

## Travail D'Étude et de Recherche

Evaluation d'une architecture SoC (ARM-FPGA) pour des applications de traitement d'images

Rapport d'étude bibliographique

**DOU Yuhan**  
**GHAOUI Mohamed Anis**  
**ZHANG Boyang**

Encadré par : **ELOUARDI Abdelhafid**

Master Électronique, Énergie Électrique et Automatique - Orsay-Cachan  
Master Électronique, Énergie Électrique et Automatique - André Ampère  
Année : 2018-2019



Comprendre le monde,  
construire l'avenir



université  
PARIS-SACLAY

# Table des matières

Introduction générale . . . . .	2
<b>1 Architecture Hétérogène / Asymétrique Multi-cœurs</b>	<b>2</b>
1.1 Introduction . . . . .	3
1.2 Définitions . . . . .	3
1.3 Illustration des différentes architectures . . . . .	3
1.4 Types d'hétérogénéité . . . . .	4
1.4.1 Cœurs identiques avec différentes configurations . . . . .	4
1.4.2 Cœurs à capacités architecturales différentes . . . . .	4
1.4.3 Différence de modèles d'exécution . . . . .	5
1.5 Avantages et Inconvénients . . . . .	5
1.6 Critères d'évaluation . . . . .	6
1.6.1 Nombre d'éléments logiques . . . . .	6
1.6.2 Multiplieurs . . . . .	6
1.6.3 Pthread . . . . .	6
1.6.4 Nombre de cycles . . . . .	6
1.6.5 Fréquence maximale d'horloge . . . . .	6
1.6.6 Wall-clock time . . . . .	6
1.7 Conclusion . . . . .	7
<b>2 Étude de la parallélisation des calculs pour architecture hétérogène</b>	<b>7</b>
2.1 Introduction . . . . .	8
2.1.1 Définitions . . . . .	8
2.1.2 Critères de parallélisation . . . . .	8
2.1.3 Avantages, Inconvénients et Perspectives . . . . .	9
2.1.4 Domaine d'application . . . . .	10
2.1.5 Perspectives . . . . .	10
2.2 Classification de calcul parallèle . . . . .	10
2.2.1 Architecture de John von Neumann . . . . .	10
2.2.2 Classification de Flynn . . . . .	11
2.3 Architecture de la mémoire de calcul parallèle . . . . .	12
2.3.1 Mémoire partagée[13, p. 713] . . . . .	12
2.3.2 Mémoire distribuée . . . . .	13
2.3.3 Mémoire hybride . . . . .	13
2.4 Modèle de calcul parallèle . . . . .	13
2.5 Conception de programme parallèle[16] . . . . .	14
2.5.1 Parallélisation automatique ou manuelle[17] . . . . .	14
2.5.2 Compréhension des problèmes et des procédures . . . . .	14
2.5.3 Partitionnement . . . . .	14

2.5.4	Communication . . . . .	15
2.5.5	Synchronisation . . . . .	15
2.5.6	Dépendance de données . . . . .	15
2.5.7	Équilibrage de charge . . . . .	15
2.5.8	Granularité . . . . .	15
2.5.9	Entrée et sortie (I/O) . . . . .	16
2.5.10	Débogage . . . . .	16
2.5.11	Analyse et optimisation des performances . . . . .	16
2.6	Conclusion . . . . .	16
<b>3</b>	<b>“CPU + FPGA” pour le traitement d’image</b>	<b>16</b>
3.1	Introduction . . . . .	17
3.2	Co-traitement et Traitement en ligne[18] . . . . .	17
3.2.1	Co-traitement . . . . .	17
3.2.2	Traitement en ligne . . . . .	18
3.3	Algorithmes de traitement d’image et vision <i>CPU</i> et <i>FPGA</i> . . . . .	18
3.4	Conclusion . . . . .	19
<b>4</b>	<b>Logiciels pour architectures hétérogènes</b>	<b>19</b>
4.1	Introduction . . . . .	20
4.2	OpenCl . . . . .	20
4.2.1	Acquisition . . . . .	20
4.2.2	Programmation . . . . .	21
4.2.3	Les dispositifs . . . . .	24
4.2.4	Le contexte avec plateforme et dispositifs . . . . .	26
4.2.5	Implémentation du noyau . . . . .	27
4.2.6	Création du programme avec le fichier noyau et le contexte . . . . .	28
4.2.7	Création du noyau avec programme construit . . . . .	29
4.2.8	Insertion des données dans le noyau . . . . .	29
4.2.9	Exécution des calculs avec les dispositifs ciblés . . . . .	29
4.2.10	Exportation des résultats . . . . .	30
4.3	Vivado-HLS . . . . .	30
4.3.1	Introduction . . . . .	30
4.3.2	Fichiers source et examen . . . . .	30
4.3.3	Génération du fichier VHDL . . . . .	33
4.4	Conclusion . . . . .	35
	Introduction générale . . . . .	36

# Table des figures

1.1	Exemple d'Architecture Hétérogène Multi-Coeurs [1]	3
1.2	Exemple d'Architecture Homogène Multi-Coeurs [1]	4
2.1	Parallélisation d'un problème pour plusieurs processeurs	8
2.2	Accélération en fonction	9
2.3	Architecture de <i>Von Neumann</i>	11
3.1	Schéma représentatif du co-traitement entre <i>FPGA</i> et <i>CPU</i>	17
3.2	Schéma représentatif du traitement en ligne entre <i>FPGA</i> et <i>CPU</i>	18
4.1	Schéma de coordination de l'hôte pour les tâches et données	20
4.2	liste de fichiers pré-requis	21
4.3	Organigramme de déroulement de la programmation <i>OpenCL</i>	22
4.4	Prototypes des fonctions <i>clGetPlatformIDs</i> et <i>clGetPlatformInfo</i>	22
4.5	Appel de la fonction <i>clGetPlatformIDs</i>	23
4.6	Appel de la fonction <i>clGetPlatformInfo</i>	23
4.7	Implémentation de la fonction <i>displayinfo</i>	23
4.8	Implémentation de la fonction <i>DisplayPlatformInfo</i>	24
4.9	Affichage des plateformes disponibles	24
4.10	Prototypes des fonctions <i>clGetDeviceIDs</i> et <i>clGetDeviceInfo</i>	24
4.11	Implémentation de la fonction <i>DisplayDevice</i>	25
4.12	affichage des dispositifs sur différentes plateformes	25
4.13	Déclaration et implémentation de la fonction <i>clCreateContextFromType</i>	26
4.14	Affichage des dispositifs dans le contexte choisi	27
4.15	Implémentation d'une boucle <i>for</i>	27
4.16	Représentation d'une opération sur deux tableaux de données	28
4.17	Implémentation du noyau pour le calcul	28
4.18	prototype des fonctions de construction de programmes	28
4.19	Détail de l'implémentation de la construction du programme	28
4.20	Prototype et implémentation de la fonction de création de noyaux	29
4.21	Prototype et implémentation de la fonction de création de noyaux	29
4.22	Choix du dispositif de calcul	29
4.23	Lancement du calculs	30
4.24	Récupération des résultats sur le buffer de sortie	30
4.25	Affichage du résultat	30
4.26	Interface de projet Vivado-HLS	31
4.27	Implémentation du mux21 en C	31
4.28	Définitions de l'entête du mux21	31
4.29	Ajout de source au projet	31

4.30	Implémentation du fichier d'examen . . . . .	32
4.31	Résultat d'une simulation en C . . . . .	32
4.32	Génération du fichier VHDL . . . . .	33
4.33	Présentation de l'arborescence de la solution 1 . . . . .	33
4.34	Fichier VHDL généré . . . . .	34
4.35	Partie fonctionnel du mux21 en VHDL . . . . .	34
4.36	Interface des information complémentaires . . . . .	35

---

## Introduction générale

Durant les dernières décennies, le besoin d'effectuer des calculs toujours plus importants n'a cessé d'augmenter à un tel point que les améliorations des performances des systèmes à  $\mu$ Processeur, dit à *architectures classiques*, n'ont pu suivre. Donc ce genre d'architecture devient obsolète.

On a alors d'abord procédé à la parallélisation des calculs sur plusieurs cœurs du même type. Ce type d'architecture est nommée *Architecture Homogène/Symétrique Multi-cœurs*. On alors un seul circuit intégré qui porte en lui plusieurs cœurs du même type.

Ce type d'architecture a grandement améliorer les performances pour plusieurs applications. De nos jours, la quasi-totalité des ordinateurs et téléphone contient une architecture multi-cœurs. Toutefois, certains systèmes temps-réel, de sécurités et cyber-physiques requièrent une autre répartition asymétrique des ressources en en nombres et type de cœurs alloués. On parle alors d'*Architecture Hétérogène/Asymétrique Multi-cœurs*.

Ces architectures consistent en une puce qui porte tout le système. On parle alors de *System on Chip*. C'est donc un seul circuit intégré qui possède plusieurs cœurs de type et de répartition différentes. On trouve alors des *SoCs* conçues pour des applications plus spécifiques car ce genre d'architecture peut intégrer un ou plusieurs cœurs *CPUs*, *GPUs*, *DSPs* ou *FPGAs* ou plusieurs combinaisons de ces cœurs la.

Dans ce rapport, nous allons premièrement voir ce qu'est une architecture hétérogène en comparaison à une architecture homogène et énumérer les différents types d'association de noyaux possibles, leurs avantages et inconvénients, les critères d'évaluation de ces architectures et les problèmes rencontrés par l'industrie.

Dans un second temps, on étudiera les algorithmes pour architectures aux calculs parallèles. On citera notamment les critères de parallélisation, les classification de ces algorithmes, leur répartition suivant les architectures connues. Aussi on abordera quelques concepts de mémoire partagée/distribuée. Pour finir, on s'intéressera à la méthodologie de conception d'un algorithme parallèle/parallélisable. Le but ici n'est pas de concevoir un tel algorithme mais de comprendre l'essence même de cette vaste discipline afin d'entrevoir son application aux architectures hétérogènes.

Puis, On prendra comme exemple la réalisation du traitement d'image grâce à un *SoC* combinant un *CPU* et un *FPGA*. Le tout en montrant la structure nécessaire à l'implémentation d'un algorithme parallèle approprié.

Enfin, On s'intéressera aux outils permettant l'interaction entre les architectures hétérogènes et les algorithmes de traitement parallèle. c-à-d, le lien entre la partie virtuelle de notre discipline et sa partie matérielle. *OpenCL*, le premier outil présenté, sert à implémenter des algorithmes parallèles sur des architectures dédiées. *Vivado-HLS*, le deuxième outil représenté, a pour fonction de transcompilateur, c-à-d générer un code *VHDL* à partir d'un Code *C/C++*. Bien-sûr, vu le nombre élevé d'architectures possibles, les différentes normes des constructeurs, la complexité des problèmes lors de la conception des algorithmes parallèles et les architectures qui leurs sont associé, on se restreindra à des exemples assez simples à faire de servir de guide pour de futures travaux sur le sujet.

Pour finir, on clôturera par une conclusion et des perspectives.

# Chapitre 1

## Architecture Hétérogène / Asymétrique Multi-cœurs

## 1.1 Introduction

Une architecture est dite multi-cœurs si elle implémente sur une même puce plusieurs cœurs fonctionnant en parallèle. Si les cœurs implémentés sont différents par leur capacité, nature, vitesse, spécificité de tâches ou même paramétrage alors l'architecture est dite hétérogène.

## 1.2 Définitions

Les architectures hétérogènes utilisent leurs différents cœurs de manière à optimiser le calcul et réduire la dissipation en puissance en associant une tâche à exécuter à un cœur conçu spécifiquement pour traiter de manière optimale ce tâche. Ces architectures sont utilisées dans la totalité des téléphones modernes et autres systèmes embarqués où les contraintes liées à l'énergie et le temps réel sont crucial.

Ce qui est n'est pas le cas des architectures homogènes/symétriques multi-cœur (*HAS*) qui sont utilisés sur les ordinateurs où tous les cœurs sont identiques conception et en fonction, donc il n'y a aucune différence en dissipation de puissance entre l'utilisation d'un cœur par rapport à un autre.

Au final, la différence n'est pas dans le code à exécuter mais plutôt dans la manière dont il sera exécuter.

## 1.3 Illustration des différentes architectures

Les figures 1.1 et 1.2 décrivent 2 architectures multi-cœurs, la première étant hétérogène, l'autre homogène. On suppose aussi que l'on a 14 applications à chaque fois traiter par 4 cœurs.

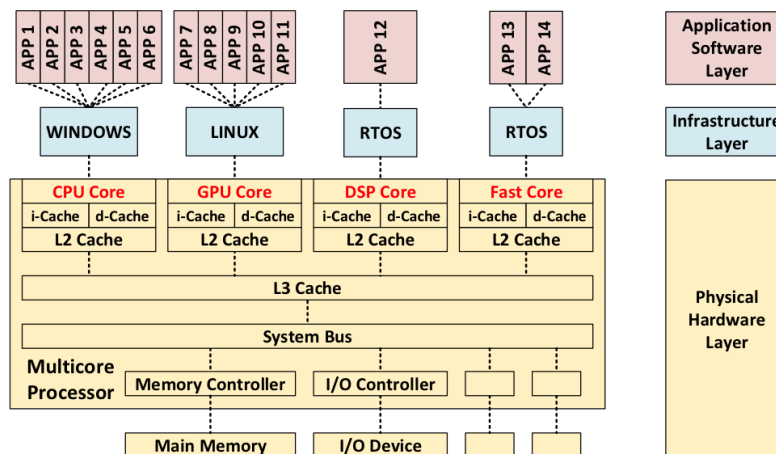


FIGURE 1.1 – Exemple d'Architecture Hétérogène Multi-Coeurs [1]

Dans ce cas de figure, les applications sont distribuées non-uniformément sur les 4 cœurs de types différents : *CPU*, *GPU*, *DSP* et *Fast*. Il est donc possible d'affecter à un cœur les applications où il est le plus performant.

Par exemple, le *GPU* s'occupe du traitement d'image car son architecture propre vise à traiter de la meilleure manière ce format de données contrairement à un *CPU* qui lui est



meilleur en opérations élémentaires. C'est ce biais des unités de traitement vers certains formats de données ou certaines instructions qui est l'élément clef de cette architecture.

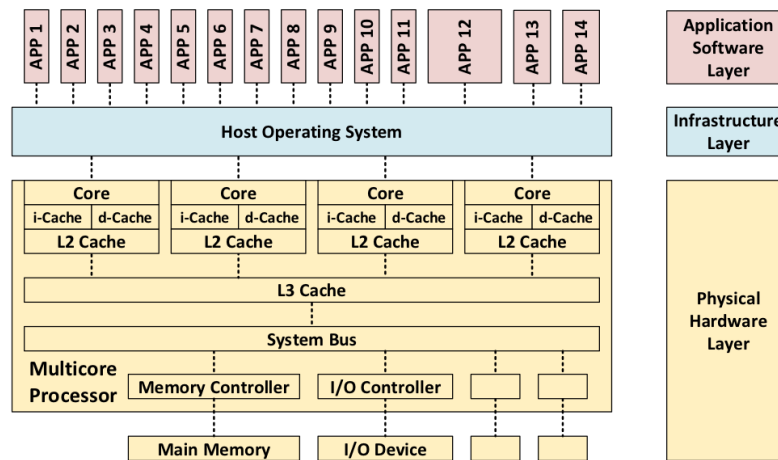


FIGURE 1.2 – Exemple d'Architecture Homogène Multi-Coeurs [1]

On a ici une variante où tout les coeurs sont exactement du même type et traite toutes les applications de la même manière. Si les 14 applications sont identiques ou quasi-identiques, cette architecture est beau plus favorable dans ce cas.

En effet, dans un système, si les données/instructions sont de formats/types variés, une architecture hétérogène est préférable. Sinon, on privilégiera une architecture homogène qui possède des traitements plus généralisés.

## 1.4 Types d'hétérogénéité

En pratique, il existe plusieurs types d'hétérogénéité qui se différencie par la configuration des coeurs ou *noyaux*. On note essentiellement 3 types :

### 1.4.1 Coeurs identiques avec différentes configurations

Même si cela peut paraître contre-intuitif, on peut avoir une architecture qui implémente les mêmes types de noyaux, voire des noyaux ayant les mêmes caractéristiques, mais ils fonctionneraient de manières différentes. Ceci est dû au fait que chaque noyau possède sa propre dynamique en tension et échelle fréquentielle (*DVFS*)[2]. Un cœur travaillant plus qu'un autre va naturellement chauffé et donc sa fréquence s'abaissera et donc ralentira et dans ce ca l'architecture est hétérogène, même noyaux mais fonctionnement différent.

### 1.4.2 Coeurs à capacités architecturales différentes

Dans ce type, un processeur pourrait contenir plusieurs coeurs "*simples*" et d'autres "*gros*" Coeurs. Ces coeurs différent de part leurs technologies d'*hyperthreading*, largeur super-scalaire, arithmétique des vecteurs, etc. On n'abordera pas ces notions car elles sont hors de portée de notre étude.

Dans ces deux types d'hétérogénéité, les noyaux *semblent* exécuter les instructions séquentiellement même si cela peut ne par être le cas dans leurs couches internes grâce à l'usage de *pipeline* ou autre par exemple.

### 1.4.3 Différence de modèles d'exécution

Ce troisième type va combiné plusieurs noyaux de natures différentes. Cette différence réside dans le modèle de traitement de l'instruction. Pour simple comparaison :

Les *GPUs* dont l'usage est très répandu dans la génération de graphisme, l'entraînement des IAs, animation, modélisation et les calculs numériques importants, utilisent un modèle *SMID* (*Single Instruction Multiple Data*) voir 2.2.2 p 11. i.e. Ce type de noyau montre une performance élevée pour l'exécution d'instructions indépendantes mais similaires pour une quantité extrêmement importante.

Si par exemple l'on voulait multiplier une matrice très large par un scalaire, un *CPU* pourrait effectuer entre un et plusieurs produits d'éléments en même temps. Un *GPU* peut effectuer tous les produits (où une large partie) en un même temps.

Dans le cas où la quantité de calcul ne serait pas assez importante, et donc utiliser un *GPU* serait une perte de ressources, ni assez légère, et donc utiliser un *CPU* serait lent et inefficace, on doit faire appel à l'équivalence logicielle-matérielle. En effet, tout programme codé/implémenté possède un équivalent matérielle construit. le problème de ce genre de cette méthode est l'inflexibilité des solutions matérielles même si elles sont plus beaucoup plus rapides. C'est pour cela que l'on utilise les *FPGAs* pour synthétiser les circuits logiques combinatoires et séquentiels.

Les *FPGA* sont de plus en plus présents, Entre *Intel* qui rachète le géant *Altera* et *Microsoft* qui les utilise dans son centre de données *Project Catapult*[3].

Une autre famille de noyau visant le domaine des statistiques, analyse de données et reconnaissances de pattern voit le jour. Développée par *Micron*[4], c'est la famille des *AP* (*Automata Processor*). Les *APs* sont construits avec des *FPGAs* mais créés pour être plus efficace pour le traitement des expressions régulières.

Il existe beaucoup d'autres types de noyaux comme les *DSP*, *ASIC*, *Puce-Neuronique...* c'est noyau ont des usages plus restreint et spécifiques. Donc, une architecture de programmation est quasiment surréaliste.

## 1.5 Avantages et Inconvénients

Les architectures hétérogènes ont plusieurs avantages et inconvénients par rapport aux architectures homogènes.

Avantages :

- Un gain de performances relatif aux tâches à traiter, le système s'adapte au type de traitement qui lui est donné ;
- Un plus grand contrôle du flot de données ;
- Une économie d'énergie remarquable. Ce qui rend ces architectures aussi présentes dans les téléphones portables (*SnapDragon*) ;
- Une meilleure répartition des ressources du système.
- Possibilités de combinaisons de différentes architectures pour une plus grande "spécialisation" des systèmes.

Inconvénients :

- Asymétrie de conception, requiert plusieurs technologies de noyau.
- Coût en programmation plus élevé
- Non modularité et plus coûteux en générale.
- Conception plus complexe même si les efforts dans le domaine tendent à réduire cette complexité comme le cite la *HSA foundation*[5].

### 1.6 Critères d'évaluation

Il est assez difficile de dire objectivement si une architecture est simplement bonne ou mauvaise. Il est plutôt question de compromis entre vitesse, précision et "non-complexité". Néanmoins, il est généralement admis que les critères suivants sont raisonnables car beaucoup de logiciel se basent sur cela :

#### 1.6.1 Nombre d'éléments logiques

On mesure souvent le nombre d'éléments logiques dans une architecture connue pour pouvoir estimer le coût associé à la conception de cette architecture.

#### 1.6.2 Multiplieurs

De manière générale les opérations de multiplication et division ne sont pas désirées dans n'importe quelle architecture car elles causent un ralentissement important du système.

#### 1.6.3 Pthread

Ce nombre représente le nombre de processus parallèles que l'architecture a pu exécuter en temps réel. Il est très important pour les architectures de traitement de données en parallèles.

#### 1.6.4 Nombre de cycles

Il on compte le nombre de cycles d'horloge requis pour toute l'exécution du programme sur l'architecture. Il n'est pas contre-intuitif de penser que ce nombre doit être minimisé mais parfois ce n'est pas le cas si la fréquence de l'horloge est assez élevée.

#### 1.6.5 Fréquence maximale d'horloge

En écho avec le point précédent, on désire généralement que cette fréquence soit la plus grande possible. Mais ce n'est plus vraiment le cas, par exemple les processeurs *Intel* post-sixième génération ont vu leurs fréquences diminués d'environ 50%.

#### 1.6.6 Wall-clock time

Ceci est sans doute le critère le plus important pour une architecture hétérogène. Il est donné par :

$$\text{Wall-clock time} = \text{latence de cycle} \times \text{période d'horloge}$$

Il permet donc de mesurer l'effet des éléments les plus long de l'architecture sur une période d'horloge. On cherche toujours à minimiser ce paramètre.

## 1.7 Conclusion

Les systèmes à architecture hétérogène sont la révolution des dernières décennies. Même si les architectures homogènes sont le standard pour les ordinateurs grâce au fait qu'elles soient plus modulaires et généralisables, Les architectures hétérogènes sont plus performantes et plus adaptatives même cela induit une plus grande complexité de conception et donc un coût tout de suite plus important. Comme on l'a relevé les efforts des constructeurs tendent à essayer de réduire cette complexité et comme le dit *Mohamed Zahran* : "*Heterogeneous computing here to stay*".

## Chapitre 2

# Étude de la parallélisation des calculs pour architecture hétérogène

## 2.1 Introduction

Le calcul parallèle peut être divisé en deux types ; parallèle dans le temps et parallèle dans l'espace. Le parallélisation dans le temps fait référence à la technologie des pipelines, tandis que le parallélisation dans l'espace consiste à effectuer des calculs simultanément avec plusieurs processeurs, c'est-à-dire connecter deux processeurs ou plus via un réseau pour calculer simultanément différentes parties d'une même tâche.

### 2.1.1 Définitions

L'idée est de pouvoir résoudre le problème de manière collaborative, c'est à dire que le problème, qui est assez complexe, est décomposé en plusieurs parties plus simples, et chacune de ces parties est traité par un processeur indépendant. Donc un système parallèle peut être un super-ordinateur spécialement conçu avec plusieurs processeurs ou un groupe d'ordinateurs indépendants inter-connectés d'une manière ou d'une autre.

Succinctement, le calcul parallèle présente les caractéristiques principales suivantes [6] :

- Un problème est divisé en une série de parties distinctes pouvant être exécutées simultanément. Chaque partie peut être décomposée en une série d'instructions discrètes.
- Les instructions de chaque partie peuvent être exécutées simultanément sur différents processeurs.
- Il y a nécessité d'un mécanisme global de contrôle ou de collaboration pour planifier l'exécution de différentes parties.

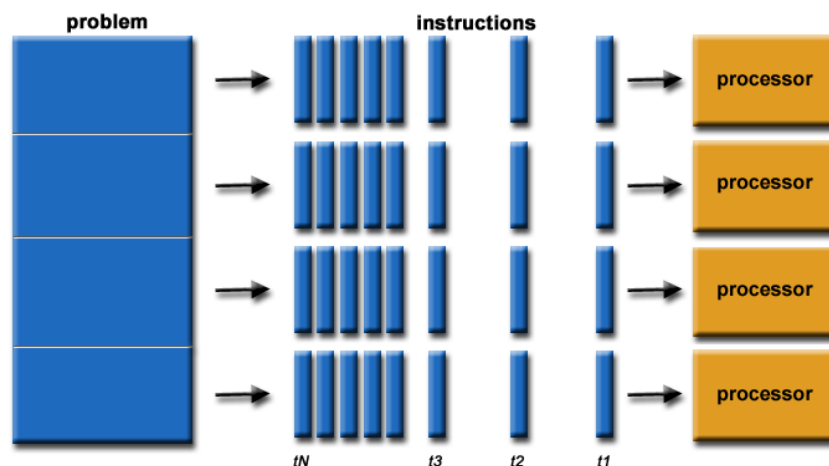


FIGURE 2.1 – Parallélisation d'un problème pour plusieurs processeurs

### 2.1.2 Critères de parallélisation

Bien entendu, tous les algorithmes ne sont pas parallélisables, ou n'en valent pas la peine. Les algorithmes parallélisables appartiennent généralement, mais pas que, à une famille d'algorithmes nommée *Divide and conquer*[7]. De manière plus générale, un algorithme parallélisable répond aux critères suivants :

- Il peut être décomposé en segments discrets pour une exécution simultanée.

Les différents segments discrets peuvent être exécutés à tout moment.

- Le temps d'utiliser plusieurs ressources de calcul doit être inférieur à celui d'utiliser une seule ressource.
- Être exécutable sur un système multi-processeurs/coeurs.

### 2.1.3 Avantages, Inconvénients et Perspectives

#### Avantages

Tout dans la nature est concurrentiel et fonctionne selon sa série chronologique inhérente. Comparé au calcul série, le calcul parallèle convient mieux à la modélisation, à la simulation et à la compréhension de phénomènes complexes dans le monde réel. La supériorité du calcul parallèle se reflète principalement dans les aspects suivants :

- Un gain de temps et d'argent. En théorie, investir plus de ressources dans une tâche permet de réduire son temps de traitement et donc de réduire les coûts. Les ordinateurs parallèles peuvent être constitués d'un grand nombre de machines autonomes peu coûteuses, ce qui permet de réduire les coûts
- Capacité à résoudre des problèmes vastes et complexe.
- Amélioration de la concurrence.
- Utilisation de ressources non locales. En effet, on peut exploiter un réseau en entier.
- Le logiciel parallèle a été appliqué à des conditions matérielles parallèles multicœurs, telles que les threads. Dans la plupart des cas, les programmes en série exécutés sur des ordinateurs modernes gaspillent beaucoup de ressources informatiques.

#### Inconvénients

Cependant, le calcul parallèle n'est pas parfaite. En parlant des défauts et des coûts, nous considérons les aspects suivants :

- Limite du calcul parallèle[8]. La loi d'*Amdahl* [9] montre que l'accélération d'un programme est déterminée par la proportion de parties parallèles du programme. On ne montera pas cette loi dans ce document mais on gardera un de ses résultats qui est :

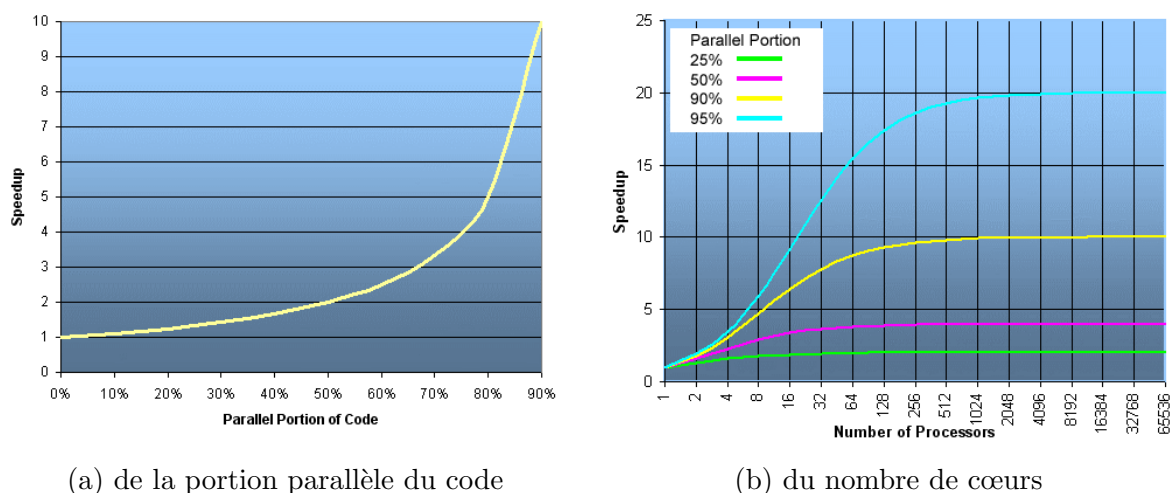


FIGURE 2.2 – Accélération en fonction

Autrement dit quelque soit le nombre de processeurs investis, les programmes parallèles ne bénéficieront pas d'une accélération au-delà de 20 fois[6].

- Complexité. La complexité est généralement déterminée par le temps nécessaire à tous les aspects du cycle de développement du logiciel : conception, codage, débogage, réglage et maintenance.
- Problème de la portabilité. La standardisation de certaines *API* , (Par exemple *MPI*, les threads *POSIX* et *OpenMP*), a fortement diminué ce problème mais il reste non négligeable.
- Besoins de plus de ressources.
- Évolutivité. L'amélioration des performances de calcul parallèle dépend d'une série de facteurs interdépendants, l'ajout de plus de processeurs n'améliore pas nécessairement les performances. De plus, certains problèmes peuvent avoir eux-mêmes des limites d'évolutivité. Par conséquent, l'ajout de ressources supplémentaires peut parfois dégrader les performances. Les bibliothèques ou sous-systèmes parallèles sont également pris en charge pour limiter l'évolutivité des programmes parallèles.

### 2.1.4 Domaine d'application

Historiquement, le calcul parallèle était considéré comme "le haut de gamme de l'informatique". De nombreuses équipes de recherche en sciences et en ingénierie ont adopté le modèle de calcul parallèle pour la modélisation des problèmes dans de nombreux domaines, notamment : l'atmosphère et l'environnement mondial, physique appliquée, sciences biologiques, science de l'ingénierie, armement[10]...

Dans ce rapport, nous étudions principalement l'application du calcul parallèle au traitement d'images.

### 2.1.5 Perspectives

Au cours des deux dernières décennies, l'évolution rapide des réseaux, des systèmes répartis et des architectures de ordinateur avec multiprocesseurs a montré que la parallélisation constituait l'avenir de l'informatique.

Au cours de la même période, les performances des supercalculateurs ont été multipliées par au moins 500 000, et il n'y a aucun signe d'atteindre la limite.[11]

## 2.2 Classification de calcul parallèle

### 2.2.1 Architecture de John von Neumann

L'architecture informatique, nommée d'après le mathématicien hongrois *John von Neumann*, est souvent appelée "ordinateur de programme de stockage" : les instructions de programme et les données sont stockées en mémoire. Depuis lors, tous les ordinateurs ont suivi cette architecture de base. Il y a quatre composants : mémoire, contrôleur, processeur et les entrées/sorties.



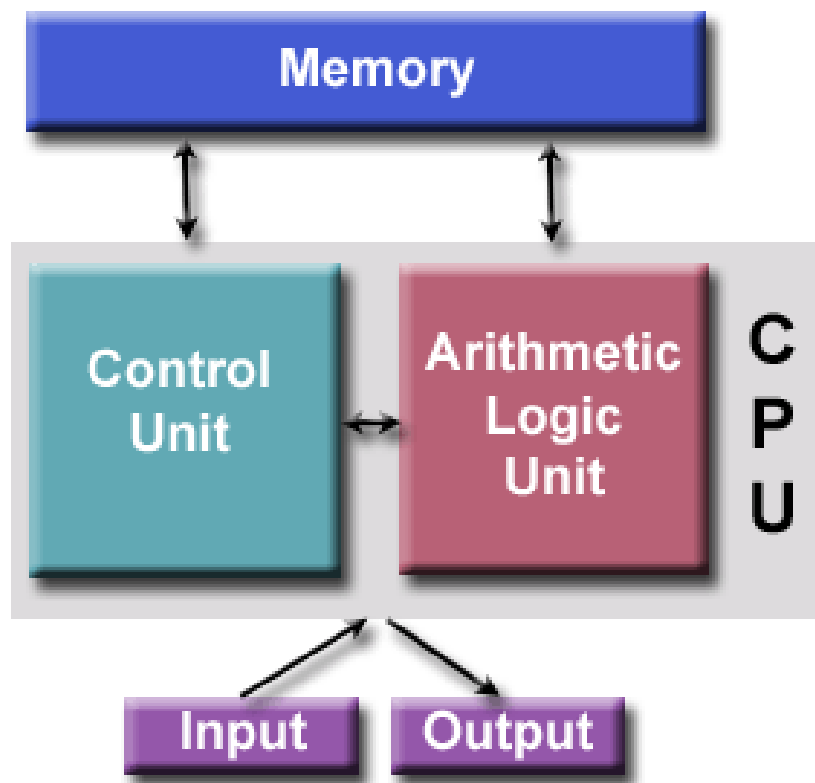


FIGURE 2.3 – Architecture de *Von Neumann*

Les ordinateurs parallèles suivent toujours cette architecture de base. La seule différence est qu'il existe plus d'une unité de traitement, tandis que l'autre architecture de base reste totalement inchangée.

### 2.2.2 Classification de Flynn

La classification de Flynn[12] est une méthode de classification largement utilisée pour les ordinateurs parallèles. Cette classification distingue les architectures de calcul avec multiprocesseurs de deux dimensions : le flux d'instructions et le flux de données. Chaque dimension a un et seulement deux états : simple ou multiple. Il y a quatre états possibles de la classification de *Flynn* : *SISD*, *SIMD*, *MISD* et *MIMD*.

#### **SISD (Single Instruction stream Single Data)**

C'est une machine série et le plus ancien type d'ordinateur.

- Instruction unique. CPU ne peut exécuter qu'un seul flux d'instructions à chaque cycle d'horloge.
- Donnée unique À chaque cycle d'horloge, le périphérique d'entrée peut recevoir uniquement un seul flux de données
- Le résultat de l'exécution est déterministe.

#### **SIMD (Single Instruction stream Multiple Data)**

- Instruction unique

- Données multiples. Chaque unité de traitement peut traiter différents éléments de données.
- Idéal pour le traitement de tâches hautement ordonnées. Ici, on utilise *SIMD* pour le traitement d'images.
- Synchronisation et exécution déterministe.
- Deux types principaux : les matrices de processeurs et les tubes vectoriels. SIMD utilise un contrôleur globale pour contrôler plusieurs micro-éléments de traitement parallèle tout en effectuant les mêmes opérations sur chacun des ensembles de données ("*vecteurs de données*") afin de réaliser un parallélisme spatial.

### MISD (Multiple Instruction stream Single Data)

- Instructions multiples. Différentes unités de traitement peuvent exécuter indépendamment différents flux d'instructions.
- Données uniques. Différentes unités de traitement reçoivent le même flux de données. Ce type d'architecture est théoriquement disponible, mais il existe très peu de tels modèles dans l'industrie en réalité.

### MIMD (Multiple Instruction stream Multiple Data)

*MIMD* est l'un des types les plus courants d'ordinateurs parallèles présentant les caractéristiques suivantes :

- Instructions multiples
- Données multiples
- L'exécution peut être synchrone ou asynchrone, et, déterministe ou incertaine.

C'est le type principale d'architecture informatique. À l'heure actuelle, les superordinateurs, les systèmes de groupes d'ordinateurs parallèles, les grilles, les ordinateurs multiprocesseurs et les ordinateurs multicœurs font tous partie de cette catégorie.

Il convient de noter que de nombreuses architectures de type *MIMD* peuvent en réalité inclure des sous-architectures de *SIMD*.

## 2.3 Architecture de la mémoire de calcul parallèle

### 2.3.1 Mémoire partagée[13, p. 713]

Il existe de nombreux types de systèmes parallèles partageant la mémoire, mais en général, en fonction du temps d'accès à la mémoire, les machines à mémoire partagée existantes peuvent être divisées en deux types : accès mémoire uniforme et accès mémoire non uniforme.

Le type de mémoire uniforme est plus communément appelé *SMP* (*Symmetric Multiprocessor*)[13, p. 549]. Chaque processeur est identique et il n'y a pas de différence entre accès à la mémoire. Alors le type de mémoire non uniforme se compose généralement de deux ou plusieurs *SMP* connectés physiquement. Un *SMP* peut accéder à la mémoire d'autres *SMP*. Tous les processeurs n'ont pas le même accès ou le même temps d'accès pour toute la mémoire.

L'espace d'adressage global fournit un moyen de programmation convivial permettant

un partage rapide et uniforme des données entre les tâches. Mais il y a un manque de bonne évolutivité entre la mémoire et le processeur.

### 2.3.2 Mémoire distribuée

La structure de mémoire distribuée nécessite un réseau de communication pour connecter différentes mémoires. L'adresse mémoire correspondant à un processeur n'est pas mappée sur d'autres processeurs, de sorte que chaque processeur peut fonctionner indépendamment[14].

Avec cette architecture, la mémoire peut évoluer en fonction du nombre de processeurs, chaque processeur peut accéder rapidement à sa propre mémoire sans interférence et il n'y a pas de surcharge de maintenance qui indique à la mémoire cache d'être cohérente. Mais les programmeurs doivent prendre en compte les détails de la communication de données entre processeurs. Les données sur le nœud distant nécessitent un temps d'accès beaucoup plus long que les données sur le nœud local.

### 2.3.3 Mémoire hybride

Actuellement, les ordinateurs parallèles les plus grands et les plus rapides possèdent souvent des architectures de mémoire hybride. L'architecture de la mémoire partagée peut être une machine à mémoire colinéaire ou une unité de traitement graphique (*GPU*). Le composant de mémoire distribuée peut être un système connecté par plusieurs mémoires partagées / *GPU*. Chaque nœud ne connaît que sa propre mémoire et ne connaît pas la mémoire des autres nœuds du réseau. Par conséquent, la communication de données via le réseau est requise sur différentes machines.

De la tendance actuelle, cette architecture de mémoire hybride dominera longtemps et deviendra le meilleur choix pour l'informatique haut de gamme dans un avenir prévisible.

## 2.4 Modèle de calcul parallèle

Les modèles de programmation parallèle courants incluent :

- modèle de mémoire partagée.
- modèle de threading.
- modèle de mémoire / messagerie distribuée
- modèle parallèle de données
- modèle hybride
- modèle mono-programme multi-données (*SPMD*)
- modèle multi-programmes multi-données (*MPMD*)

Le modèle informatique parallèle est une abstraction sur une architecture matérielle et mémoire. Bien que cela ne soit pas forcément évident, ces modèles ne sont pas liés à des architectures de machine et de mémoire spécifiques.

Ici, nous nous concentrons sur le modèle multi-données à programme unique (*SPMD*)[15].

En fait, *SPMD* (*Single Program Multiple Data*) est un modèle de programmation plus "avancé" qui peut être architecturé par-dessus d'autres modèles de programmation parallèle :

- Programme unique. Toutes les tâches exécutent une copie du même programme. Le programme peut ici être un fil d'exécution, la transmission de messages, des données parallèles ou même mélangé.
- Données multiples. Différentes tâches fonctionnent sur des données différentes.

SMPD doit généralement spécifier la logique d'exécution de la tâche. Différentes tâches peuvent effectuer certaines parties du programme dans son ensemble en fonction de la création de branches et de relations logiques, ce qui signifie que toutes les tâches ne doivent pas nécessairement exécuter le programme dans son intégralité : il peut ne s'agir que d'une partie du programme.

Ce modèle *SPMD* avec messagerie ou programmation hybride est le modèle d'informatique parallèle le plus répandu qui s'exécute actuellement sur des systèmes de grappes multicœurs.

## 2.5 Conception de programme parallèle[16]

### 2.5.1 Parallélisation automatique ou manuelle[17]

Les programmeurs sont généralement responsables de l'identification et de la mise en oeuvre du parallélisme. Beaucoup d'outils ont été développés pour les aider à convertir des programmes sériels en programmes parallèles. Les outils les plus courants sont les compilateurs parallèles ou les pré-processeurs.

Les compilateurs parallèles fonctionnent généralement de deux manières : complètement automatiquement ou selon les instructions du programmeur. Le paralléliseur le plus commun généré par le compilateur est implémenté en utilisant une mémoire partagée et des threads à l'intérieur du noeud (par exemple, *OpenMP*).

### 2.5.2 Compréhension des problèmes et des procédures

Avant de commencer à essayer de développer une solution parallèle, il faut déterminer si le problème peut réellement être mis en parallèle. Comprendre les problèmes et les procédures principalement à travers les trois étapes d'identification des points clés, d'identification des goulots d'étranglement et d'identification des inhibiteurs de la parallélisation.

Si possible, il est meilleur d'étudier d'autres algorithmes à l'aide des logiciels parallèles tiers ou de bibliothèques mathématiques hautement sophistiquées (*ESSL* d'*IBM*, *MKL* d'*Intel*, *AMCL* d'*AMD*, etc.)

### 2.5.3 Partitionnement

La troisième étape de la conception d'un programme parallèle consiste à diviser le programme en "blocs" pouvant être affectés à différentes tâches. Ceci est appelé décomposition ou partitionnement du programme. Il existe généralement deux méthodes de base pour décomposer des tâches parallèles :

- La décomposition de domaine, les données sont décomposées en fonction du problème. Chaque tâche parallèle traite une partie des données.
- La décomposition fonctionnelle, l'accent est mis sur les calculs à effectuer mais pas sur les données manipulées. Le problème est décomposé en fonction du travail à effectuer, puis chaque tâche exécute une partie du travail complet.

### 2.5.4 Communication

Un certain nombre de facteurs importants doivent être pris en compte lors de la conception de la communication entre les tâches du programme. Par exemple, le délai, la dépense, la visibilité de la communication, l'étendue, l'efficacité, la complexité, et le synchronisme ou l'asynchronisme, etc.

### 2.5.5 Synchronisation

L'ordre des tâches et l'exécution sont la clé du bon fonctionnement du programme parallèle, mais aussi de l'amélioration de ses performances. Il est généralement nécessaire de "sérialiser" certains programmes.

### 2.5.6 Dépendance de données

Lorsque l'ordre d'exécution des instructions affecte le résultat en cours du programme, il existe une dépendance entre les instructions du programme. Ces dépendances sont générées par plusieurs accès au même emplacement mémoire par différentes tâches.

La dépendance des données est également essentielle dans la programmation parallèle, car il s'agit d'un inhibiteur important de la parallélisation.

### 2.5.7 Équilibrage de charge

L'équilibrage de charge est la pratique qui consiste à allouer des quantités de travail à peu près égales entre les tâches de manière à ce que toutes les tâches restent occupées à tout moment. Elle est également considérée comme une réduction du temps d'inactivité des tâches.

Nous pouvons utiliser la méthode de répartition uniforme du nombre de tâches ou adopter la méthode d'allocation dynamique des tâches. Lorsque la charge de travail de chaque tâche est variable ou imprévisible, il faut utiliser la méthode du pool de tâches du planificateur. Chaque fois qu'une tâche termine son travail, elle peut extraire une nouvelle tâche de la file d'attente de travail.

### 2.5.8 Granularité

Dans le calcul parallèle, la granularité décrit quantitativement le rapport entre le calcul et la communication. Une granularité grossière signifie que beaucoup de travail de calcul doit être fait dans le processus de communication, tandis qu'une granularité fine signifie qu'il n'y a pas beaucoup de travail de calcul à effectuer dans le processus de communication.

La granularité la plus efficace dépend de l'algorithme spécifique et de l'environnement matériel dans lequel il s'exécute. Dans la plupart des cas, la dépense associée à la communication / synchronisation est très grosse. Par conséquent, le problème avec une taille de grain grossière est relativement simple. D'autre part, une granularité fine peut aider à réduire les déficits généraux causés par le déséquilibre de charge.

### 2.5.9 Entrée et sortie (I/O)

Il existe déjà de nombreux systèmes de fichiers parallèles, tels que : *GPFS*, *Lustre*, *HDFS*, *PanFS*, etc. Il faut faire attention aux points suivants lors de la conception d'E/S :

- Minimisez les opérations d'E/S globales
- Limitez les opérations d'E/S à des parties série spécifiques du travail, puis utilisez la communication parallèle pour distribuer les données aux tâches parallèles.
- Une meilleure stratégie consiste à n'avoir qu'un sous-ensemble de tâches pour effectuer les opérations d'E/S.

### 2.5.10 Débogage

Le débogage de code parallèle peut être très difficile, en particulier à mesure que la quantité de code augmente. Mais nous pouvons maintenant utiliser d'excellents débogueurs : *Threaded - Pthreads et OpenMP*, *MP*, *GPU/accélérateur et Hybride*.

### 2.5.11 Analyse et optimisation des performances

Il existe également de nombreux outils d'analyse et de débogage des performances de programmes parallèles. La plupart de ces outils sont hors de portée de notre étude. On se maintiendra aux cas simples d'algorithmes parallélisables.

## 2.6 Conclusion

Dans ce chapitre nous avons vu les principes fondamentaux des modèles de parallélisation possibles pour une architecture ciblée, les avantages et inconvénients associées à ces modèles. On a aussi pu constater qu'il exister plusieurs contraintes pour la conception d'algorithmes parallèles et parallélisable. On comprend l'ampleur de la tâche à réaliser car ces algorithmes ni évidents à concevoir, ni à déboguer/maintenir. Il n'en reste pas moins que ces algorithmes sont les plus utilisés de nos jours et cette tendance ne changera pas d'aussi tôt.

## Chapitre 3

“CPU + FPGA” pour le traitement  
d'image

### 3.1 Introduction

À mesure que les performances des processeurs continuent d’augmenter et que les technologies de traitement parallèle, telles que les *CPU* multicœurs et les *FPGA*, deviennent plus sophistiquées, désormais, on peut appliquer des algorithmes sophistiqués pour visualiser les données et créer des systèmes plus intelligents.

Grâce aux performances améliorées, un débit de données supérieur peut être obtenu pour une acquisition plus rapide des images. Dans le même temps, les images peuvent être traitées plus rapidement : les algorithmes de prétraitement (tels que les seuillages et les filtrages) ou de traitement (tels que la mise en correspondance de formes) peuvent également être exécutés plus rapidement.

Alors que les systèmes de vision intègrent de plus en plus de processeurs et de *FPGA* multicœurs de dernière génération, nous devons non seulement exécuter les algorithmes appropriés sur le matériel approprié, mais également déterminer quelles architectures sont les mieux adaptées à leur conception.

### 3.2 Co-traitement et Traitement en ligne[18]

#### 3.2.1 Co-traitement

Lors du développement d’un système de vision basé sur une architecture hétérogène de *CPU* et de *FPGA*, deux principaux cas d’utilisation doivent être pris en compte : le traitement et le coprocessing intégrés.

S’il s’agit d’un co-traitement *FPGA*, le *FPGA* et la *CPU* vont travailler ensemble et partager la charge de traitement. Les images peuvent être capturées à l’aide du *CPU*, puis envoyées au *FPGA* via un accès direct à la mémoire (DMA) afin que le *FPGA* puisse effectuer des opérations telles que le filtrage ou l’extraction de plan de couleur. L’image sera ensuite renvoyée au *CPU* pour des opérations plus avancées telles que la reconnaissance optique de caractères (*OCR*) ou la correspondance de motifs.

Dans certains cas, toutes les étapes de traitement peuvent être implémentées sur le *FPGA* et seuls les résultats de traitement sont renvoyés à la *CPU*. Cela permet à la *CPU* d’utiliser plus de ressources pour d’autres opérations telles que la commande de mouvement, la communication réseau et l’affichage d’images.

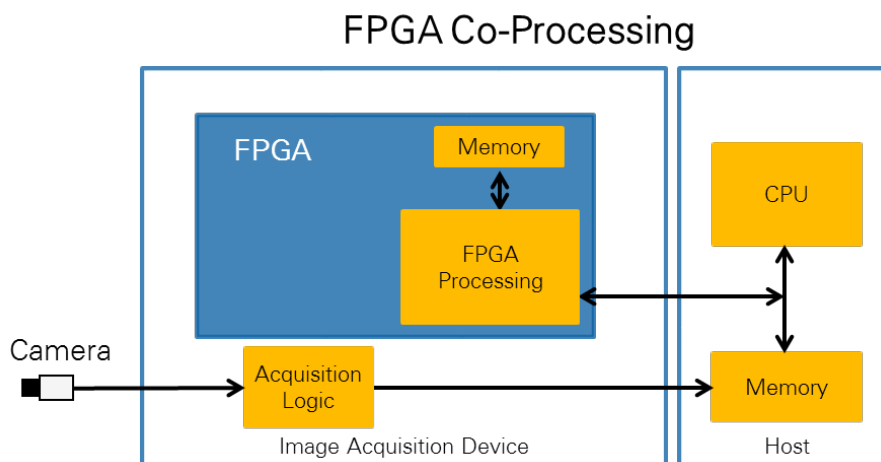


FIGURE 3.1 – Schéma représentatif du co-traitement entre *FPGA* et *CPU*



### 3.2.2 Traitement en ligne

Dans une architecture de traitement *FPGA* embarquée, l’interface de la caméra peut être directement connectée aux broches du *FPGA* afin que les pixels puissent être envoyés directement de la caméra au *FPGA*. La logique d’acquisition d’image est facile à mettre en oeuvre à l’aide de circuits numériques sur le *FPGA*.

Cette architecture présente deux avantages principaux. Tout d’abord, comme pour le co-traitement, lors du traitement préalable sur un *FPGA*, on peut utiliser le traitement embarqué pour déplacer une partie du travail du processeur vers le *FPGA*. Par exemple, un pré-traitement à grande vitesse (tel que le filtrage ou le seuillage), peut être effectué sur le *FPGA* avant que les pixels ne soient envoyés à la *CPU*. Cela réduit également la quantité de données que la *CPU* doit traiter, car la logique de la *CPU* doit seulement capturer les pixels de la région d’intérêt. Par conséquence, cela augmente finalement le débit de l’ensemble du système.

Le second avantage de cette architecture est la possibilité d’effectuer des opérations de contrôle à grande vitesse directement dans le *FPGA* sans utiliser de processeur. Les *FPGA* sont idéaux pour les applications de contrôle car ils fournissent des cadences très rapides et très déterministes.

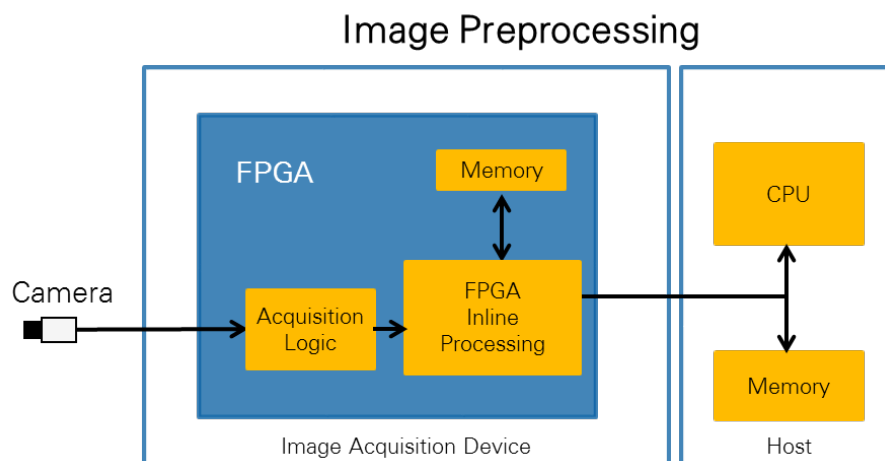


FIGURE 3.2 – Schéma représentatif du traitement en ligne entre *FPGA* et *CPU*

## 3.3 Algorithmes de traitement d’image et vision *CPU* et *FPGA*

La *CPU* effectue les opérations dans l’ordre, de sorte que la deuxième opération puisse être lancée une fois la première opération terminée sur l’ensemble de l’image. Les *FPGA* sont essentiellement massivement parallèles, ce qui permet à plusieurs opérations de fonctionner simultanément sur différents pixels de l’image.

Bien que les *FPGA* soient plus utiles que les processeurs pour le traitement des images et des images, il existe des compromis pour profiter de ces avantages. Par exemple, la fréquence d’horloge du *FPGA* est de l’ordre de 100 à 200 MHz. Or la *CPU* peut fonctionner à 3 GHz ou plus. De toute évidence, la fréquence d’horloge du *FPGA* est inférieure à la fréquence d’horloge de la *CPU*. Par conséquent, si un processus nécessite un algorithme de traitement d’image qui doit être itéré, et que le parallélisme du *FPGA* ne peut pas

être utilisé, la *CPU* peut traiter plus rapidement. Toutefois, si l’algorithme utilise des étapes telles que la mise en correspondance des modèles et l’*OCR* nécessitant une analyse immédiate de toute l’image, les avantages des *FPGA* ne sont pas mis en évidence. Cela est dû au manque de parallélisation des étapes de traitement et à la nécessité de disposer de grandes quantités de mémoire pour l’analyse comparative d’images et de modèles. Bien que les *FPGA* aient un accès direct à la mémoire interne et externe, en général, la quantité de mémoire disponible pour le *FPGA* est de loin inférieure à la quantité de *CPU* disponible.

Les avantages des *FPGA* pour le traitement des images dépendent des exigences de chaque application, notamment des algorithmes spécifiques à l’application, des exigences en matière de latence, de la synchronisation des E/S et de la consommation d’énergie. L’approche habituelle consiste à utiliser une architecture avec *FPGA* et *CPU* pour tirer pleinement parti des avantages du *FPGA* et du *CPU*, tout en offrant un avantage concurrentiel en termes de performances, de coût et de fiabilité.

Cependant, le développement d’algorithmes visuels est essentiellement un processus itératif. L’un des plus grands défis de la mise en oeuvre de systèmes de vision basés sur *FPGA* est donc de surmonter la complexité de la programmation des *FPGA*.

## 3.4 Conclusion

En résumé, il faut comprendre les objectifs du traitement pour choisir les méthodes les mieux adaptés au design. Toutefois, quelle que soit l’application, l’architecture à base de *CPU* ou de *FPGA* et ses avantages inhérents peuvent considérablement améliorer les performances des traitements d’images et de vision des machines.

La conception d’algorithme parallélisable est assez délicate. Pour notre étude, il nous faudra d’abord prendre des programmes assez simples afin de bien réussir leur implémentation. Puis étendre le raisonnement aux programmes plus complexe.

# Chapitre 4

## Logiciels pour architectures hétérogènes

## 4.1 Introduction

Quand un programme s'exécute, il y a beaucoup de processus différents qui doivent être pris en compte quelque soit l'architecture matérielle. Afin de pouvoir concevoir des programmes pour des systèmes à architecture hétérogène, il faut posséder les outils nécessaires. Il existe plusieurs logiciels permettant cette programmation. Ces logiciels diffèrent suivant des critères matériels, où chaque constructeur privilégie son logiciel pour ses cartes programmable, des critères d'optimisation et même des critères liés au type de licence qui peut être accordée. Dans ce chapitre, on se limite à l'étude de *OpenCL* et *Vivado-HLS*.

## 4.2 OpenCL

*OpenCL* (*Open Computing Language*) est un langage de programmation qui offre la possibilité de programmer les différents dispositifs et de distribuer les tâches pour chaque dispositif[19]. Les dispositifs en question sont les différents types de noyaux vus précédemment [?]. Par exemple, on peut utiliser un *CPU* pour faire les tâches de contrôle et un *GPU* ou *FPGA* pour faire les tâches de calcul. *OpenCL* est conçu pour paralléliser les calculs entre les différents noyaux.

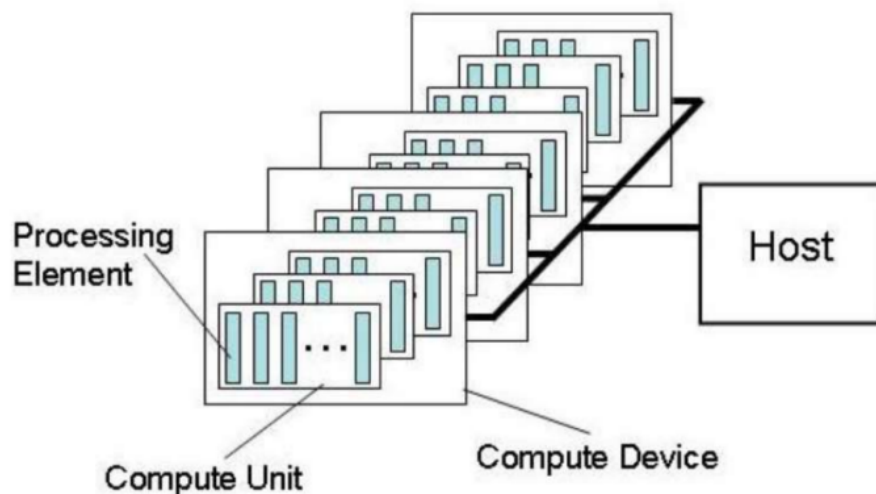


FIGURE 4.1 – Schéma de coordination de l'hôte pour les tâches et données

### 4.2.1 Acquisition

On peut le trouver dans les *SDK* (*Software Development Kit*) offert par les différentes entreprises :

- *Intel SDK for OpenCL Application*.
- *CUDA* par *NVIDIA*.
- *ATI Stream* par *AMD*.

Une fois l'acquisition des *SDK* faite, on reçoit des fichiers d'en-tête (*.h*) et des fichiers de lien de bibliothèque (*.lib*).

LOCAL (C:) > Programmes > NVIDIA GPU Computing Toolkit > CUDA > v10.0 > include > CL				
Nom	Modifié le	Type	Taille	
cl.h	26/08/2018 12:33	Header file	63 Ko	
cl.hpp	26/08/2018 12:33	Header file	305 Ko	
cl_d3d9_ext.h	26/08/2018 12:33	Header file	6 Ko	
cl_d3d10.h	26/08/2018 12:33	Header file	5 Ko	
cl_d3d10_ext.h	26/08/2018 12:33	Header file	5 Ko	
cl_d3d11.h	26/08/2018 12:33	Header file	5 Ko	
cl_d3d11_ext.h	26/08/2018 12:33	Header file	5 Ko	
cl_dx9_media_sharing.h	26/08/2018 12:33	Header file	6 Ko	
cl_egl.h	26/08/2018 12:33	Header file	6 Ko	
cl_ext.h	26/08/2018 12:33	Header file	15 Ko	
cl_gl.h	26/08/2018 12:33	Header file	8 Ko	
cl_gl_ext.h	26/08/2018 12:33	Header file	3 Ko	
cl_platform.h	26/08/2018 12:33	Header file	42 Ko	
opencl.h	26/08/2018 12:33	Header file	2 Ko	

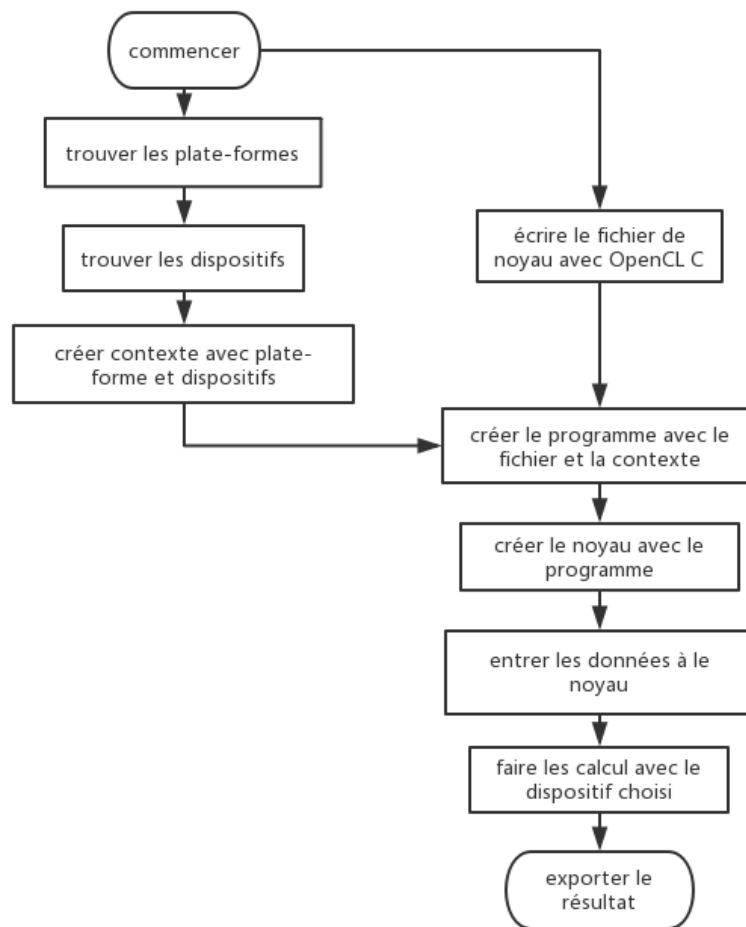
  

LOCAL (C:) > Programmes > NVIDIA GPU Computing Toolkit > CUDA > v10.0 > lib > Win32				
Nom	Modifié le	Type	Taille	
cuda.lib	26/08/2018 12:33	Object File Library	109 Ko	
cudadevrt.lib	26/08/2018 12:33	Object File Library	490 Ko	
cudart.lib	26/08/2018 12:33	Object File Library	82 Ko	
cudart_static.lib	26/08/2018 12:33	Object File Library	1 893 Ko	
OpenCL.lib	26/08/2018 12:33	Object File Library	28 Ko	

FIGURE 4.2 – liste de fichiers pré-requis

### 4.2.2 Programmation

Une fois les préparatifs finis, on peut commencer à programmer avec *OpenCL* qui suit Le processus suivant [20] :

FIGURE 4.3 – Organigramme de déroulement de la programmation *OpenCL*

### Les plateformes

Il faut d'abord trouver les plateformes d'*OpenCL*, c-à-d le *SDK* qu'offre l'entreprise. On commence par les deux fonctions [21]

```

cl_int clGetPlatformIDs (cl_uint num_entries,
                        cl_platform_id * platforms,
                        cl_uint * num_platforms)

cl_int clGetPlatformInfo (cl_platform_id platform,
                        cl_platform_info param_name,
                        size_t param_value_size,
                        void * param_value,
                        size_t * param_value_size_ret)

```

FIGURE 4.4 – Prototypes des fonctions *clGetPlatformIDs* et *clGetPlatformInfo*

La fonction *clGetPlatformIds* renvoie les IDs. Il est nécessaire de l'appeler deux fois, La première fois pour mettre les paramètres *num\_entries* à 0 et *platforms* à NULL. Ainsi, on a le nombre de plateformes ; La deuxième pour mettre le paramètre *num\_platforms* à NULL et donc trouver les IDs.

```
errnum = clGetPlatformIDs(0, NULL, &numPlatforms);
if (errnum != CL_SUCCESS || numPlatforms <= 0)
{
    cerr << "Failed to find any OpenCL platform1." << endl;
    return;
}
platformIds = (cl_platform_id *)alloca(sizeof(cl_platform_id) * numPlatforms);
errnum = clGetPlatformIDs(numPlatforms, platformIds, NULL);
if (errnum != CL_SUCCESS) {
    cerr << "Failed to find any OpenCL platforms2." << endl;
    return;
}
```

FIGURE 4.5 – Appel de la fonction *clGetPlatformIDs*

De même, pour trouver les infos des OpenCL plate-formes, on doit utiliser la fonction *clGetPlatformInfo* deux fois aussi.

```
errNum = clGetPlatformInfo(id, name, 0, NULL, &paramValueSize);
if (errNum != CL_SUCCESS) {
    cerr << "Failed to find OpenCL platform" << str<< "." << endl;
    return;
}
char *info = (char *)alloca(sizeof(char)*paramValueSize);
errNum = clGetPlatformInfo(id, name, paramValueSize, info, NULL);
if (errNum != CL_SUCCESS) {
    cerr << "Failed to find OpenCL platform" << str << "." << endl;
    return;
}
```

FIGURE 4.6 – Appel de la fonction *clGetPlatformInfo*

Le code devient donc :

```
void displayInfo(void) {
    cl_int errnum;
    cl_uint numPlatforms;
    cl_platform_id *platformIds;
    cl_context context = NULL;

    errnum = clGetPlatformIDs(0, NULL, &numPlatforms);
    if (errnum != CL_SUCCESS || numPlatforms <= 0)
    {
        cerr << "Failed to find any OpenCL platform1." << endl;
        return;
    }
    platformIds = (cl_platform_id *)alloca(sizeof(cl_platform_id) * numPlatforms);
    errnum = clGetPlatformIDs(numPlatforms, platformIds, NULL);
    if (errnum != CL_SUCCESS) {
        cerr << "Failed to find any OpenCL platforms2." << endl;
        return;
    }
    cout << "Number of platforms:\t" << numPlatforms << endl;
    for (cl_uint i = 0; i < numPlatforms; i++) {
        DisplayPlatformInfo(platformIds[i], CL_PLATFORM_PROFILE, "CL_PLATFORM_PROFILE");
        DisplayPlatformInfo(platformIds[i], CL_PLATFORM_VERSION, "CL_PLATFORM_VERSION");
        DisplayPlatformInfo(platformIds[i], CL_PLATFORM_VENDOR, "CL_PLATFORM_VENDOR");
        DisplayPlatformInfo(platformIds[i], CL_PLATFORM_EXTENSIONS, "CL_PLATFORM_EXTENSIONS");
    }
}
```

FIGURE 4.7 – Implémentation de la fonction *displayinfo*

```

void DisplayPlatformInfo(cl_platform_id id, cl_platform_info name, std::string str) {
    cl_int errNum;
    size_t paramValueSize;
    errNum = clGetPlatformInfo(id, name, 0, NULL, &paramValueSize);
    if (errNum != CL_SUCCESS) {
        cerr << "Failed to find OpenCL platform" << str << "." << endl;
        return;
    }

    char *info = (char *)alloca(sizeof(char)*paramValueSize);
    errNum = clGetPlatformInfo(id, name, paramValueSize, info, NULL);
    if (errNum != CL_SUCCESS) {
        cerr << "Failed to find OpenCL platform" << str << "." << endl;
        return;
    }

    cout << "\t" << str << ": \t" << info << endl;
}

```

FIGURE 4.8 – Implémentation de la fonction *DisplayPlatformInfo*

Il suffit d'appeler la fonction *DisplayInfo* dans le main et on observe le résultat suivant :

```

Number of platforms:    2
CL_PLATFORM_PROFILE:    FULL_PROFILE
CL_PLATFORM_VERSION:    OpenCL 1.2 CUDA 10.1.112
CL_PLATFORM_VENDOR:     NVIDIA Corporation
CL_PLATFORM_EXTENSIONS: cl_khr_global_int32_base_atomics cl_khr_global_int32_extended_atomics
cl_khr_local_int32_base_atomics cl_khr_local_int32_extended_atomics cl_khr_fp64 cl_khr_byte_addressable_store
cl_khr_icd cl_khr_gl_sharing cl_nv_compiler_options cl_nv_device_attribute_query cl_nv_pragma_unroll
cl_nv_d3d10_sharing cl_khr_d3d10_sharing cl_nv_d3d11_sharing cl_nv_copy_opts cl_nv_create_buffer
CL_PLATFORM_PROFILE:    FULL_PROFILE
CL_PLATFORM_VERSION:    OpenCL 1.2
CL_PLATFORM_VENDOR:     Intel(R) Corporation
CL_PLATFORM_EXTENSIONS: cl_intel_dx9_media_sharing cl_khr_3d_image_writes cl_khr_byte_addressable_store
cl_khr_d3d11_sharing cl_khr_depth_images cl_khr_dx9_media_sharing cl_khr_gl_sharing cl_khr_global_int32_base_atomics
cl_khr_global_int32_extended_atomics cl_khr_icd cl_khr_local_int32_base_atomics cl_khr_local_int32_extended_atomics
cl_khr_spir

```

FIGURE 4.9 – Affichage des plateformes disponibles

On voit qu'il y a deux plateformes *OpenCL* dans mon ordinateur : *Intel SDK for OpenCL Application* de intel et *CUDA* de *NVIDIA*.

### 4.2.3 Les dispositifs

Après avoir trouvé les plateformes *OpenCL*, la deuxième étape est de trouver les dispositifs *OpenCL*. on peut utiliser les deux fonctions comme suit :

```

cl_int clGetDeviceIDs (cl_platform_id platform,
                      cl_device_type device_type,
                      cl_uint num_entries,
                      cl_device_id *devices,
                      cl_uint *num_devices)

cl_int clGetDeviceInfo (cl_device_id device,
                      cl_device_info param_name,
                      size_t param_value_size,
                      void * param_value,
                      size_t * param_value_size_ret)

```

FIGURE 4.10 – Prototypes des fonctions *clGetDeviceIDs* et *clGetDeviceInfo*



L'usage de ces deux fonctions est presque le même que les deux précédentes, chaque méthode on doit l'appeler deux fois. Pour la fonction `clGetDeviceIDs()`, on peut mettre les paramètres `num_entries` à 0, et `devices` à NULL, donc on peut avoir la valeur de `num_devices` pour le premier appel. Dans le deuxième fois, on peut mettre le paramètre `num_devices` à NULL et recevoir les IDs des dispositifs.

On peut changer le valeur de `device_type` pour trouver les différents dispositifs (*GPU, CPU, Accelerator...*). pour simplifier, on prend l'exemple de *GPU*, donc on peut mettre la variable `device_type` égale à `CL_DEVICE_GPU`.

Avec les IDs des dispositifs, on peut trouver leurs infos par la méthode `clGetDeviceInfo()`, par exemple, si on met le paramètre `param_name` égale à `CL_DEVICE_NAME`, on peut avoir les noms des dispositifs.

```
err = clGetDeviceInfo(deviceIds[i], CL_DEVICE_NAME, 0, NULL, &size);
char *name = (char *)alloca(sizeof(char)* size );
err = clGetDeviceInfo(deviceIds[i], CL_DEVICE_NAME, size, name, NULL);

void DisplayDevice(cl_platform_id id) {
    cl_int errNum;
    cl_uint numDevices;
    cl_device_id deviceIds[1];
    errNum = clGetDeviceIDs(id, CL_DEVICE_TYPE_GPU, 0, NULL, &numDevices);
    if (numDevices < 1) {
        cout << "no GPU device found for platform" << endl;
        exit(1);
    }
    errNum = clGetDeviceIDs(id, CL_DEVICE_TYPE_GPU, 1, &deviceIds[0], NULL);
    for (cl_uint i = 0; i < numDevices; i++) {
        cl_int err;
        size_t size;
        err = clGetDeviceInfo(deviceIds[i], CL_DEVICE_NAME, 0, NULL, &size);
        char *name = (char *)alloca(sizeof(char)* size );
        err = clGetDeviceInfo(deviceIds[i], CL_DEVICE_NAME, size, name, NULL);
        cout << "Dispositif est " << name << endl;
    }
}
```

FIGURE 4.11 – Implémentation de la fonction *DisplayDevice*

```
Number of platforms:    2
CL_PLATFORM_PROFILE:   FULL_PROFILE
CL_PLATFORM_VERSION:   OpenCL 1.2 CUDA 10.1.112
CL_PLATFORM_VENDOR:    NVIDIA Corporation
Dispositif est GeForce GTX 860M
CL_PLATFORM_PROFILE:   FULL_PROFILE
CL_PLATFORM_VERSION:   OpenCL 1.2
CL_PLATFORM_VENDOR:    Intel(R) Corporation
Dispositif est Intel(R) HD Graphics 4600
```

FIGURE 4.12 – affichage des dispositifs sur différentes plateformes

On peut voir que les *dispositifs OpenCL* pour les différentes plateformes sont différents. C'est parce qu'on met le paramètre `device_type` égale à `CL_DEVICE_GPU`, on a seulement trouvé les *GPU* sur les deux plateformes, si l'on mettait le paramètre `device_type` égale à `CL_DEVICE_ALL`, on peut trouver *CPU* et *GPU* sur la plateforme d'*Intel*.

#### 4.2.4 Le contexte avec plateforme et dispositifs

Le troisième étape est de créer un contexte de *OpenCL*, avec ce contexte ,on peut coordonner l'hôte avec les *dispositifs OpenCL*, Pour créer le contexte, il faut utiliser la fonction *clCreateContextFromType*. Les paramètres de la fonction qui nous intéressent sont seulement les IDs des plateformes et le type de dispositif. Les autres sont NULL.

```

clCreateContextFromType (
    const cl_context_properties *properties,
    cl_device_type device_type,
    void (CL_CALLBACK *pfn_notify)
        (const char *errinfo,

    const void *private_info,

    size_t cb,

    void *user_data),
    void *user_data,
    cl_int *errcode_ret)

void DisplayContext(cl_platform_id id) {
    cl_context context = NULL;
    size_t size;
    cl_context_properties properties[] = { CL_CONTEXT_PLATFORM, (cl_context_properties)id, 0 };
    context = clCreateContextFromType(properties, CL_DEVICE_TYPE_ALL, NULL, NULL, NULL);
    clGetContextInfo(context, CL_CONTEXT_DEVICES, 0, NULL, &size);
    cl_device_id *devices = (cl_device_id *)alloca(sizeof(cl_device_id)*size);
    clGetContextInfo(context, CL_CONTEXT_DEVICES, size, devices, NULL);
    for (size_t i = 0; i < size / sizeof(cl_device_id); i++) {
        cl_device_type type;
        clGetDeviceInfo(devices[i], CL_DEVICE_TYPE, sizeof(cl_device_type), &type, NULL);
        switch (type) {
            case CL_DEVICE_TYPE_GPU: cout << "GPU" << endl; break;
            case CL_DEVICE_TYPE_CPU: cout << "CPU" << endl; break;
            case CL_DEVICE_TYPE_ACCELERATOR: cout << "Accelerator" << endl; break;
            default: cout << "pas du tout" << endl; break;
        }
    }
}

```

FIGURE 4.13 – Déclaration et implémentation de la fonction *clCreateContextFromType*

On sait que l'on peut créer deux contextes différents sur les deux plate-formes. il est clair qu'une plateforme ne peut pas contenir tous les dispositifs dans mon ordinateur. En fait, les différentes plateformes soutiennent différents dispositifs :

- La plateforme *CUDA* soutient seulement les *GPUs* de *NVIDIA*;
- La plateforme de *Intel* soutient ses propres *CPUs* et *GPUs*;
- La plateforme *ATI* soutient *GPU* de *AMD* et *CPU* de *Intel*;

On peut regarder le résultat, on a créé les différents contextes sur pour deux plateformes.

```

Number of platforms:    2
  CL_PLATFORM_PROFILE:  FULL_PROFILE
  CL_PLATFORM_VERSION:  OpenCL 1.2 CUDA 10.1.112
  CL_PLATFORM_VENDOR:   NVIDIA Corporation
Dispositif est GeForce GTX 860M
GPU

  CL_PLATFORM_PROFILE:  FULL_PROFILE
  CL_PLATFORM_VERSION:  OpenCL 1.2
  CL_PLATFORM_VENDOR:   Intel(R) Corporation
Dispositif est Intel(R) HD Graphics 4600
CPU
GPU

```

FIGURE 4.14 – Affichage des dispositifs dans le contexte choisi

Sur la plateforme *CUDA*, on peut seulement créer un contexte avec *GPU*, parce qu'il y a seulement un *dispositif OpenCL GPU* sur cette plateforme.

### 4.2.5 Implémentation du noyau

La conception du noyau est une étape clef où le caractère le plus important pour *OpenCL C* est le vecteur variable. Par exemple, les variables de type *floatN* sont des vecteurs de *N* variables de type *float*. Ceci permet d'éviter les boucles lors du calcul de vecteur. De plus, il est possible de sélectionner un vecteur variable tel que :

```

(float4) ( float, float, float, float)
(float4) ( float2, float, float)
(float4) ( float, float2, float)
(float4) ( float, float, float2)
(float4) ( float2, float2 )
(float4) ( float3, float)
(float4) ( float, float3)
(float4) ( float)

```

Si on fait l'addition de tableau avec le langage *C*, l'opérateur est de type série ; et donc on doit utiliser une boucle *for*.

```

void
scalar_add (int n, const float *a, const float *b, float *result)
{
    int i;
    for (i=0; i<n; i++)
        result[i] = a[i] + b[i];
}

```

FIGURE 4.15 – Implémentation d'une boucle *for*

Si la longueur du tableau est *N*, le programme doit répéter la boucle *N* fois, C'est ce que l'on veut éviter. Le langage *OpenCL C* est dessiné pour les noyaux qui peuvent faire les calculs parallèles.

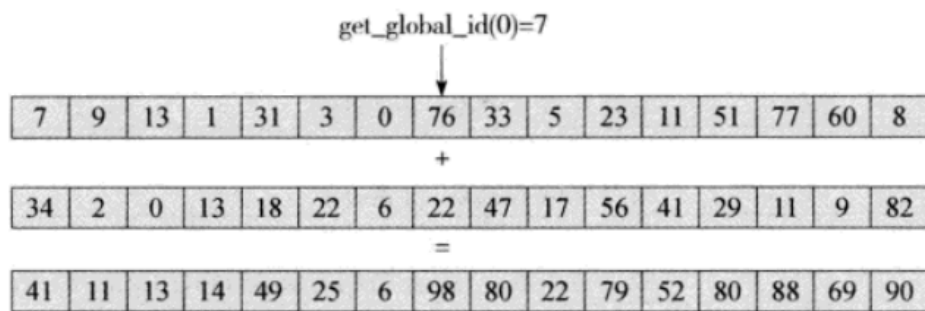


FIGURE 4.16 – Représentation d’une opération sur deux tableaux de données

Le programme pour faire l’addition des tableaux avec le langage *OpenCL C* est :

```
kernel void hello_kernel(global const float *a, global const float *b, global float *result) {
    int gid = get_global_id(0);
    result[gid] = a[gid] + b[gid];
}
```

FIGURE 4.17 – Implémentation du noyau pour le calcul

On écrit les codes dans un fichier du noyau (par exemple kernel1.cl)

#### 4.2.6 Création du programme avec le fichier noyau et le contexte

Il faut créer le noyau de calcul dans notre fonction *main*. On doit créer un noyau avec le programme et on peut commencer avec les deux fonctions suivantes :

```
cl_program clCreateProgramWithSource(cl_context context,
                                    cl_uint count,
                                    const char **strings,
                                    const size_t *lengths,
                                    cl_int *errcode_ret)

cl_int clBuildProgram(cl_program program,
                     cl_uint num_devices,
                     const cl_device_id *device_list,
                     const char *options,
                     void (CL_CALLBACK * pfn_notify)
                     (cl_program program,
                      void *user_data),
                     void *user_data)
```

FIGURE 4.18 – prototype des fonctions de construction de programmes

De manière plus détaillée, Les étapes pour créer et construire un programme sont :

```
std::ifstream srcFile("kernel1.cl");
checkErr(srcFile.is_open() ? CL_SUCCESS : -1, "reading kernel1.cl");
std::string srcProg(std::istreambuf_iterator<char>(srcFile), (std::istreambuf_iterator<char>()));
const char *src = srcProg.c_str();
size_t length = srcProg.length();
program = clCreateProgramWithSource(context, 1, &src, &length, &errnum);
checkErr(errnum, "create program with source");

errnum = clBuildProgram(program, 1, deviceIds, NULL, 0, 0);
checkErr(errnum, "build program");
```

FIGURE 4.19 – Détail de l’implémentation de la construction du programme

### 4.2.7 Création du noyau avec programme construit

Une fois le programme construit, on obtient un fichier *kernel1.cl*, il ne reste qu'à simplement créer le noyau de calcul grâce à la fonction *clCreateKernel*.

```
cl_kernel clCreateKernel(cl_program program,
                        const char * kernel_name,
                        cl_int * errcode_ret)

kernel = clCreateKernel(program, "hello_kernel", NULL);
if (kernel == NULL) {
    cerr << "fail to create kernel" << endl;
}
```

FIGURE 4.20 – Prototype et implémentation de la fonction de création de noyaux

### 4.2.8 Insertion des données dans le noyau

Maintenant, le noyau de calcul est créé. Il faut alors entrer les données dans le noyau pour faire le calcul. On doit utiliser les buffers pour réaliser le transmission de données entre la fonction main et le noyau.

```
float result[ARRAY_SIZE];
float a[ARRAY_SIZE];
float b[ARRAY_SIZE];
for (int i = 0; i < ARRAY_SIZE; i++) {
    a[i] = i;
    b[i] = i * 2;
}

cl_mem a_buffer = clCreateBuffer(context, CL_MEM_READ_WRITE | CL_MEM_USE_HOST_PTR, ARRAY_SIZE * sizeof(float), a, &errnum);
cl_mem b_buffer = clCreateBuffer(context, CL_MEM_READ_WRITE | CL_MEM_USE_HOST_PTR, ARRAY_SIZE * sizeof(float), b, &errnum);
cl_mem result_buffer = clCreateBuffer(context, CL_MEM_READ_WRITE | CL_MEM_USE_HOST_PTR, ARRAY_SIZE * sizeof(float), result, &errnum);
checkErr(errnum, "setting buffers");

errnum = clSetKernelArg(kernel, 0, sizeof(cl_mem), &a_buffer);
errnum = clSetKernelArg(kernel, 1, sizeof(cl_mem), &b_buffer);
errnum = clSetKernelArg(kernel, 2, sizeof(cl_mem), &result_buffer);
checkErr(errnum, "setting kernel arguments");
```

FIGURE 4.21 – Prototype et implémentation de la fonction de création de noyaux

La fonction *clCreateBuffer* crée les buffers dans la mémoire qui peuvent stocker les données. Et la fonction *clSetKernelArg* permet d'assigner un buffer au noyau.

### 4.2.9 Exécution des calculs avec les dispositifs ciblés

Avant de commencer le calcul, il faut choisir un dispositif de calcul dans notre contexte courant. Ici, on choisira le dispositif *GPU* qui peut calculer en parallèle.

```
commandQueue = clCreateCommandQueue(context, deviceIds[0], 0, NULL);
```

FIGURE 4.22 – Choix du dispositif de calcul

On appelle la fonction *clEnqueueNDRangeKernel* qui peut exécuter le calcul par le noyau avec le dispositif choisi.

```
size_t globalWorkSize[1] = { ARRAY_SIZE };
size_t localWorkSize[1] = { 1 };
errnum = clEnqueueNDRangeKernel(commandQueue, kernel, 1, NULL, globalWorkSize, localWorkSize, 0, NULL, NULL);
checkErr(errnum, "queuing kernel for execution");
```

FIGURE 4.23 – Lancement du calculs

### 4.2.10 Exportation des résultats

Après exécution, le résultat est stocké dans le buffer *result\_buffer*. Il faut donc le lire avec la fonction *clEnqueueReadBuffer*.

```
errnum = clEnqueueReadBuffer(commandQueue, result_buffer, CL_TRUE, 0, ARRAY_SIZE*sizeof(float), result, 0, NULL, NULL);
checkErr(errnum, "reading result buffer");
```

FIGURE 4.24 – Récupération des résultats sur le buffer de sortie

On a réussi à faire le calcul par *GPU*, si le tableau est grand, la vitesse de calcul par les dispositifs parallèles (*GPU, FPGA...*) est beaucoup plus rapidement que le calcul par *CPU*.

```
for (int i = 0; i < ARRAY_SIZE; i++) {
    cout << result[i] << " ";
}

cout << endl;
cout << "tout marche bien!" << endl;
int fini;
cin >> fini;
return 0;
```

0 3 6 9 12 15 18 21 24 27  
tout marche bien!

FIGURE 4.25 – Affichage du résultat

## 4.3 Vivado-HLS

### 4.3.1 Introduction

*Vivado* est un logiciel créé par l'entreprise *Xilinx* pour développer les programmes sur *FPGA*. Avec *vivado-HLS*, on peut programmer sur *FPGA* avec les langages *C, C++*. En vérité, le code écrit par l'humain sera du *C, C++* mais il y a un trans-compileur qui générera le code pour *VHDL, Verilog*. Pour notre étude, on a besoin d'opérer sur des matrices avec *FPGA*, car programmer avec *VHDL* directement peu devenir assez compliqué, mais faire un programme pour calcul matriciel avec le langage *C, C++* est sans difficulté [22].

Le code généré n'est optimisé que pour les cartes de *Xilinx*.

### 4.3.2 Fichiers source et examen

Ici on peut prendre un exemple très simple : Réaliser un mux21 avec *vivado-HLS*. Une fois le projet créé sur *vivado-HLS*, on peut voir :

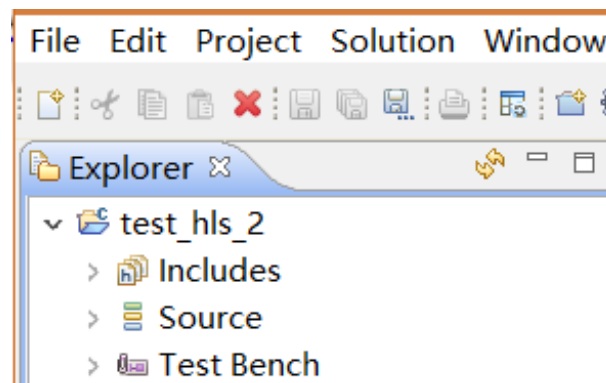


FIGURE 4.26 – Interface de projet Vivado-HLS

La première étape est d'ajouter les fichiers sources dans le projet. On a ici *mux21.c*

```

1  #include "mux21.h"
2
3  int1 mux21(int1 sig_a, int1 sig_b, int1 select)
4  {
5      if(0==select)
6          return sig_a;
7      else
8          return sig_b;
9  }
  
```

FIGURE 4.27 – Implémentation du mux21 en C

La fichier "mux21.h" est comme suite :

```

1  #include <ap_cint.h>
  
```

FIGURE 4.28 – Définitions de l'entête du mux21

Avec le fichier *ap\_cint.h*, on peut utiliser le type *int1*, c'est-à-dire les variables ont seulement un bit. Il faut seulement ajouter le fichier *mux21.c* dans le projet. les fichiers d'en-tête(.h) vont être ajoutés automatiquement.

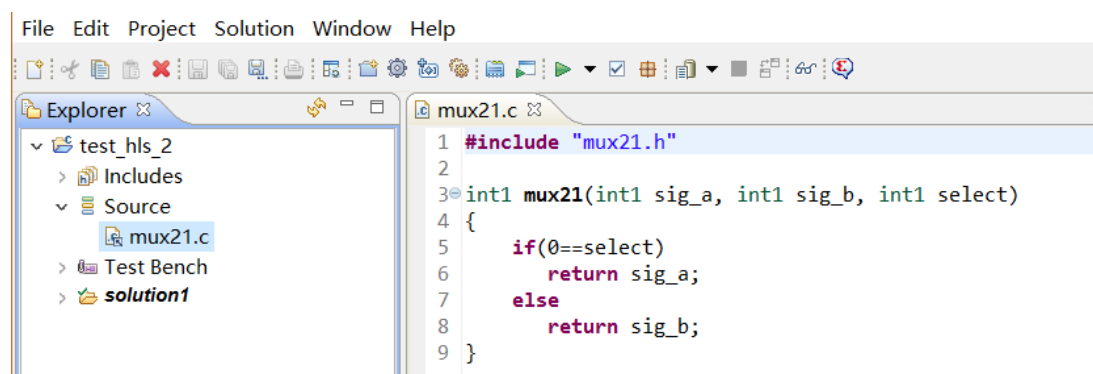
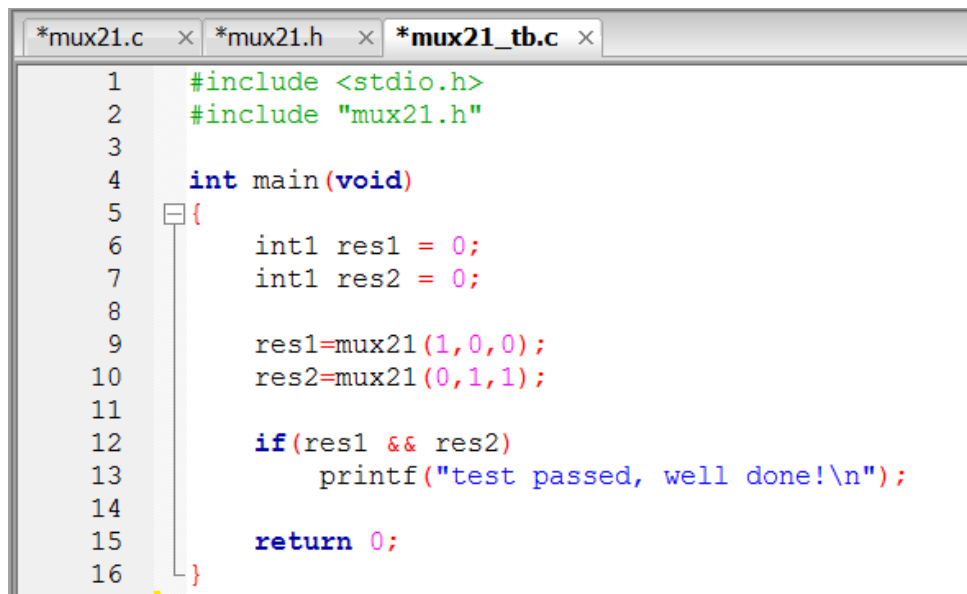


FIGURE 4.29 – Ajout de source au projet

La deuxième étape est d'ajouter les fonctions de examen dans *Test Bench*, on peut créer le fichier *mux21\_tb.c*



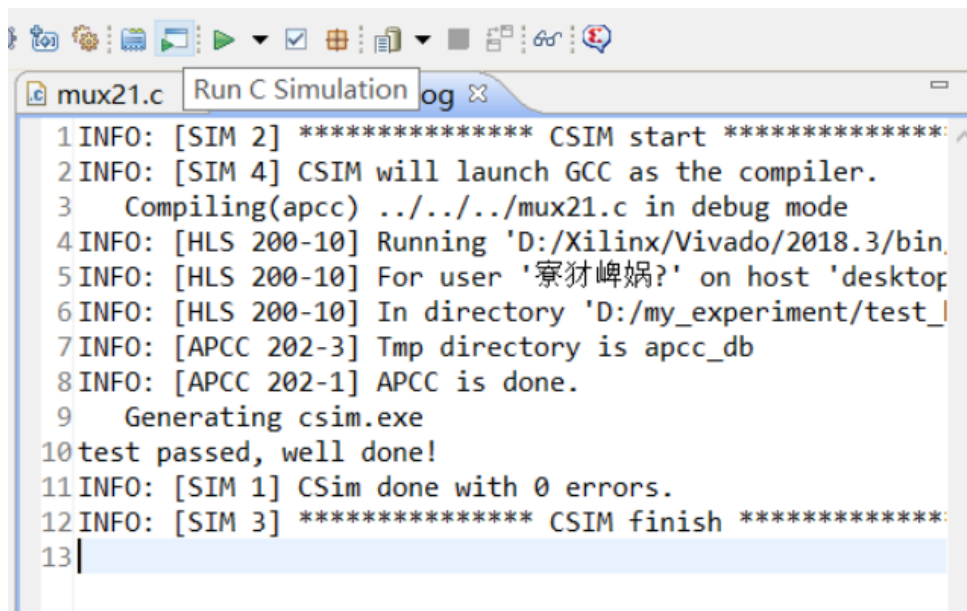
```

1  #include <stdio.h>
2  #include "mux21.h"
3
4  int main(void)
5  {
6      int1 res1 = 0;
7      int1 res2 = 0;
8
9      res1=mux21(1,0,0);
10     res2=mux21(0,1,1);
11
12     if(res1 && res2)
13         printf("test passed, well done!\n");
14
15     return 0;
16 }

```

FIGURE 4.30 – Implémentation du fichier d'examen

Dans la fonction d'examen, on peut donner différents nombres pour les entrées du mux21, et on va regarder si la valeur de sortie est sensée. Dans la fonction d'examen, on peut donner différent nombres pour les entrées du mux21, et on va regarder si la valeur de sortie est sensé.



```

1 INFO: [SIM 2] ***** CSIM start *****
2 INFO: [SIM 4] CSIM will launch GCC as the compiler.
3   Compiling(apcc) ../../../mux21.c in debug mode
4 INFO: [HLS 200-10] Running 'D:/Xilinx/Vivado/2018.3/bin.
5 INFO: [HLS 200-10] For user '寮豺崙?' on host 'desktop
6 INFO: [HLS 200-10] In directory 'D:/my_experiment/test_|
7 INFO: [APCC 202-3] Tmp directory is apcc_db
8 INFO: [APCC 202-1] APCC is done.
9   Generating csim.exe
10 test passed, well done!
11 INFO: [SIM 1] CSim done with 0 errors.
12 INFO: [SIM 3] ***** CSIM finish *****
13

```

FIGURE 4.31 – Résultat d'une simulation en C

Si le fichier de source et le fichier d'examen sont corrects, on peut commencer "Run C Simulation", c'est-à-dire exécuter la fonction dans le fichier d'examen. On peut observer le résultat. (ligne 10 :test passed,well done!).

Dans des cas de plus compliqué, il faut utiliser beaucoup de valeurs différentes pour tester le fichier.c.



### 4.3.3 Génération du fichier VHDL

Pour créer les fonctions *VHDL* par le fichier de source, on peut utiliser *Active Solution*

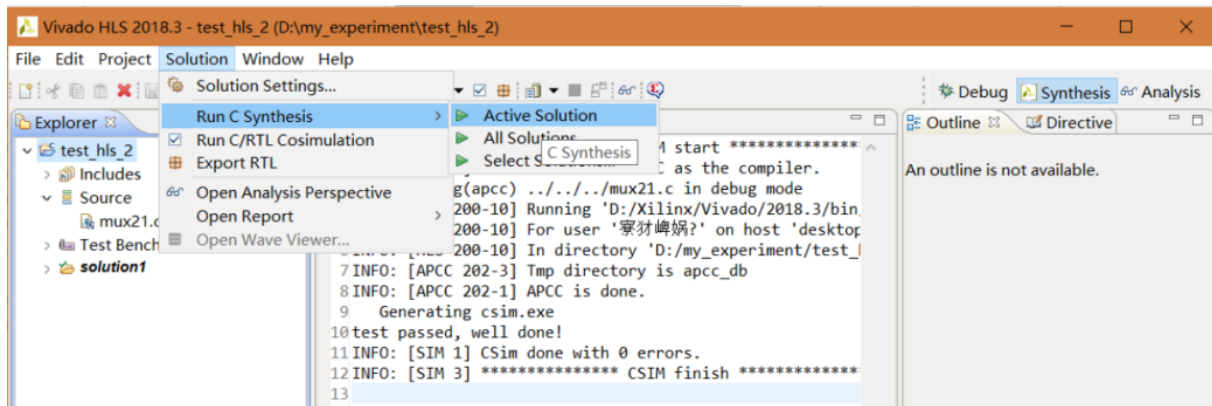


FIGURE 4.32 – Génération du fichier VHDL

Après *Active Solution*, il est créé un répertoire “solution 1”, en plus, on peut trouver le fichier *solution 1/syn/vhdl/mux21.vhd*,

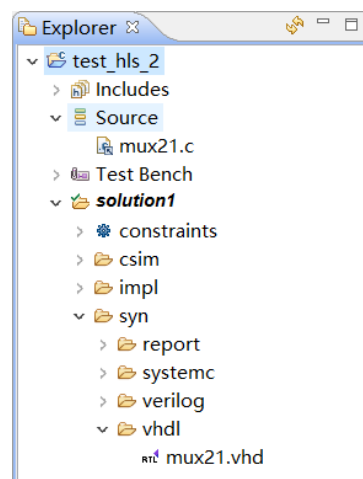


FIGURE 4.33 – Présentation de l’arborescence de la solution 1

Le fichier *mux21.vhd* est le programme *VHDL* généré à partir du fichier *C*.

```

1-- =====
2-- RTL generated by Vivado(TM) HLS - High-Level Synthesis from C, C++ and SystemC
3-- Version: 2018.3
4-- Copyright (C) 1986-2018 Xilinx, Inc. All Rights Reserved.
5-- =====
6-- =====
7
8library IEEE;
9use IEEE.std_logic_1164.all;
10use IEEE.numeric_std.all;
11
12entity mux21 is
13port (
14    ap_start : IN STD_LOGIC;
15    ap_done : OUT STD_LOGIC;
16    ap_idle : OUT STD_LOGIC;
17    ap_ready : OUT STD_LOGIC;
18    sig_a : IN STD_LOGIC_VECTOR (0 downto 0);
19    sig_b : IN STD_LOGIC_VECTOR (0 downto 0);
20    select_r : IN STD_LOGIC_VECTOR (0 downto 0);
21    ap_return : OUT STD_LOGIC_VECTOR (0 downto 0) );
22end;
23
24
25architecture behav of mux21 is
26    attribute CORE_GENERATION_INFO : STRING;
27    attribute CORE_GENERATION_INFO of behav : architecture is
28        "mux21,hls_ip_2018_3,{HLS_INPUT_TYPE=c,HLS_INPUT_FLOAT=0,HLS_INPUT_FIXED=1,HLS_INPUT_PART=xa7a12tcsg325-1q,HLS_INPUT_CLOCK=10
29    constant ap_const_logic_1 : STD_LOGIC := '1';
30    constant ap_const_logic_0 : STD_LOGIC := '0';
31    constant ap_const_boolean_1 : BOOLEAN := true;
32
33
34
35begin
36
37
38
39    ap_done <= ap_start;
40    ap_idle <= ap_const_logic_1;
41    ap_ready <= ap_start;
42    ap_return <=
43        sig_b when (select_r(0) = '1') else
44        sig_a;
45end behav;
46

```

FIGURE 4.34 – Fichier VHDL généré

Le mux21 est super simple donc il y a beaucoup de signal qu'on n'a pas utiliser. Mais on peut regarder uniquement la partie fonctionnel. C'est plus clair :

```

42    ap_return <=
43        sig_b when (select_r(0) = '1') else
44        sig_a;

```

FIGURE 4.35 – Partie fonctionnel du mux21 en VHDL

On a généré un programme VHDL de mux21, de plus, il y a beaucoup d'information qui est noté dans le fichier.rpt. On peut trouver *“General Information”*, *“Performance Estimates”*, *“Utilization Estimates”* et *“Interface”* du mux21.

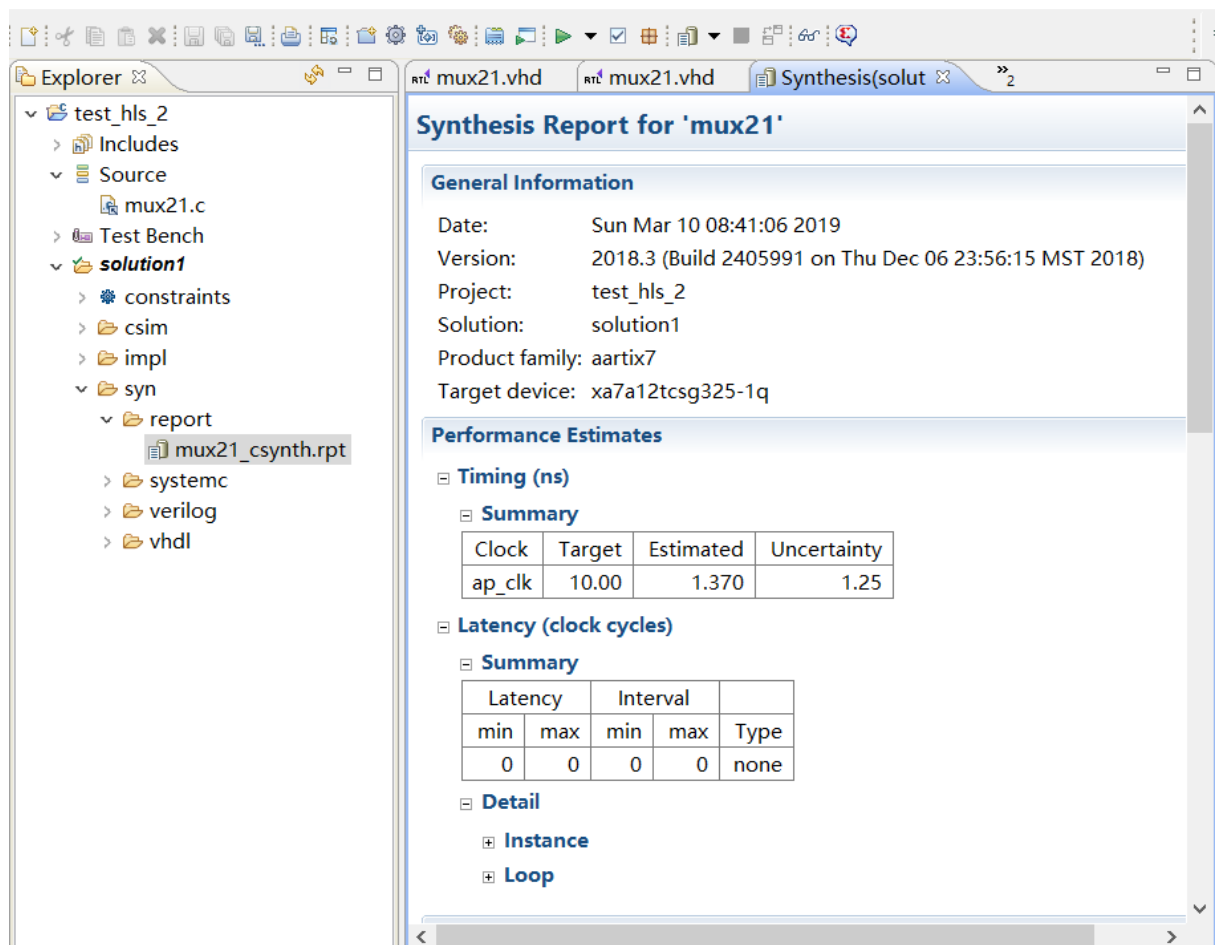


FIGURE 4.36 – Interface des information complémentaires

## 4.4 Conclusion

Dans ce chapitre on a pu voir comment utiliser deux outils très puissants, l'un pour la conception de programme parallèle pour plusieurs plateformes constructeurs et dispositifs associés ; L'autre pour la génération de code pour carte ciblée.

Les programmes implémentés sont très simples car ce genre d'outils peut rapidement devenir très compliqué à gérer. On commence par des exemples très simples afin de pouvoir étendre le raisonnement au programme de plus grande complexité tels que les programmes de traitement d'image et calcul numérique.

## Conclusion générale

Dans ce rapport, nous avons pu approcher une problématique qui semble très compliquée au premier abord. Les architectures hétérogènes se différencient des architectures homogènes par la nature de leurs noyaux et ont donc des applications beaucoup plus ciblées. Ces architectures se différencient elles même en trois sous catégories : noyaux identiques mais avec configurations différentes, noyaux à capacités architecturales différentes, et différence de modèles d'exécution (*MIMD*, *SIMD*, ...). Les normes autour de ces architectures ne cessent d'évoluer, et il n'est pas si En effet, on a aussi pu constaté que les architectures hétérogènes étaient assez compliquées à comprendre et à concevoir mais le fait qu'elle soit autant utilisée de nos jours montre que c'est un marché en essor et ceci ne risque pas de s'arrêter dans un avenir proche.

Nous avons par la suite décortiqué la méthodologie derrière la conception d'algorithme parallèle dans l'optique de pouvoir en concevoir pour des architectures hétérogènes ciblées. Cette conception requiert plusieurs pré-connaissances sur l'architecture des ordinateurs et leur classification ainsi que sur les différentes configuration mémoire/bus/contrôleurs. Ainsi, on a présenté une méthodologie de conception d'algorithme parallèle et vu l'omniprésence de ces algorithmes dans quasiment tous les domaines de recherches et de l'industrie ; on peut dire que faire des algorithmes non-parallèles est quasiment obsolète. On remarque alors une tendance dual entre ces algorithmes et les architectures hétérogènes car elles évoluent suivant la même courbe.

Le traitement d'image représente une des applications classiques où le traitement rapide de grandes quantités de donnée est primordial. Et c'est là où excelle un algorithme parallèle implémentée sur une architecture dédiée à ce traitement basé sur un *CPU* et *FPGA*. Cette implémentation n'est pas que théorique. Elle est belle est bien mise en place par plusieurs systèmes embarqués tel que la détection d'obstacles pour les voitures sans conducteur.

Enfin, on a présenté deux outils *OpenCL* et *Vivado-HLS*. Le premier est un ensemble de bibliothèques qui facilitent grandement la parallélisation d'algorithme et leur implémentation sur des structures hétérogènes physiques. Il possède une variété d'outils permettant d'exploiter les données et de mesurer les performances de ces systèmes. Le deuxième, *Vivado-HLS* de *Xilinx*, lui permet de générer un code *VHDL* optimisé pour les cartes de *Xilinx*. Cet outil confère un confort pour le programmeur qui n'a plus à penser à la synthèse du circuit sur la carte car il n'a qu'à implémenter un code *C/C++* et l'outil arrive à le convertir de la meilleur manière pour les cartes dédiées.

## Perspectives

- Découvrir d'autres outils dédiés à la conception d'architecture hétérogène : *LegUp*, *Gauche*, *Intel-HLS*...
- Explorer plus une plus grande variété de noyau : *DSP*, *AP*...
- Produire des programmes parallèles sur différentes outils et mener les tests de comparaison nécessaires concernant les performances et la portabilité.
- Évaluer la qualité du code *VHDL* généré par les différents transcompilateurs.
- Approfondir les recherches sur les problématiques de complexité rencontrés.
- Examiner les étapes de contrôles inter-noyau : bus, contrôleur.

# Bibliographie

- [1] Donald Firesmith. Multicore processing. [https://insights.sei.cmu.edu/sei\\_blog/2017/08/multicore-processing.html](https://insights.sei.cmu.edu/sei_blog/2017/08/multicore-processing.html), Août 21, 2017.
- [2] Mohamed Zahran. Heterogeneous computing here to stay, 10 janvier 2017.
- [3] Microsoft. Project catapult. <https://www.microsoft.com/en-us/research/project/project-catapult/>, 2016.
- [4] Ke Wang, Kevin Angstadt, Chunkun Bo, Nathan Brunelle, Elaheh Sadredini, Tommy Tracy II, Jack Wadden, Mircea Stan, and Kevin Skadron. An overview of micron's automata processor. [https://www.cs.virginia.edu/~skadron/Papers/wang\\_APOverview\\_CODES16.pdf](https://www.cs.virginia.edu/~skadron/Papers/wang_APOverview_CODES16.pdf), Oct 7, 2016.
- [5] HSA Foundation. Website. <http://www.hsafoundation.com/>.
- [6] Ananth Grama, Anshul Gupta, Geogre Karpis, and Vpin Kumar. *Introduction to Parallel Computing*.
- [7] G. Brassard and P. Bratley. *Fundamental of Algorithmics*. 1996.
- [8] Marti Milo. Performance and benchmarking. [http://www.cis.upenn.edu/~milom/cis501-Fall12/lectures/04\\_performance.pdf](http://www.cis.upenn.edu/~milom/cis501-Fall12/lectures/04_performance.pdf).
- [9] Joel F. Klein. Amdahl's law. [https://commons.wikimedia.org/wiki/Category:Amdahl's\\_law](https://commons.wikimedia.org/wiki/Category:Amdahl's_law).
- [10] Jack Dongarra A.J. van der Steen. *Overview of Recent Supercomputers*.
- [11] top500. <https://www.top500.org/>.
- [12] Michael J Flynn. *Computer architecture : pipelined and parallel processor design*. MA : Jones and Bartlett.
- [13] Hennessy Patterson. *Computer Architecture*.
- [14] Patterson and David A. and. *L. Computer Architecture : A Quantitative Approach (4th ed.)*. 4 edition.
- [15] F. Darema. Spmd model : past, present and future. e, recent advances in parallel virtual machine and message.
- [16] Lan Foster. *Designing and Building Parallel Programs*.
- [17] Mikko H. Lipasti Shen, John Paul. *fundamentals of superscalar processors*. McGraw-Hill, 1 edition.
- [18] Brandon Treece. Cpu or fpga for image processing : Which is best ? <https://forums.xilinx.com/>.
- [19] Benedict R.Gaster, Lee Howes, David R.Kaeli, Perhaad Mistry, and Dana Schaa. Heterogeneous computing with opencl.
- [20] Aaftab Munshi, Benedit R.Gaster, Timothy G.Mattson, James Fung, and Dan Ginsburg. Opencl programming guide.

[21] Khronos Group. Opengl 1.1 reference pages.

[22] GAO Yajun. To learn vivado from here.