mechanical Engineering, 9, pp.325-334, 2013.

[129] Obuchowski J., Wyłomańska A., and Zimroz R. The local maxima method for enhancement of time–frequency map and its application to

local damage detection in rotating machines', Mechanical Systems and Signal Processing, vol.46,no.2, pp.389 -405, 2014.

[130] Kim D., and Kim S. K. Comparing patterns of component loadings: Principal Component Analysis (PCA) versus Independent Component Analysis (ICA) in analyzing multivariate non-normal data. Behavior research methods, 2012, vol44, no.4, pp. 1239-1243, 2012.

[131] Zaoui, T. Belarbi, M. and Debdab, M.(2016) 'Based Principal Component Analysis of Associated Ionic Liquid Biosensors', Procedia Computer Science, 83, pp.1038-1043.

[132] "GeForce GT 630 Description," referring Online: http://www.geforce.com/hardware/notebook-gpus/geforce-gt-630m/description . 2012 (Accessed June 2022).

[133] "Bio-Rad KnowItAll Software Analytical Edition": www.bio-rad.com/webroot/web/pdf/spectroscopy/global/english/literature/docs/95376-Bio-Rad_KnowItAll_Software_Analytical_Edition_Brochure.pdf. 2017 (Accessed June 2022)

[134] "KnowItAll- The World's Largest Spectral Database": http://www.bio-rad.com/en-cn/product/knowitall-u-chemistry-spectra-database. 2016. (Accessed June 2022)

[135] Spectral database Searching": www.horiba.com/scientific/products/raman-spectroscopy/software/functionality/spectral-database-searching/ 2017 (Accessed June 2022)

[136] Araujo C F., Nolasco M M., Ribeiro Antonio M.P., Ribeiro P. Identification of microplastics using Raman spectroscopy: Latest developments and future prospects, Water Research Journal, Vol. 142, pp. 426-440, 2018.

[137] Rojalin T., Koster H J., Liu J., Mizenko R R, Tran D, Wachsmann-Hogiu S., and Carney R. Hybrid Nanoplasmonic Porous Biomaterial Scaffold for Liquid Biopsy Diagnostics Using Extracellular Vesicles, *ACS Sensors*,Vol.5, No.9,pp. 2820-2833, 2020.

[138] Colniță A., Dina NE., Leopold N., Vodnar DC., Bogdan D., Porav SA., David L. Characterization and Discrimination of Gram-Positive Bacteria Using Raman Spectroscopy with the Aid of Principal Component Analysis', Nanomaterials (Basel), vol.7, No.9, pp.248, 2017.

Annexe 1

Annexe 1 : Classes générées par la méthode ACP

Order	Material	Class	Sub-Class	Correlated	Strongly correlated	Observations
88	**L-Proline**	1	1.1			S.C
62	**Magnesium hydrogen phosphate,trihydrate**	1	1.1			No comments
75	**L-Arginine HCL**	1	1.1			S.C
77	**L-Aspartic acid**	1	1.1	X		SC
83	**L-Histidine HCL H2O**	1	1.1	X		S.C
92	**L-Threonine**	1	1.1		X	S.C / F.C
82	**L-Hydroxyproline**	1	1.1		X	S.C/ F.C
90	**L-Serine**	1	1.1			S.C
48	**L-Tyrosine**	1	1.1			S.C
80	**L-Glutamine**	1	1.1	X		SC
81	**Glycine**	1	1.1	X		SC
79	**L-Glutamic acid**	1	1.1	X		SC
64	**Octocalium phosphate,pentahydrate**	1	1.2			SC
110	**Calcium magnesium phosphate hydrat**	1	1.2			SC
95	**Starch**	1	1.2			-
9	**Amoxicillin,trihydrate**	1	1.2			No comments
32	**Cholesterol**	1	1.2			-
34	**Indinavir,monohydrate**	1	1.2			-
40	**Gypsum**	1	1.2			No comments
62	Magnesium hydrogen phosphate,trihydrate	2	2.1			-
44	Hypoxanthine	2	2.2			SC
5	Allopurinol	2	2.2		X	SC
41	Hydroxylapatite	2	2.2			No comments
101	Uric acid ,dihydrat	2	2.2	X		SC/FC
94	Sodium urate monohydrate	2	2.2			SC
6	Allopurinol	2	2.2			SC
54	N-Acetylsulfadiazine	2	2.2			-
35	Cystine	2	2.2			-
109	Weddellite	2	2.2			-
108	Weddellite	2	2.2			-
91	L-Thryptophan	2	2.2		X	SC
55	N-Acetylsulfadiazine	2	2.2			-
2	Dihdroxyadenine	2	2.2		X	SC
74	L-Asparagine	2	2.2			No comments
97	Struvite	2	2.2			-
99	Talc	2	2.2			No comments
96	Struvite	2	2.2			-

102	Uric acid ,dihydrat anhydrous	2	2.2		X	No comments
11	Aragonite	2	2.2			No comments
105	Calcium carbonate	2	2.2	X		-/ FC
25	Calcite	2	2.2			No comments
103	Calcium carbonate	2	2.2			-
60	Nalidixic acid	2	2.2			-
10	**Amoxicillin,trihydrate**	3	3.1			No comments
100	**Uric acid ,dihydrat**	3	3.1			SC
65	**Octocalium phosphate,pentahydrate**	3	3.1	X		SC
71	**Polyglactin**	3	3.1			No comments
41	**Hydroxylapatite**	3	3.1	X		
24	**Calcium hydrogen phosphate**	3	3.1	X		SC
63	**Magnesium hydrogen phosphate,trihydrate**	3	3.1			SC
15	**Brushite**	3	3.1	X		
37	**Glafenic acide**	3	3.1			No comments
21	**Dicalcium pyrophosphate**	3	3.1			SC
23	**Calcium hydrogen phosphate**	3	3.1	X		SC
19	**Uric acid salt, hydrate**	3	3.1			SC
28	**Carbonate hydroxylapatite**	3	3.1	X		-
29	**Ceftriaxone,calcium salt,hydrate**	3	3.1			-
27	**Carbonate hydroxylapatite**	3	3.1	X		-
26	**Carbonate hydroxylapatite**	3	3.1	X		-
59	**Sodium potassium urate**	3	3.1			SC
104	**Calcium carbonate**	3	3.1			SC
4	**Amorphous carbonated calcium**	3	3.1			-
3	**Amorphous carbonated calcium**	3	3.1	X		-
16	**Brushite**	3	3.1			-
22	**Tricalcium phosphate**	3	3.1			SC
8	**Ammonium urate**	3	3.1			SC
53	**N-Acetylsulfadiazine**	3	3.1			No comments
7	**Ammonium urate**	3	3.1			SC
13	1,3,6,7-Tetrameth	4	4.1			-
42	Hydroxylapatite	4	4.1			-
12	Aragonite	4	4.1			-
31	Cholesterol	4	4.1		X	SC
39	Glycocholic acid,sodium salt	4	4.1			SC
30	Cholesterol	4	4.1			SC
50	Triamterene	4	4.1			SC
86	L-Leucine	4	4.1			SC
56	2-[(N-Acetylsufanily)aminolpyrimidine	4	4.1			SC

57	N-Acetylsufamenthoxazole HCL	4	4.1	X		SC
33	Indinavir,monohydrate	4	4.1		X	No comments
43	Hydroxylapatite	4	4.1			-
66	Orotic acid,anhydrous	4	4.1			SC
111	Xanthine	4	4.1			SC
58	N-Acetylsufamenthoxazole HCL	4	4.1	X		SC
67	Dihdroxypyrazolo[3,4)]yrimidine	4	4.1	X		SC
72	**Polyglactin**	4	4.2			SC
98	**2-(Sulfanilylamino)pyrimidine**	4	4.2			No comments
17	**Calcium citrate,tribasic,tetrahydrate**	4	4.2			SC
107	**Paraffin**	4	4.2			SC
49	**Stearic acid,magnesium salt**	4	4.2			SC
106	**Paraffin**	4	4.2			SC
52	**Mucopolysaccarides**	4	4.2			-

Tableau 11. Les classes générées

SC : Structural comparison
FC : Functional Comparison,
-: Lack of structure information.

ملخص

نظام الشريحة (SoC) هو نظام إلكتروني متكامل مدمج على شريحة واحدة. يمكن أن يتكون من وحدة حاسوبية واحدة أو أكثر بما في ذلك وحدة معالجة الرسومات (GPU). تعتبر وحدة معالجة الرسومات بمثابة معالج مساعد يسمح بموازاة تنفيذ المهام في نظام الشريحة وتفريغ وحدة المعالجة المركزية. مرحلة التصميم و الوصف الدقيق للنظام المدمج المراد تنفيذه على شريحة هي أصعب مرحلة في تطوير الأنظمة المدمجة على شريحة. هناك العديد من الطرق لتصميم هذا النوع من الأنظمة. نقترح الجمع بين طريقتين: طريقة تصميم و نمذجة شبه رياضية باستعمال UML و البروفايل MARTE و طريقة تصميم و نمذجة رياضية باستعمال Event B و تعتبر طريقة آمنة وصالحة ومثبتة بواسطة أداة ذات صلة تسمى RODIN . قمنا بإنشاء كود GPU (CUDA و OpenCL) من خلال التحسينات المتتالية للنمذجة الرياضية للنظام SOC بواسطة Event B. كما قمنا باقتراح أنواع بيانات جديدة باستعمال Theory. لقد قمنا بتطوير بعض التطبيقات المتوازية على وحدة معالجة الرسومات مثل: جمع الجداول، الكشف عن قمم الرسومات البيانية لجهاز Raman وتصنيف البيانات بواسطة ACP لتحسين وقت التنفيذ وإجراء مقارنة مع الكود الذي تم إنشاؤه تلقائيًا.

كلمات مفتاحية: نظام الشريحة، نظام مدمج ذو وقت فعلي، وحدة معالجة الرسومات، طريقة UML، البروفايل MARTE، الطريقة Event B، إنشاء الكود، الكشف عن القمم، طريقة التصنيف ACP.

Résumé

Un système sur Puce (SoC) est un système électronique complet intégré sur une puce. Il peut être constitué d'une ou plusieurs unités de calcul dont le GPU. Le GPU est considéré comme un coprocesseur permettant de paralléliser l'exécution des tâches sur le SoC et de décharger le CPU. La modélisation et la spécification d'un Système sur Puce embarqué n'est pas une tâche facile à faire. Il existe de nombreuses méthodes pour modéliser ce type d'applications. Nous proposons de faire le couplage entre deux méthodes : une spécification semi-formelle par UML et le profil MARTE chargé de la modélisation des systèmes embarqués temps réel, et une spécification formelle en Event B sûre, valide et prouvée par un outil pertinent appelé RODIN. Nous avons généré un code GPU (CUDA et OpenCL) à travers les raffinements successifs de la spécification formelle du SoC en Event B et nous avons aussi proposé de nouveaux types en utilisant le Theory plug-in. Nous avons développé quelques applications parallèles sur GPU tels que l'addition vectoriel, la détection de pics de spectre Raman et la classification par ACP pour optimiser le temps d'exécution et faire une comparaison avec le code généré automatiquement.

Mots clés : Système sur Puce, Système Embarqué Temps Réel, GPU, UML, MARTE, Event B, Theory Event B, Génération de code, Détection de pics, ACP.

Abstract

A System on a Chip (SoC) is a fully integrated electronic system on a chip. It could be composed of one or more computing units, including the GPU. The GPU is regarded as a coprocessor, allowing task execution on the SoC to be parallelized and the CPU to be offloaded. It is not easy to model and specify an embedded System-on-Chip. There are many methods to model this type of applications. We propose to couple two methods together: a semi-formal specification by UML and the MARTE profile in charge of modeling real-time embedded systems, and a formal specification in Event B that is safe, valid, and proved by a relevant tool known as RODIN. We generated GPU code (CUDA and OpenCL) through successive refinements of the formal SoC specification in Event B, and we proposed new types using the Theory plug-in. We have developed some parallel GPU applications, such as vector addition, Raman spectrum peak detection, and PCA classification, to optimize execution time and compare to automatically generated code.

Keywords: System on Chip, Embedded Real Time System, GPU, UML, MARTE, Event B, Event B Theory, Code generation, Peak detection, PCA.