Méthodes d'instrumentation et d'optimisation d'applications élaborées en OpenCL mise en œuvre sur des plateformes hétérogènes reconfigurables

par

Hachem BENSALEM

THÈSE PRÉSENTÉE À L'ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
COMME EXIGENCE PARTIELLE À L'OBTENTION DU
DOCTORAT EN GÉNIE
Ph.D.

MONTRÉAL, LE 15 OCTOBRE 2021

ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
UNIVERSITÉ DU QUÉBEC

PRÉSENTATION DU JURY

CETTE THÈSE A ÉTÉ ÉVALUÉE

PAR UN JURY COMPOSÉ DE :

M. Yves Blaquière, directeur de thèse
Département de génie électrique à l'École de technologie supérieure

M. Yvon Savaria, codirecteur de thèse
Département de génie électrique à Polytechnique Montréal

M. Éric Granger, président du jury
Département de génie des systèmes à l'École de technologie supérieure

M. Pascal Giard membre du jury
Département de génie électrique à l'École de technologie supérieure

M. Amine Miled, examinateur externe
Département de génie électrique et de génie informatique à l'université Laval

ELLE A FAIT L'OBJET D'UNE SOUTENANCE DEVANT JURY ET PUBLIC

LE 30 SEPTEMBRE 2021

À L'ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

REMERCIEMENTS

J’aimerais exprimer ma profonde gratitude et mes sincères remerciements à mon directeur de thèse, monsieur Yves Blaquière, de m’avoir guidé et conseillé durant mes travaux de recherche. Je tiens aussi à lui remercier pour ses conseils, sa disponibilité et le support financier qu’il m’a offert durant ces années d’études. Ses conseils, son expertise ainsi que le support moral qu’il m’a offert étaient essentiels à la réalisation de cette thèse. Je remercie profondément le Professeur Yves Blaquière pour l’humain qu’il est.

Je tiens aussi à remercier profondément mon codirecteur, Professeur Yvon Savaria pour ses conseils précieux. Je suis reconnaissant de ses contributions qui ont mené à la réalisation de cette thèse.

Je remercie également les membres de jury, messieurs Éric Granger, Pascal Giard et Amine Miled pour m’avoir fait l’honneur d’accepter l’évaluation de mon travail.

Je tiens aussi à remercier Normand Gravel, mes collègues et membres du laboratoire Lacime ainsi que ceux côtoyés à l’ÉTS.

Je remercie également mes parents, mes frères et sœurs pour leur support.

Méthodes d'instrumentation et d'optimisation d'applications élaborées en OpenCL mise en œuvre sur des plateformes hétérogènes reconfigurables

Hachem BENSALEM

RÉSUMÉ

Le besoin de plateformes de calcul de haute performance s'est récemment intensifié avec l'émergence de nouvelles applications, telles l'intelligence artificielle et la science de données. De nouvelles plateformes hétérogènes reconfigurables sont ainsi déployées avec un potentiel d'accélération de calcul nettement supérieur à celles basées sur une architecture multiprocesseur. Ces plateformes se caractérisent par la diversité croissante de leurs éléments de calcul tels des processeurs multi-noyaux (*central processing unit* (CPU)), des processeurs spécialisés pour le traitement graphique (*graphics processing unit* (GPU)) et des unités de calcul reconfigurables de type FPGA (*field-programmable gate array*). Les fournisseurs de FPGA offrent maintenant des environnements logiciels de conception pour supporter cette diversité architecturale, appelés outils de synthèse de haut niveau (*high-level synthesis* (HLS) *tools*). En revanche, la performance d'une application créée par ces outils HLS est fortement dépendante de sa description de haut niveau, généralement peu compétitive par rapport à celle obtenue lorsque décrite avec des langages de description de matériel opérant à bas niveau. De plus, cette dégradation de performance est difficile à analyser et diagnostiquer, puisque ces outils HLS n'offrent pas un mécanisme permettant d'analyser les causes de cette dégradation comme ceux qui existent pour les CPU et GPU.

L'objectif principal des travaux présentés dans cette thèse consiste à améliorer la productivité de développement d'applications sur les plateformes hétérogènes reconfigurables intégrant CPU et FPGA. Ils visent en particulier à améliorer l'observation d'événements temporels dans les accélérateurs élaborés en OpenCL ouvrant la voie vers le diagnostic précis des pertes de performance, comme le décrochage du pipeline. Un autre objectif spécifique consiste à faciliter l'élaboration de méthodologies d'optimisation pour améliorer la performance des accélérateurs sur FPGA décrits en OpenCL.

La première contribution de cette thèse propose une solution pour l'analyse in-situ de performances temporelles d'un noyau OpenCL implémenté sur FPGA en intégrant un moniteur développé avec un langage de description matériel. Ce moniteur peut facilement être intégré dans des noyaux OpenCL grâce à une approche de conception matérielle/logicielle. Il peut mesurer les évènements dans le FPGA avec une précision au cycle d'horloge du FPGA. La deuxième contribution exploite cette nouvelle solution d'instrumentation par la proposition d'une plateforme qui permet d'extraire les performances temporelles des noyaux OpenCL. Une modélisation théorique est proposée pour identifier le nombre d'instruments à insérer afin de calculer précisément les performances temporelles d'une application. Au meilleur de nos connaissances, cette plateforme d'instrumentation est la première qui propose la mesure des performances temporelles tels les latences et les intervalles d'initiation de boucles d'une application sur FPGA. Les résultats ont montré que cette plateforme permet d'identifier les causes de dégradation de performances.

Les troisième et quatrième contributions présentent les assises vers un outil d'optimisation automatique pour les accélérateurs de type FPGA élaborés en OpenCL. En effet, des techniques d'optimisation basées sur OpenCL sont proposées et elles sont appliquées sur les algorithmes du hachage (*secure hash algorithm* (SHA)) SHA-2 et SHA-3. Les résultats ont montré que les implémentations du SHA-2 et SHA-3 offrent un débit de traitement de données de 3.9 Gbps et de 22.36 Gbps respectivement, qui sont 4.3 fois et 2 fois plus élevées que les meilleures publiées dans la littérature implémentées par des outils HLS.

Mots-clés : FPGA, OpenCL, HLS, instrumentation, optimisation, hétérogène reconfigurable, SHA-2, SHA-3.

Framework and methods for the instrumentation and the optimization for OpenCL-based designs on heterogenous and reconfigurable platforms

Hachem BENSALEM

ABSTRACT

The need for high-performance computing platforms has recently intensified with the emergence of new applications, such as artificial intelligence and data science. New reconfigurable heterogeneous platforms are thus deployed with a much greater potential of computational acceleration than those based on a multiprocessor architecture. These platforms are powered with several computing elements such as multi-core processing units (CPUs), specialized processors for graphic processing (GPUs) and field-programmable gate arrays (FPGAs). FPGA vendors now offers high-level synthesis (HLS) tools to support this architectural diversity. On the other hand, the performance of an application created by these HLS tools is strongly dependent on its high-level description, which is generally less competitive with that described with low-level hardware description languages. Moreover, this performance degradation is difficult to analyze and diagnose, since these HLS tools do not provide a mechanism to analyze the causes of this degradation like those that exist for CPUs and GPUs.

The main objective of the work presented in this thesis is to improve the productivity of application development on FPGA-based reconfigurable heterogeneous platforms. This work aims to improve the observation of temporal events, in FPGA-based accelerators described in OpenCL, paving the way for the precise diagnosis of performance losses such as pipeline stalls. Another specific objective is to ease the elaboration of optimization methodologies to improve the performance of FPGA-based accelerators described in OpenCL.

The first contribution of this thesis proposes a solution for the in-situ analysis of timing performance of an OpenCL kernel implemented in FPGA by integrating a monitor developed with a hardware language. This monitor can easily be integrated into OpenCL kernels through hardware/software design approach. It can then extract runtime information at FPGA clock cycle accuracy. The second contribution exploits this new instrumentation method by proposing a framework to extract the timing performance of OpenCL kernels. A theoretical model is proposed to identify the number of instruments to be inserted to precisely compute the timing performance of an application. To the best of our knowledge, the proposed instrumentation framework is the first framework that proposes the measurement of timing performance such as latency and initiation intervals of loops and to identify performance bottlenecks of OpenCL-based designs on FPGA.

The third and fourth contributions of this thesis open the door to design an automatic optimization tool and methodology for OpenCL-based designs on FPGA. Indeed, OpenCL-based optimization techniques are proposed to accelerate OpenCL-based designs and they are applied to the secure hash algorithms (SHA) SHA-2 and SHA-3. The results showed that

implementations of SHA-2 and SHA-3 derived from OpenCL descriptions offer throughputs of 3.9 Gbps and 22.36 Gbps respectively, which are 4.3 times and 2 times higher than the best previously published HLS-based designs.

Keywords : FPGA, OpenCL, high-level synthesis, instrumentation, optimization, heterogenous reconfigurable platforms, SHA-2, SHA-3.

TABLE DES MATIÈRES

Page

INTRODUCTION 1

CHAPITRE 1 ÉTAT DE L'ART DES MÉTHODES D'INSTRUMENTATION ET D'OPTIMISATION DES APPLICATIONS ÉLABORÉES EN OPENCL ET MISES EN ŒUVRE SUR DES PLATEFORMES HÉTÈROGÈNES RECONFIGURABLES 8

1.1 Architecture hétérogène reconfigurable 9

1.1.1 Définition 9

1.1.2 Historique et motivation 9

1.1.3 Plateformes hétérogènes reconfigurables intégrant des FPGA 10

1.1.4 Problématique et défis des plateformes hétérogènes reconfigurables 12

1.2 Méthodes de conception sur des plateformes hétérogènes reconfigurables 12

1.2.1 Synthèse de haut niveau 18

1.2.1.1 Historique et évolution de la HLS 18

1.2.1.2 Flot de conception basé sur la HLS 19

1.2.1.3 Exploration de l'espace de conception 23

1.2.1.4 Défis et limitations des outils HLS 25

1.2.2 Langage de programmation OpenCL 27

1.2.2.1 Structure générale d'une description d'application en OpenCL 28

1.2.2.2 Noyaux OpenCL 29

1.2.2.3 Modèles de mémoires 30

1.3 Méthodes d'instrumentation et d'analyse de performances sur FPGA 31

1.3.1 Instrumentation par analyseurs logiques (p. ex. ChipScope, SignalTap) 33

1.3.2 Instrumentation par modification du flot de compilation 34

1.3.3 Instrumentation par transformation du code 35

1.4 Accélération des algorithmes de hachage « Secure Hash Algorithm (SHA) » sur des plateformes reconfigurables 36

1.4.1 SHA-2 37

1.4.2 SHA-3 40

1.5 Contributions de la thèse : Méthodes d'instrumentation et d'optimisation des accélérateurs sur FPGA élaborés en OpenCL 44

1.6 Conclusion 47

CHAPITRE 2 MÉTHODOLOGIE ET PLATEFORME D'INSTRUMENTATION PROPOSÉE POUR DES ACCÉLÉRATEURS SUR FPGA ÉLABORÉE EN OPENCL 49

2.1 Problématiques et défis 49

2.2 Développement d'un moniteur au niveau RTL 50

2.2.1 Concept général d'instrumentation des noyaux OpenCL 51

2.2.2 Intégration d'un moniteur développé au niveau RTL 52
2.2.2.1 Architecture RTL du moniteur 52
2.2.2.2 Interface de communication entre le moniteur et le noyau OpenCL 53
2.2.2.3 Flot de conception d'OpenCL pour FPGA 54
2.2.2.4 Plateforme CPU+FPGA 54
2.2.3 Étude de cas : Multiplicateur-accumulateur (MAC) 55
2.2.3.1 Architecture et implémentation du MAC 55
2.2.3.2 Résultats 56
2.2.3.3 Comparaison avec un moniteur OpenCL 59
2.2.3.4 Discussion 60
2.3 Plateforme d'instrumentation pour les accélérateurs de type FPGA élaborés en OpenCL 61
2.3.1 Vue d'ensemble d'un noyau avec instruments 61
2.3.2 Étapes d'instrumentation d'un noyau OpenCL 62
2.3.3 Modélisation de performances des noyaux OpenCL 63
2.3.3.1 Modèle de performance d'un seul noyau 63
2.3.3.2 Modèle de performance d'une structure multi-noyaux 66
2.3.4 Implémentation de la plateforme d'instrumentation 68
2.3.4.1 Partie matérielle : Architecture RTL de l'instrument 68
2.3.4.2 Partie logicielle 70
2.3.4.3 Implémentation de la bibliothèque d'instrumentation 70
2.3.5 Instrumentation et résultats 72
2.3.5.1 Instrumentation d'un seul noyau 72
2.3.5.2 Instrumentation d'une structure multi-noyaux 77
2.3.6 Évaluation de la plateforme d'instrumentation 80
2.3.7 Discussion 83

CHAPITRE 3 OPTIMISATION D'APPLICATIONS ÉLABORÉES EN OPENCL MISE EN ŒUVRE SUR DES PLATEFORMES HÉTÉROGÈNES RECONFIGURABLES 85
3.1 Les techniques d'optimisation des noyaux OpenCL pour une cible FPGA 85
3.1.1 Motivation 85
3.1.2 Modèles d'exécution des noyaux OpenCL 86
3.1.3 Optimisation par l'insertion des mémoires locales 86
3.1.4 Optimisation des opérations de chargement et de stockage (load/store) 87
3.1.5 Optimisations des boucles 87
3.1.5.1 Pipelinage des boucles 88
3.1.5.2 Fractionnement des boucles (*Loop splitting*) 89
3.1.5.3 Déroulement des boucles 89
3.1.5.4 L'optimisation des boucles imbriquées 90
3.1.6 Directives spécifiques « *attribute* » 90
3.2 Optimisation de l'algorithme SHA-2 (SHA-256) 91
3.2.1 Version de base du SHA-256 91

3.2.2 Métriques de performance 92
3.2.3 Les différentes implémentations des noyaux du SHA-256..... 93
3.2.4 Implémentations et résultats 95
3.3 Optimisation de l'algorithme SHA-3.....98
3.3.1 Les différentes versions du noyau SHA-3 98
3.3.2 Implémentations et résultats 101
3.4 Conclusion 104

CHAPITRE 4 DISCUSSION GÉNÉRALE.....105
4.1 Résumé des contributions 105
4.2 Comparaison de la plateforme d'instrumentation proposée 106
4.3 Comparaison de l'optimisation des implémentations OpenCL du SHA-2 et SHA-3 109
4.3.1 Optimisation du SHA-2 109
4.3.2 Optimisation du SHA-3 110
4.4 Discussion et travaux futurs recommandés.....112

CONCLUSION 116

LISTE DE RÉFÉRENCES BIBLIOGRAPHIQUES.....119

LISTE DES TABLEAUX

Page

Tableau 1.1 Spécification du SHA-3 41

Tableau 2.1 Temps d'exécution du noyau *single work-item* (ms) 59

Tableau 2.2 Ressources utilisées sur FPGA avec différents nombres de moniteurs intégrés (% est l'augmentation par rapport au noyau sans moniteurs) 60

Tableau 2.3 Comparaison des fonctionnalités des moniteurs 60

Tableau 2.4 Performances temporelles exprimées en cycles d'horloge du FPGA et mesurées avec 50 k éléments (charge de 200 Ko) de deux modèles d'exécution 76

Tableau 2.5 Ressources logiques utilisées par les noyaux avec 4 instruments 76

Tableau 2.6 Temps d'exécution (ms) des noyaux SWI (Figure 2.16) 77

Tableau 2.7 Surcharge de ressources utilisées sur FPGA de la structure multi-noyaux avec 6 instruments insérés (Figure 2.18) 79

Tableau 2.8 Quantité de ressources logiques utilisées sur FPGA pour les 17 benchmarks sans insertion d'instruments 80

Tableau 3.1 Directives permettant les transformations des boucles 88

Tableau 3.2 Les directives spécifiques au compilateur de Intel pour OpenCL 91

Tableau 3.3 Les neuf versions du SHA-256 auxquels s'appliquent les techniques d'optimisation 95

Tableau 3.4 Performances pour 2 Mo d'éléments 96

Tableau 3.5 Les différentes versions du noyau du SHA-3 101

Tableau 3.6 Les performances des différentes implémentations du SHA-3 103

Tableau 4.1 Comparaison de l'utilisation des ressources logique par instrument (%) 108

Tableau 4.2 Les FPGA utilisés par les différents travaux 108

Tableau 4.3 Comparaison des fonctionnalités offertes par diverses plateformes d'instrumentation 108

Tableau 4.4 Comparaison avec les meilleurs résultats comparables publiés dans la littérature 109

Tableau 4.5 Comparaison par rapport aux travaux antérieurs 110

Tableau 4.6 Comparaison des ressources logiques.....111

LISTE DES FIGURES

Page

Figure 1.1 Architecture d'une carte CPU+FPGA avec mémoires dédiées au FPGA....11

Figure 1.2 Structure de la carte Nallatech 510T, plateforme CPU+FPGA utilisée dans cette thèse....12

Figure 1.3 Flot de conception des applications sur des plateformes hétérogènes reconfigurables....16

Figure 1.4 Catégories des techniques DES....24

Figure 1.5 Modèle de programmation OpenCL avec FPGA....28

Figure 1.6 Modèles de mémoire d'OpenCL....31

Figure 1.7 La structure du SHA-3....41

Figure 2.1 Vue d'ensemble de l'intégration d'un moniteur (instrument) dans un FPGA élaboré en OpenCL....51

Figure 2.2 L'architecture RTL du moniteur....52

Figure 2.3 L'interaction entre le moniteur et le noyau OpenCL....53

Figure 2.4 Flot de conception du *Intel FPGA SDK for OpenCL*....54

Figure 2.5 Intégration des moniteurs dans le noyau OpenCL pipeliné du multiplicateur-accumulateur MAC. (a) MAC sans moniteur (b) MAC avec 4 moniteurs....56

Figure 2.6 Noyau OpenCL du circuit MAC. (a) Noyau sans moniteur. (b) avec moniteur....56

Figure 2.7 Courbe de performance temporelles de l'opération *load* intégré dans le noyau MAC de type single work-item décrit à la figure 2.6(b)....58

Figure 2.8 Courbe de II généré en exécutant le noyau NDRange du MAC décrit à la figure 2.6(b)....58

Figure 2.9 Vue d'ensemble d'un kernel avec instruments....62

Figure 2.10 Étapes d'instrumentation d'un noyau OpenCL....63

Figure 2.11 Métriques de performance d'un seul noyau 64

Figure 2.12 Modélisation d'une structure multi-noyaux 67

Figure 2.13 Architecture de l'instrument proposé 69

Figure 2.14 Le pseudocode des fichiers de description de l'instrumentation (fichiers Verilog, XML et .h) 71

Figure 2.15 Pseudo-code de l'algorithme présenté sous forme d'un seul noyau 73

Figure 2.16 II mesuré par l'instrument I4 dans le modèle d'exécution SWI (WLsize=1, N=50K) 74

Figure 2.17 II des modèles d'exécution NDRange (WLsize=N=50k) (a) sans barrière (les 100 éléments sont présentés), (b) avec barrière (50 k éléments) 75

Figure 2.18 Structure multi-noyaux avec six instruments insérés 78

Figure 2.19 Matrices de II (Γ_{68}) et la matrice de différence Δ_{67}, calculés selon les équations (2.8) and (2.9) respectivement, pour l'implémentation multi-noyaux de l'algorithme MAC 79

Figure 2.20 Taux d'occupation de ALUT sur FPGA pour les 17 applications 82

Figure 2.21 Taux d'occupation de FF sur FPGA pour les 17 applications 82

Figure 2.22 Taux d'occupation de LE sur FPGA pour les 17 applications 82

Figure 2.23 Variation de la fréquence de fonctionnement pour les 17 applications 83

Figure 3.1 Concept de la technique *loop pipelining* 88

Figure 3.2 Concept du déroulement d'une boucle sur des architectures matérielles 89

Figure 3.3 Pseudo-code de la version de base du SHA-256 92

Figure 3.4 Implémentation du SHA-256 tel que décrite par Figure 3.3 93

Figure 3.5 Noyau SHA-256 avec insertion d'une mémoire locale 94

Figure 3.6 Structure du noyau avec fractionnement de la boucle 94

Figure 3.7 Taux d'accélération des implémentations optimisées par rapport à la version de base 97

Figure 3.8 Nombre de ressources utilisées par les différentes implémentations du SHA-256 98

Figure 3.9 Pseudocode de l'implémentation de base de SHA-3 99

Figure 3.10 Structure multi-noyaux du SHA-3 101

Figure 3.11 Débit des différentes versions d'implémentation de SHA-3 102

Figure 3.12 La surcharge logique moyenne sur FPGA comparée par rapport à la version de base du SHA-3 (VB) 103

Figure 4.1 Illustration de la plateforme d'instrumentation proposée dans le flot OpenCL pour les plateformes hétérogènes reconfigurables 106

Figure 4.2 Flot de conception OpenCL pour FPGA 114

LISTE DES ABRÉVIATIONS, SIGLES ET ACRONYMES

ASIC	Application-specific integrated circuit
FPGA	Field-programmable gate array
CPU	Central processing unit
GPU	Graphics processing unit
HDL	Hardware description language
OpenCL	Open computing language
PCIe	Peripheral component interconnect express
HLS	High-level synthesis
VHDL	Very high-speed integrated circuit hardware-description language
RTL	Register transfer level
Avalon-ST	Avalon streaming interface
SRAM	Static random-access memory
NIST	National institute of science and technology
SoC	System-on-chip
HPC	High performance computing
SWI	Single work-item
PE	Processing element
CU	Compute unit
ELA	Embedded logic analyzer
BRAM	Block random access memory
ADM	Array duplicate minimization
MAC	Multiplicateur accumulateur
SHA	Secure hash algorithm
II	Initiation Interval
WI	Work item
LE	Logic element
ALUT	Adaptive look-up table
FF	Flip flop

INTRODUCTION

Contexte général

Plusieurs avancées technologiques récentes sont basées sur la capacité de collecter de grandes quantités de données et de les traiter rapidement. Cette capacité alimente ce qu'on appelle la science des données, domaine pour lequel les traitements s'effectuent souvent grâce à l'apprentissage machine ou à l'extraction de connaissances à partir des données collectées (Dhar, 2013). Les algorithmes d'apprentissage automatique (*machine learning*) (Jordan & Mitchell, 2015) et d'apprentissage profond (*deep learning*) (Goodfellow, Bengio and Courville, 2016) émergent comme la nouvelle force motrice de l'évolution de la science de données et de la nouvelle architecture informatique. Au surplus, plusieurs applications sont soumises à des contraintes de temps réel qui ne peuvent être respectées que par l'utilisation des technologies et d'architectures informatiques basées sur l'apprentissage. Ce besoin de rapidité de traitement a mené vers une croissance spectaculaire de l'industrie des centres de données et des calculateurs. En effet, les entreprises spécialisées en conception de puces continuent d'améliorer les performances des calculateurs en poussant de nouvelles puces sur le marché pour mieux prendre en charge les applications basées sur la science des données (Monroe, 2018). Ces puces peuvent contenir un système complet de traitement de données incluant des supercalculateurs multicœurs et homogènes dont le nombre de cœurs évolue en suivant la loi de Moore (Moore, 1965). Ce progrès technologique au niveau de la conception et de la fabrication des puces et des calculateurs s'explique par l'augmentation du nombre de transistors que l'on peut intégrer sur une puce de silicium, puisque la diminution des dimensions des transistors depuis les années 1960 a été le principal moteur qui a permis d'augmenter de manière exponentielle le nombre de transistors dans les circuits intégrés (J. H. Lau, 2011). Cependant, l'amélioration des performances par la multiplication du nombre de cœurs atteint un plafond technologique causé par certaines limitations telles que le coût de fabrication des puces et leur efficacité énergétique (Esmaeilzadeh, Blem, Amant, Sankaralingam, & Burger, 2012). De nouvelles approches de conception de calculateurs sont requises pour offrir des supercalculateurs qui répondent à ce besoin de rapidité pour le traitement de données (Theis & Wong, 2017).

Motivation : Enjeux des plateformes hétérogènes reconfigurables

De nouvelles puces et plateformes de calcul, qui intègrent des supercalculateurs de haute performance et qui se caractérisent par la diversité croissante de leurs éléments de calcul, sont proposées comme une solution qui permet d'augmenter la capacité de calcul (Rogers & Fellow, 2013). Alors qu'historiquement les architectures des calculateurs étaient principalement composées de processeurs conventionnels (CPU) et de processeurs graphiques (GPU; Graphic Processing Unit), certaines de ces nouvelles plateformes intègrent de plus en plus des unités de calcul reconfigurables de type FPGA (Field Programmable Gate Array). Ces unités de calcul peuvent être intégrées dans la même puce (Nunez-Yanez et al., 2018) ou dans une machine qui intègre plusieurs accélérateurs interconnectés (Reichenbach et al., 2019), (Q. Wu et al., 2014). Ces plateformes sont souvent hétérogènes et peuvent contenir des circuits intégrés personnalisés tels que les ASIC (*application-specific integrated circuit*) (Iyer, 2016). Ces accélérateurs sont généralement jumelés à des mémoires partagées ou privées.

Plus récemment, plusieurs entreprises ont mis sur le marché des plateformes hétérogènes infonuagiques qui sont accessibles à distance. Citons à titre d'exemple la plateforme *F1 compute instance* offerte par Amazon (Amazon, 2017), équipée de plusieurs CPU et FPGA. D'autres entreprises, comme Intel (Oliver et al., 2011), Microsoft (Derek, 2017) et AlphaData (AlphaData), fournissent des plateformes hétérogènes du même type. Cependant, la prise en charge de cette hétérogénéité matérielle exige la présence d'un environnement logiciel adéquat pour supporter cette diversité architecturale afin de maximiser la productivité de développement et d'implémentation des applications sur de telles plateformes (Jack & L, 2009), (Terzo, Djemame, Scionti, & Pezuela, 2019). En outre, un tel environnement logiciel représente un enjeu important puisqu'il doit non seulement supporter les différents flots de conception des accélérateurs, mais aussi les ramener à un niveau d'abstraction unifié qui permet de minimiser le temps de développement et de déploiement des applications sur les différents accélérateurs présents dans la plateforme (H. Andrade, Lwakatare, Crnkovic, & Bosch, 2019).

Le nouvel enjeu est de pouvoir assembler les différents environnements de développement pour les CPU, GPU et FPGA dans un environnement unifié pour répondre au besoin de programmabilité. En effet, les environnements de développement logiciel acceptent comme entrées deux catégories de langage : un langage de bas niveau, tels les langages de description du matériel (*Hardware Description Language*; *HDL*) VHDL et Verilog, ainsi que des langages de haut niveau, citons à titre d'exemples C, C++, SystemC, Java et *Open Computing Language* (OpenCL). Les langages de haut niveau permettent de décrire une application au niveau système, le plus haut niveau d'abstraction, sans se soucier de l'architecture matérielle de l'accélérateur. Le compilateur prend en charge la traduction d'une spécification de haut niveau en code de bas niveau ou en code binaire propre à l'accélérateur ciblé. Néanmoins, l'implémentation d'une application utilisant une spécification de bas niveau doit prendre en considération la propriété matérielle et architecturale des accélérateurs où le développeur doit avoir l'expertise du flot de conception matérielle nécessaire pour optimiser une application donnée avec un tel langage.

Les environnements logiciels des CPU et des GPU fournissent des plateformes de programmation qui acceptent des langages de haut niveau depuis des dizaines d'années. En revanche, les environnements de développement des circuits FPGA présentent souvent une barrière pour les développeurs de logiciels sans expérience antérieure en conception matérielle pour décrire des circuits au niveau RTL (*register-transfer level*). L'adoption d'un langage unifié de description au niveau système vise à améliorer la productivité de développement des applications sur des plateformes hétérogènes reconfigurables sans devoir utiliser un langage HDL pour la description RTL des FPGA (Koch, Ziener, & Hannig, 2016). Pour répondre à ce besoin, plusieurs industriels et chercheurs ont fourni des plateformes qui permettent de configurer les FPGA à partir d'une spécification de haut niveau, tels que le C ou le C++. Cette transformation d'une spécification de haut niveau vers une description RTL est appelée la synthèse de haut niveau (*high level synthesis* (HLS)). Grâce à ces outils, il est possible d'unifier les environnements de développement ciblant des plateformes hétérogènes qui combinent des CPU, GPU et FPGA.

Motivation : Synthèse de haut niveau

Les outils HLS doivent permettre d'abstraire partiellement ou complètement les propriétés matérielles des FPGAs tout en fournissant une plateforme de programmation de haut niveau (Cong et al., 2011). Plusieurs outils HLS ont vu le jour; citons à titre d'exemple l'outil Vivado HLS de Xilinx (Xilinx, 2018b) (O'Loughlin, Coffey, Callaly, Lyons, & Morgan, 2014), l'outil académique et open-source « *LegUp* » (Canis et al., 2011) et l'outil « *Intel FPGA SDK for OpenCL »* de Intel (Altera) (Intel, 2013). Bien que ces outils aient contribué significativement à démocratiser les FPGA en les rendant plus programmables, ils ne permettent pas d'abstraire complètement les propriétés matérielles des FPGA. En effet, les implémentations générées par ces outils offrent souvent de mauvaises performances en comparaison de celles conçues à bas niveau en utilisant les langages HDL (Nane et al., 2016), (Benjamin Carrion Schafer & Wang, 2019). D'ailleurs, la conception en utilisant un outil HLS impose plusieurs modifications et optimisations du style de codage afin de pousser le compilateur à générer des circuits performants (Sakari Lahti, Sjövall, Vanne, & Hämäläinen, 2018). Dans cette thèse, on s'intéresse plus particulièrement à l'évaluation des outils OpenCL pour les plateformes hétérogènes reconfigurables.

Problématique de recherche

OpenCL est un langage de haut niveau qui a été proposé par KhronosGroup (KhronosGroup, 2011). Il permet de décrire des applications qui s'exécutent sur des CPU, GPU, FPGA et d'autres accélérateurs matériels. Il fournit une plateforme de programmation commune qui supporte une grande variété d'accélérateurs hétérogènes ainsi que l'organisation des mémoires qui peuvent être intégrées dans des plateformes hétérogènes. Des industriels offrent des outils HLS qui supportent OpenCL : Intel avec son outil « *Intel FPGA SDK for OpenCL »* et Xilinx avec son outil *SDAccel* (Xilinx, 2016), qui est intégré à sa suite dans l'outil *Vitis Unified Software Platform* (Kathail, 2020). Malgré les bénéfices apportés par OpenCL en termes de programmabilité des FPGA, plusieurs défis et problèmes empêchent OpenCL d'être adopté à grande échelle pour l'accélération des applications sur des FPGA (Sanaullah & Herbordt, 2018). D'ailleurs, plusieurs chercheurs ont évalué l'implémentation de quelques applications de référence (*benchmarks*) et ont montré que des techniques d'optimisation sont nécessaires

pour obtenir de meilleures performances (Krommydas, Helal, Verma, & Feng, 2016), (Krommydas, 2017). En effet, l'augmentation du niveau d'abstraction vient au détriment des performances et c'est également le cas pour les autres outils HLS. De plus, cette dégradation de performance est difficile à analyser en l'absence d'un instrument embarqué ou d'un outil de débogage ciblant les circuits FPGA comme ceux qui existent pour les CPU. Un instrument est un mécanisme qui permet de dévoiler le comportement interne d'un circuit. En outre, les circuits générés par les outils HLS, incluant ceux exprimés en OpenCL sous forme de spécifications HDL sont généralement désorganisés et illisibles, et ce même par les développeurs qui ont une expertise en langage matériel. Les compilateurs OpenCL pour FPGA exploitent des méthodes heuristiques ou métaheuristiques (Hussain, Salleh, Cheng, & Shi, 2019) qui génèrent des descriptions matérielles génériques dont les performances peuvent varier d'un FPGA à l'autre. Par ailleurs, ces algorithmes heuristiques peuvent être limités par la considération de fausses dépendances lors des accès aux mémoires externes, lorsque celles-ci sont utilisées. Ils peuvent aussi violer des contraintes de conception lorsqu'elles ne sont pas explicitement définies dans le code OpenCL. Ces contraintes découlent souvent de primitives spécifiques aux compilateurs HLS qui les guident dans leurs optimisations spécifiques afin de générer les circuits avec les performances désirées. Citons à titre d'exemple la contrainte d'intervalle d'initiation (II) ou la latence d'une boucle. L'intervalle d'initiation est le nombre de cycles mesurés par un instrument pour deux entrées ou sorties successives. La latence est le nombre de cycles nécessaires pour une fonction ou un noyau OpenCL pour finir le traitement d'une seule entrée. Ces deux contraintes sont essentielles à l'identification des parties moins performantes dans un noyau OpenCL.

En d'autres termes, un problème persistant avec les outils OpenCL pour FPGAs ainsi que les autres outils HLS est le manque d'observabilité dans les circuits générés, ce qui empêche les développeurs d'identifier les goulots d'étranglement en matière de performances (Wang, He, Zhang, & Jiang, 2016). Cette problématique a poussé les chercheurs à explorer des solutions pour améliorer l'observabilité dans les circuits générés avec les outils HLS, y compris ceux qui sont générés avec les compilateurs OpenCL. Certains groupes de recherche ont intégré dans le flot de conception des outils personnalisés pour extraire les performances ou pour

observer le comportement du circuit lorsqu'il est implémenté sur FPGA (Sanaullah & Herbordt, 2018). D'autres chercheurs ont proposé des modèles pour prédire les performances d'une application avant d'être implémentés (Zhao et al., 2019). Dans le travail de Jihye Kwon et Luca Carloni (Jihye Kwon & Luca P Carloni, 2020), les chercheurs ont amélioré l'exploration de l'espace de conception (*design space exploration*) en proposant des outils pour identifier la spécification optimale à implémenter.

Objectifs

Dans ce contexte, l'objectif principal de cette thèse consiste à améliorer la productivité de développement d'applications élaborées en OpenCL pour leur mise en œuvre sur des plateformes hétérogènes reconfigurables intégrant CPU et FPGA.
Cet objectif principal peut être divisé en deux objectifs secondaires :

1) améliorer l'observation et la mesure d'événements temporels, tel le décrochage du pipeline, dans les accélérateurs de type FPGA élaborés en OpenCL ;
2) établir les bases pour l'amélioration des performances des accélérateurs de type FPGA pour des applications décrites en OpenCL et vers la conception d'un outil d'optimisation automatisé.

Organisation de la thèse

Les objectifs poursuivis sont explorés à travers les quatre chapitres de cette thèse. Le premier chapitre présente l'état de l'art avec une revue détaillée des méthodes d'instrumentation et d'optimisation des accélérateurs élaborés avec OpenCL qui sont implémentés sur des plateformes reconfigurables. Cette revue inclut les méthodes d'instrumentation des circuits générés par les différents outils HLS ainsi que ceux élaborés en OpenCL. Nous concluons le premier chapitre par une description détaillée des contributions effectuées dans cette thèse. Le chapitre 2 décrit les première et deuxième contributions apportées par cette thèse. Il illustre la méthode proposée pour l'instrumentation des accélérateurs élaborés en OpenCL ciblant des FPGA ainsi que la plateforme d'instrumentation proposée. Le chapitre 3 couvre les troisième et quatrième contributions. Il évalue les méthodes d'optimisation d'applications élaborées en OpenCL qui visent à améliorer les mises en œuvre ciblant des plateformes CPU+FPGA. Le

dernier chapitre compare les performances de la plateforme d'instrumentation et des algorithmes de hachage implémentés à celles de travaux précédemment publiés dans la littérature. Finalement, une conclusion générale clôture cette thèse.

CHAPITRE 1

ÉTAT DE L'ART DES MÉTHODES D'INSTRUMENTATION ET D'OPTIMISATION DES APPLICATIONS ÉLABORÉES EN OPENCL ET MISES EN ŒUVRE SUR DES PLATEFORMES HÉTÈROGÈNES RECONFIGURABLES

Ce chapitre introduit les concepts de base nécessaires à la compréhension de cette thèse et résume les travaux de recherche publiés dans la littérature relative aux contributions proposées. Ce chapitre est composé de cinq sections. La première section présente l'état de l'art et les concepts de base relatifs aux plateformes hétérogènes reconfigurables en relevant les défis et les problématiques qui empêchent leur adoption à une grande échelle. Parmi ces défis, on retrouve une méthodologie de conception commune qui permet de supporter les différents accélérateurs intégrés dans de telles plateformes. Cette méthode de conception unifiée est traitée essentiellement dans la deuxième section. Les différents flots de conception des circuits, plus particulièrement ceux pour les FPGA sont décrits dans cette section. Le langage OpenCL est celui adopté et utilisé dans cette thèse et y est spécifiquement détaillé. On conclut cette section en relevant les directions de recherche traitées par cette thèse. Le premier axe de recherche abordé est l'instrumentation d'applications élaborées en OpenCL mise en œuvre sur des plateformes hétérogènes reconfigurables. Les récentes méthodes d'instrumentation connues sont ainsi présentées dans la troisième section. La quatrième section couvre le deuxième axe de recherche, qui est l'étude d'impact des optimisations appliquées à haut niveau de conception sur les performances d'applications élaborées en OpenCL et implémentées sur FPGA. Deux algorithmes y sont présentés, le *secure hash algorithm*-2 (SHA-2) et SHA-3. Finalement, la dernière section expose les objectifs de recherche ainsi que les contributions originales décrites dans la thèse.

1.1 Architecture hétérogène reconfigurable

1.1.1 Définition

L'architecture reconfigurable n'a pas de définition précise et cette dernière peut varier avec le contexte d'employabilité. Selon Mansureh et al (2017), l'architecture reconfigurable est une architecture informatique combinant une partie logicielle flexible et une partie matérielle qui possède une logique configurable (*configurable fabric*) pour exécuter une tâche spécifique. Nous définissons dans cette thèse l'architecture reconfigurable comme suit : une architecture dont au moins l'un de ses éléments matériels peut être reconfiguré à l'aide d'un outil logiciel. La reconfigurabilité peut se référer à un système ou à n'importe quel élément logique du système. Ces systèmes peuvent être un simple circuit ou une plateforme qui intègrent un grand nombre de puces. En particulier, une plateforme est dite hétérogène reconfigurable lorsqu'elle contient plusieurs puces incluant des processeurs et des accélérateurs matériels tels des FPGA, et qui sont partiellement ou totalement reconfigurables.

1.1.2 Historique et motivation

Au cours des dernières décennies, le calcul à haute performance (*high performance computing* ; HPC) s'est essentiellement imposé comme un facteur principal du progrès scientifique et qui a lui-même bénéficié de l'avènement de la technologie des semiconducteurs. La croissance du HPC a été considérablement favorisée par l'intégration de processeurs multicœurs (Gepner & Kowalik, 2006) ainsi que par l'amélioration des performances des réseaux de communication (Biberman & Bergman, 2012). En revanche, l'amélioration des performances de calcul est fortement limitée entre autres par le nombre de cœurs qui peuvent être intégrés dans une puce, ainsi que par la capacité de ces processeurs à exécuter des algorithmes venant de différents domaines d'applications. De ce fait, les plateformes plus spécialisées offrent une solution permettant de répondre aux besoins de calcul et de surmonter les limites de la densité d'intégration, de la puissance et de la parallélisation. Les architectures spécialisées sont des architectures matérielles contenant des cœurs de calcul conçus spécialement pour un domaine bien déterminé (Cong, Sarkar, Reinman, & Bui, 2010).

Cependant, ces architectures spécialisées limitent considérablement le champ d'applications et elles manquent de flexibilité au niveau de la configuration.

De ce fait, les plateformes hétérogènes reconfigurables se sont révélées comme une des solutions potentielles et prometteuses permettant non seulement d'améliorer les performances de calcul, mais aussi de répondre à la diversité d'applications qui peuvent être déployées. En particulier, plusieurs fournisseurs commerciaux offrent des plateformes incluant des CPU, des FPGA, et des GPU telles que les plateformes d'Intel (Oliver et al., 2011), de Huawei (Huawei, 2019) et d'Amazon (Abel, Weerasinghe, Hagleitner, Weiss, & Paredes, 2017). Ces plateformes ont été proposées pour répondre à la demande croissante de puissance de calcul (Liu, Zydek, Selvaraj, & Gewali, 2012).

1.1.3 Plateformes hétérogènes reconfigurables intégrant des FPGA

L'intégration des FPGA dans les plateformes HPC et les centres de données est considérée comme l'une des approches les plus prometteuses pour soutenir la croissance en termes de puissance de calcul. Cette section décrit principalement les plateformes hétérogènes reconfigurables intégrant des FPGA en détaillant leurs spécifications matérielles ainsi que leurs approches d'intégration physique. Cette section met en évidence les limitations qui accompagnent l'utilisation de ces plateformes. Les défis à relever sont finalement présentés.

Les plateformes hétérogènes reconfigurables intégrant des FPGA sont classifiées selon leurs approches d'intégration physique et les modèles de mémoires qui y sont associés (Y.-k. Choi et al., 2016) (Y.-K. Choi et al., 2019). En effet, l'architecture typique d'une plateforme hétérogène reconfigurables de type CPU+FPGA est basée sur un système de communication de type PCI express, d'une série de mémoires dédiées attachées aux FPGA et aux CPU, telle qu'illustrée à la Figure 1.1. Les mémoires peuvent être partagées (Oliver et al., 2011) ou dédiées au FPGA, telles que conçues sur la carte Nallatech 510T (Nallatech, 2021). La Figure 1.1 présente une architecture typique avec mémoires privées aux FPGA.

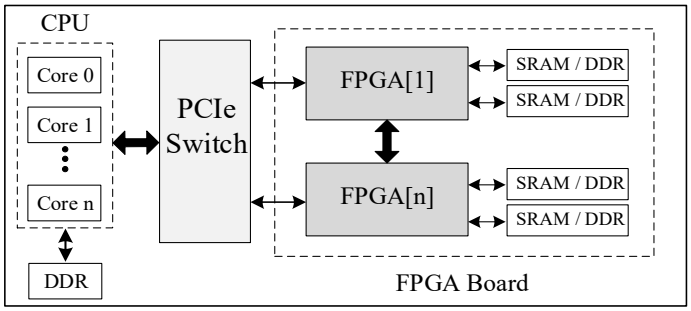


Figure 1.1 Architecture d'une carte CPU+FPGA
avec mémoires dédiées au FPGA

La plateforme de type CPU+FPGA est caractérisée par sa flexibilité et son extensibilité où plusieurs cartes FPGA sont connectées à plusieurs CPU via l'interface PCIe (Abel et al., 2017). Plusieurs entreprises comme Intel (Oliver et al., 2011), Amazon (Amazon, 2017), Microsoft (Derek, 2017) et Alpha data (AlphaData) fournissent des plateformes de même type dans lesquelles une interface de communication PCIe est utilisée pour connecter plusieurs CPU et FPGA. Les fournisseurs des FPGA fournissent des cœurs logiques (*intellectual property* (IP)) pour l'interfaçage des mémoires et pour le contrôle de la communication PCIe afin de faciliter le développement d'applications sur de telles plateformes. Un exemple de plateforme hétérogène reconfigurable intégrant CPU et FPGA, utilisée dans ces travaux de thèse, est constituée de la carte Nallatech 510T installée sur une machine Dell à l'aide d'une interface PCIe (3.0), telle qu'illustrée à la **Erreur ! Source du renvoi introuvable.**. La carte Nallatech 510T contient deux FPGA Arria 10 1150 GX et quatre bancs de mémoire privée de 4 Go par FPGA (32 Go au total).

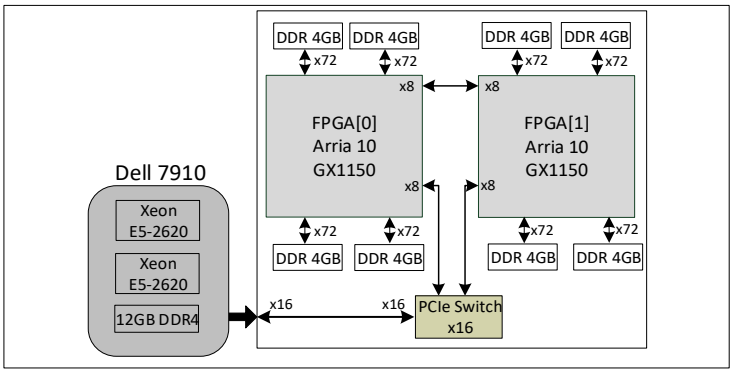


Figure 1.2 Structure de la carte Nallatech 510T,
plateforme CPU+FPGA utilisée dans cette thèse

1.1.4 Problématique et défis des plateformes hétérogènes reconfigurables

Bien que les plateformes hétérogènes reconfigurables améliorent la puissance de calcul et sont de plus en plus disponibles pour l'usage commercial ou académique, elles sont encore difficiles à utiliser efficacement (Zahran, 2017). En particulier, la diversité en termes de langage de programmation, de flots de conception, d'instruments et d'outils utilisés limite leur utilisation à grande échelle et limite la productivité de développement (Bell, Pu, Hegarty, & Horowitz, 2018). D'ailleurs, la principale limitation est l'absence d'une infrastructure logicielle multiparadigme qui unifie les flots de conception ainsi que les outils utilisés ; permettant alors l'amélioration de la productivité de développement des applications sur des plateformes hétérogènes reconfigurables (H. Andrade et al., 2019) (J. Lau et al., 2020). Par exemple, la configuration d'accélérateurs comme les FPGA nécessite des connaissances approfondies du flot de conception matériel ainsi du débogage et l'instrumentation des performances des circuits implantés. Ces connaissances requises s'ajoutent aux autres connaissances de développement logiciel visant les FPGA et les CPU. Une telle infrastructure logicielle doit aussi offrir une stratégie robuste, notamment pour le déploiement et l'exploitation d'outils, dans lesquels les applications doivent être correctement décomposées, ordonnancées, allouées, exécutées et enfin instrumentées pour des fins d'optimisations (Hugo Andrade, Schroeder, & Crnkovic, 2019). Ces outils d'instrumentation et de diagnostic sont utilisés par les développeurs dans le cas où une dégradation de performances est détectée lors de l'exécution d'une application donnée.

1.2 Méthodes de conception sur des plateformes hétérogènes reconfigurables

La cohabitation de différents accélérateurs et unités de traitement dans une même plateforme hétérogène reconfigurable implique un défi associé à la distribution et l'ordonnancement des tâches d'une application sur les accélérateurs. De plus, cette distribution amène à adopter les

différents flots de conception associés au FPGA, CPU et GPU. Ces flots sont résumés à la

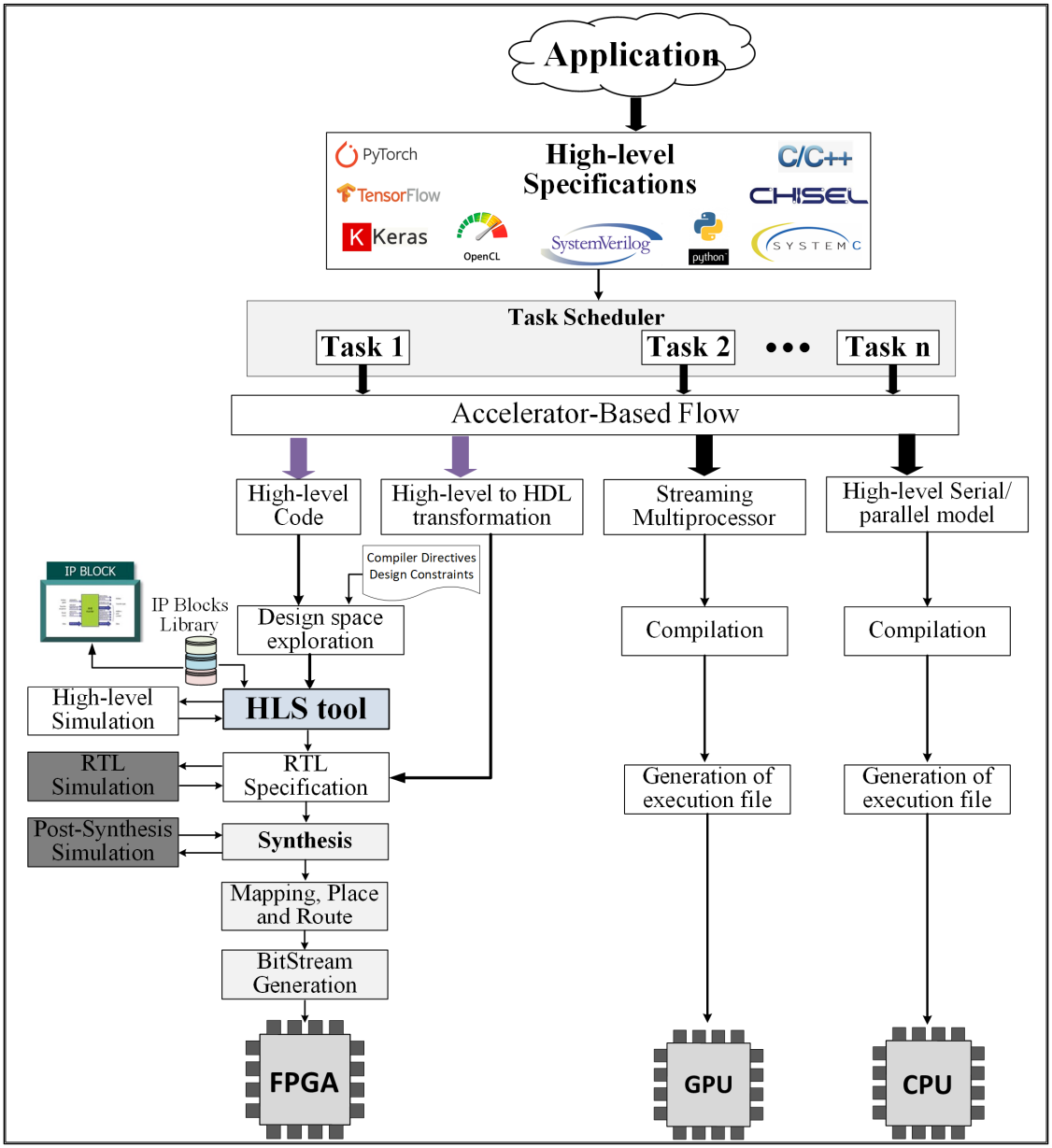


Figure 1.3. Ces flots prennent comme entrée une spécification de haut niveau de l'application décrite avec un langage idéalement unifié, tel que C/C++, Python, SystemVerilog et OpenCL. Une application passe premièrement par un outil d'ordonnancement qui divise l'application en

plusieurs tâches envoyées chacune à l'accélérateur cible (

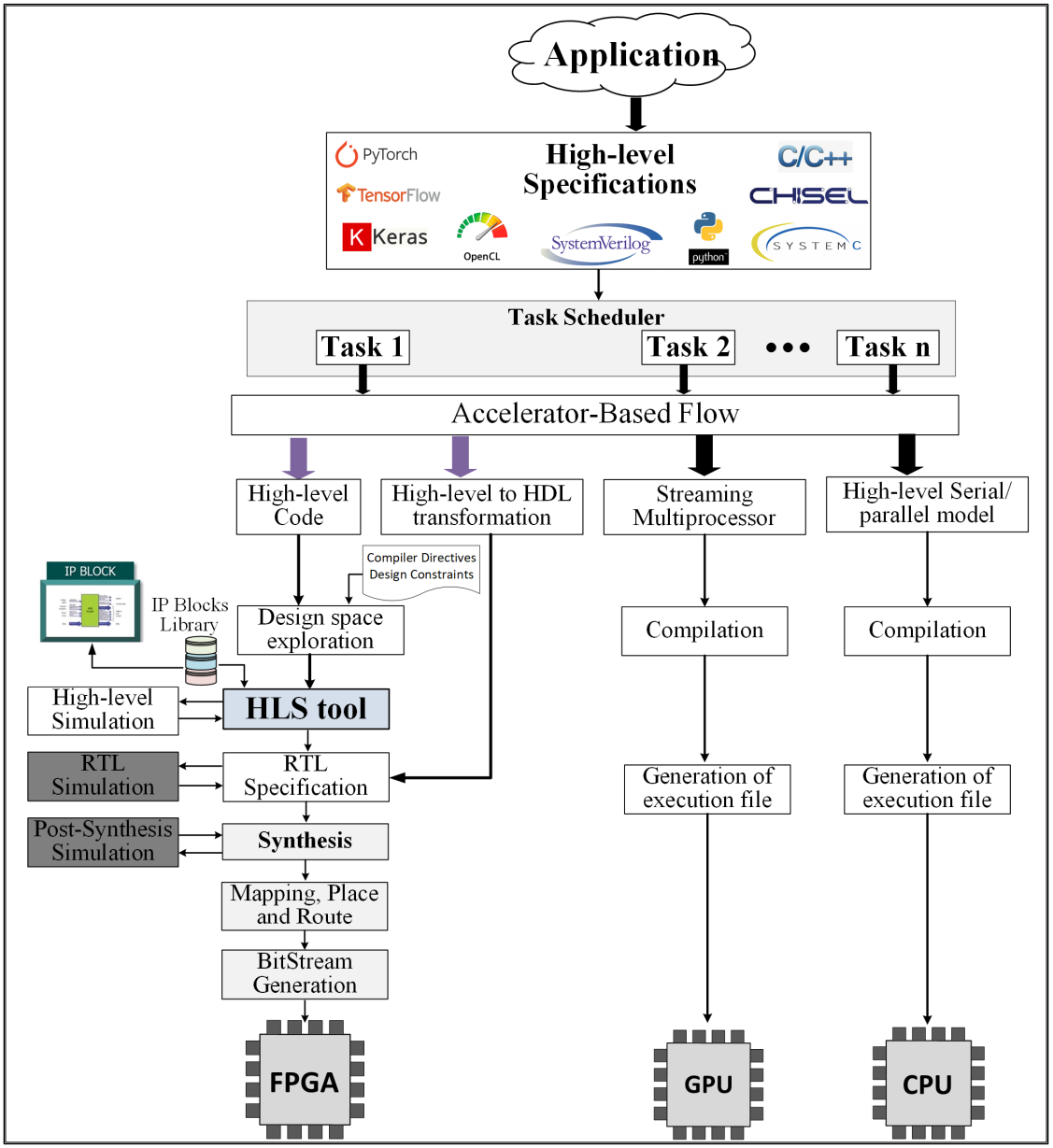


Figure 1.3). Les tâches ciblant les CPU ou GPU sont compilées en utilisant les compilateurs logiciels tels GCC, Microsoft Visual Studio et CUDA (appelé NVCC) (Luebke, 2008), tandis que celles qui devront s'exécuter sur FPGA passent par le flot de conception basé sur les outils HLS. L'un des obstacles qui empêchent une utilisation des outils HLS à grande échelle, tel le cas pour les outils de CPU ou GPU, est le manque d'efficacité d'outils permettant l'instrumentation et l'analyse de performances. En effet, la simulation RTL avant ou après synthèse ne permet pas une compréhension de fonctionnement du code à haut niveau, car les

outils HLS à base d'OpenCL n'offrent aucun lien d'identification entre les deux niveaux d'abstraction. En effet, les modules et les signaux à bas niveau n'ont aucune signification liée aux variables et aux fonctions à haut niveau. Nous expliquerons en détail ces défis et limitations dans la section 1.2.1.4. En revanche, les développeurs utilisent efficacement des outils d'analyse de performances tels Intel VTune (Intel, 2020b) pour CPU et NSight (NVIDIA, 2019) pour GPU (Y.-k. Choi, 2019). Par ailleurs, il n'existe pas encore une structure de développement logiciel permettant l'analyse de performances lors du déploiement d'applications sur des plateformes hétérogènes reconfigurables.

Plusieurs travaux de recherche proposent des environnements dédiés à des domaines d'application spécifiques avec leur propre langage de description (*domain-specific languages* (DSL)) pour améliorer la productivité de développement et les performances. Entre autres, les auteurs de l'article « *HeteroCL*: *A Multi-Paradigm Programming Infrastructure for Software-Defined Reconfigurable Computing* » (Lai et al., 2019) proposent le modèle de programmation HeteroCL dédiée à l'apprentissage profond. Il est conçu en utilisant une plateforme appelée TVM (T. Chen et al., 2018), qui est un compilateur d'apprentissage automatique, à code ouvert, pour les CPU et les GPU. Leur approche vise à éviter les outils commerciaux ; cela implique l'évitement des limites qui y sont associées afin d'obtenir une plateforme dédiée spécialement à l'apprentissage profond. Des plateformes similaires ont été proposées pour les applications de traitement d'images telles Halide (Ragan-Kelley et al., 2013) ou en traitement de données comme Spark (Zaharia, Chowdhury, Franklin, Shenker, & Stoica, 2010).

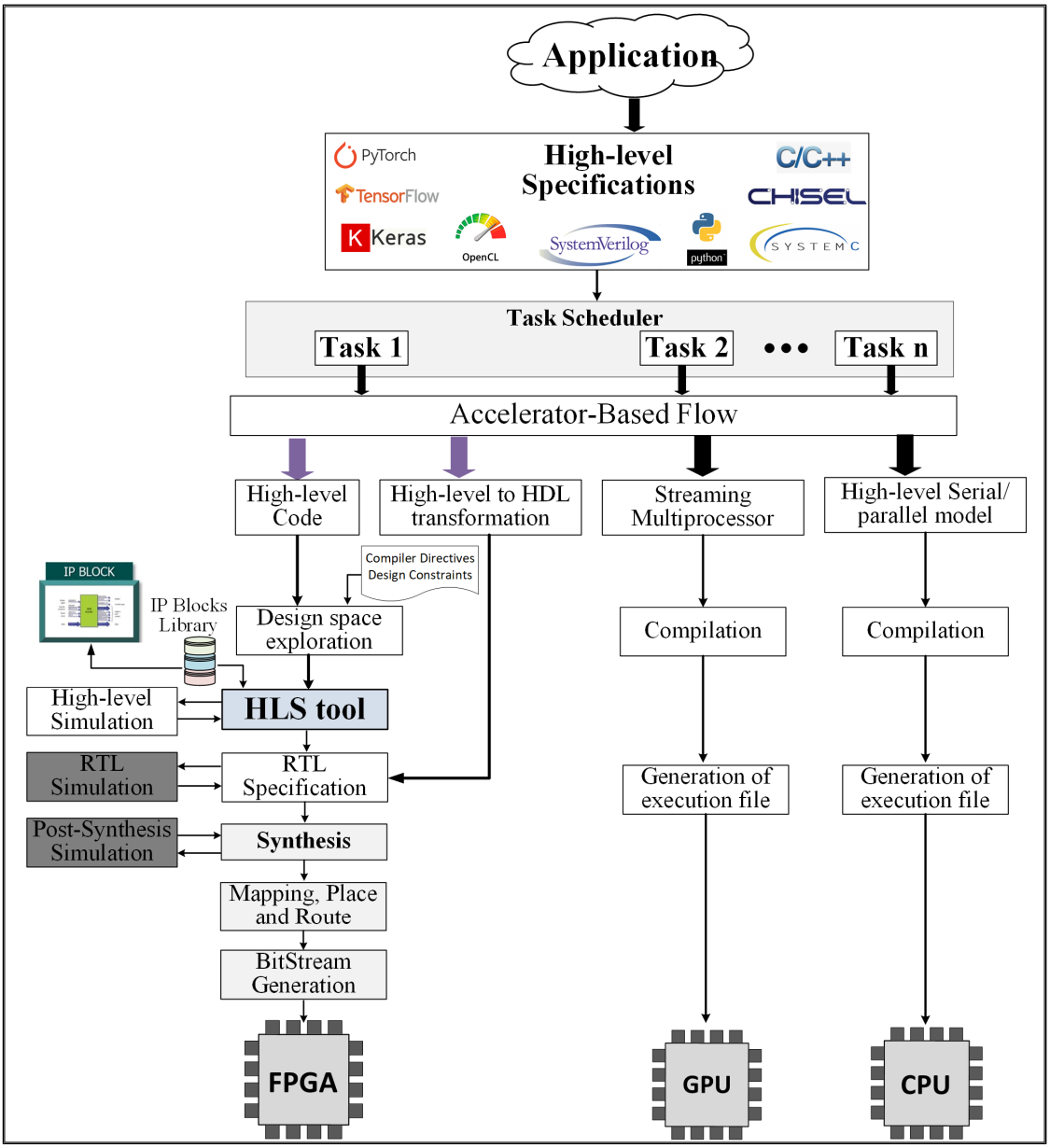


Figure 1.3 Flot de conception des applications sur des plateformes hétérogènes reconfigurables

Plutôt que de viser un domaine bien déterminé, les travaux présentés dans cette thèse visent à utiliser un langage universel tel OpenCL qui est associé à un outil commercial prêt à l'utilisation. L'objectif principal est donc d'améliorer le développement des applications élaborées en OpenCL pour leur mise en œuvre sur des plateformes hétérogènes

reconfigurables. Cette thèse vise également à bonifier les flots présentés par la

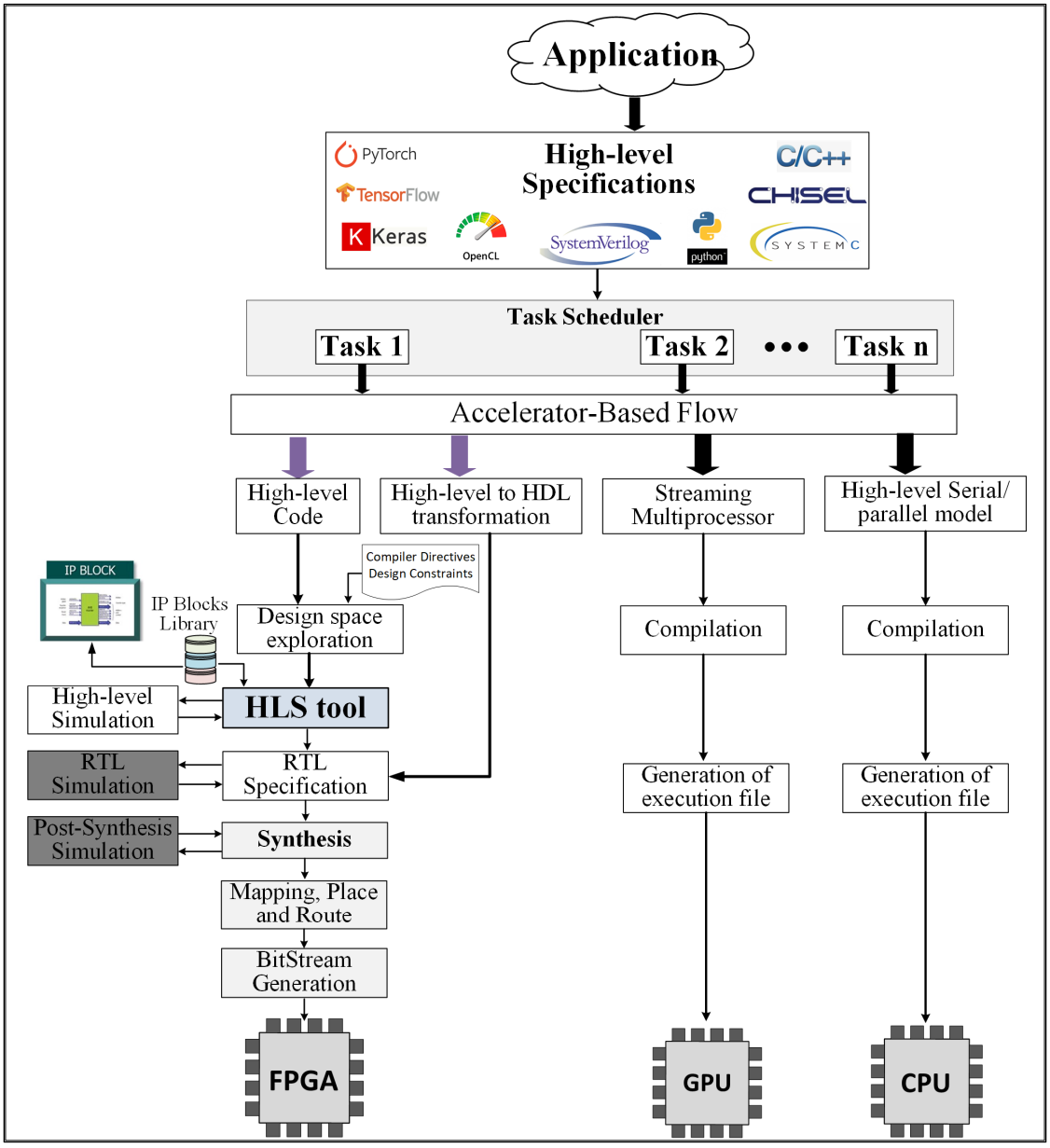


Figure 1.3 afin d'améliorer la productivité de développement, ce qui pourrait éventuellement contenir des outils de support au déploiement des applications sur des FPGA et CPU. La section 1.2.1 explique le concept de synthèse de haut niveau en relevant les limitations et les défis qui y sont associés. La section 1.2.2 introduit les concepts de base nécessaires à la compréhension du langage de programmation OpenCL.

1.2.1 Synthèse de haut niveau

La synthèse de haut niveau (*high-level synthesis* (HLS)) est depuis plusieurs décennies un axe de recherche très actif pour les chercheurs spécialisés en outils de conception des circuits intégrés. Cette section retrace l'historique des outils HLS et leur évolution. Elle est suivie d'une description plus détaillée du flot de conception basé sur la HLS. Cette section se termine par une revue critique des outils HLS, des défis qui y sont associés et de leurs limitations.

1.2.1.1 Historique et évolution de la HLS

La synthèse de haut niveau a commencé vers les années 1980 par les travaux sur les méthodes heuristiques visant la conception de compilateurs (Trickey, 1987) et d'outils de synthèse destinés aux circuits numériques et aux circuits intégrés à très grande échelle « VLSI » (T. J. Kowalski & Thomas, 1983) (T. J. Kowalski & Thomas, 1985) (T. Kowalski, Geiger, Wolf, & Fichtner, 1985). La naissance du premier circuit FPGA à base de mémoire SRAM (*static random-access memory*), mise en maché par Xilinx en 1985 (Hsieh, Mahoney, Ngo, & Shelly, 1986), ainsi que l'apparition des langages de description de matériel (HDL) au niveau RTL (*Register Transfert Logic*) tels Verilog (Flake, Moorby, Golson, Salz, & Davidmann, 2020) et VHDL (Lipsett, Marschner, & Shahdad, 1986) ont mené à l'élaboration d'une suite d'outils automatique pour la synthèse, le placement et le routage des circuits numériques (Lis & Gajski, 1988). Les HDL sont encore largement utilisés comme entrées aux outils de simulation et de synthèse logique, conduisant alors à la définition de leurs circuits synthétisables. La synthèse à partir d'une description de matériel à un plus haut niveau d'abstraction que le RTL vise à remédier au problème de complexité des circuits et des systèmes.

L'approche HLS utilise comme entrée une description de l'application au niveau système. Cette entrée permet une description sous la forme logicielle classique (exemple, C++) en omettant les détails d'implémentation matérielle tels que les contraintes de temps (*design constraints*) et la nature des interfaces utilisées par une telle application. Dans cette thèse la synthèse de haut niveau est définie comme étant la génération d'une spécification synthétisable décrite avec un langage matériel (VHDL ou Verilog) à partir d'une spécification décrite au

niveau système avec un langage de haut niveau (p. ex. C/C++, OpenCL) (Coussy, Gajski, Meredith, & Takach, 2009).

Les travaux de recherche sur la synthèse de haut de niveau se sont intensifiées par l'imbrication de plusieurs sous-domaines, notamment la conception conjointe matérielle/logicielle, l'estimation de performances, l'exploration de l'espace de conception, le partitionnement sur plusieurs éléments de calcul, l'interfaçage entre les éléments de calcul, la covérification et cosimulation matérielle/logicielle. D'ailleurs, l'intérêt des chercheurs aux domaines qui sont fortement liés à la synthèse de haut niveau a pris de l'ampleur, de manière que le nombre d'articles scientifiques qui traitent les problèmes liés à la synthèse de haut niveau ne cessent d'augmenter significativement. Ceci montre l'envergure de la recherche sur la synthèse de haut niveau en tant qu'approche de conception pour les circuits FPGA et les plateformes hétérogènes configurables. Parmi les résultats de ces recherches, on trouve la naissance de plusieurs outils HLS fournis par le milieu académique tel l'outil LegUp (Canis et al., 2013) ou les outils commerciaux Vivado HLS (Xilinx, 2021) ou Intel HLS compiler (Intel, 2021b). Par ailleurs, Intel offre l'outil *Intel FPGA SDK for OpenCL* (Intel, 2013) qui permet l'implémentation des applications élaborées en OpenCL sur des FPGA. Cette thèse se limitera à l'exploration d'OpenCL, un des langages les plus répandus dans ce domaine, en vue de son adoption comme un langage de programmation des plateformes hétérogènes reconfigurables de type FPGA+CPU.

1.2.1.2 Flot de conception basé sur la HLS

La conception des circuits numériques en utilisant la méthode HLS simplifie le processus de développement en utilisant un flot similaire au développement logiciel puisque les outils HLS prennent en charge tous les détails relatifs à l'implémentation du matériel telle l'insertion des

cœurs de propriété intellectuelle (*IP cores*) et des interfaces de communication. La

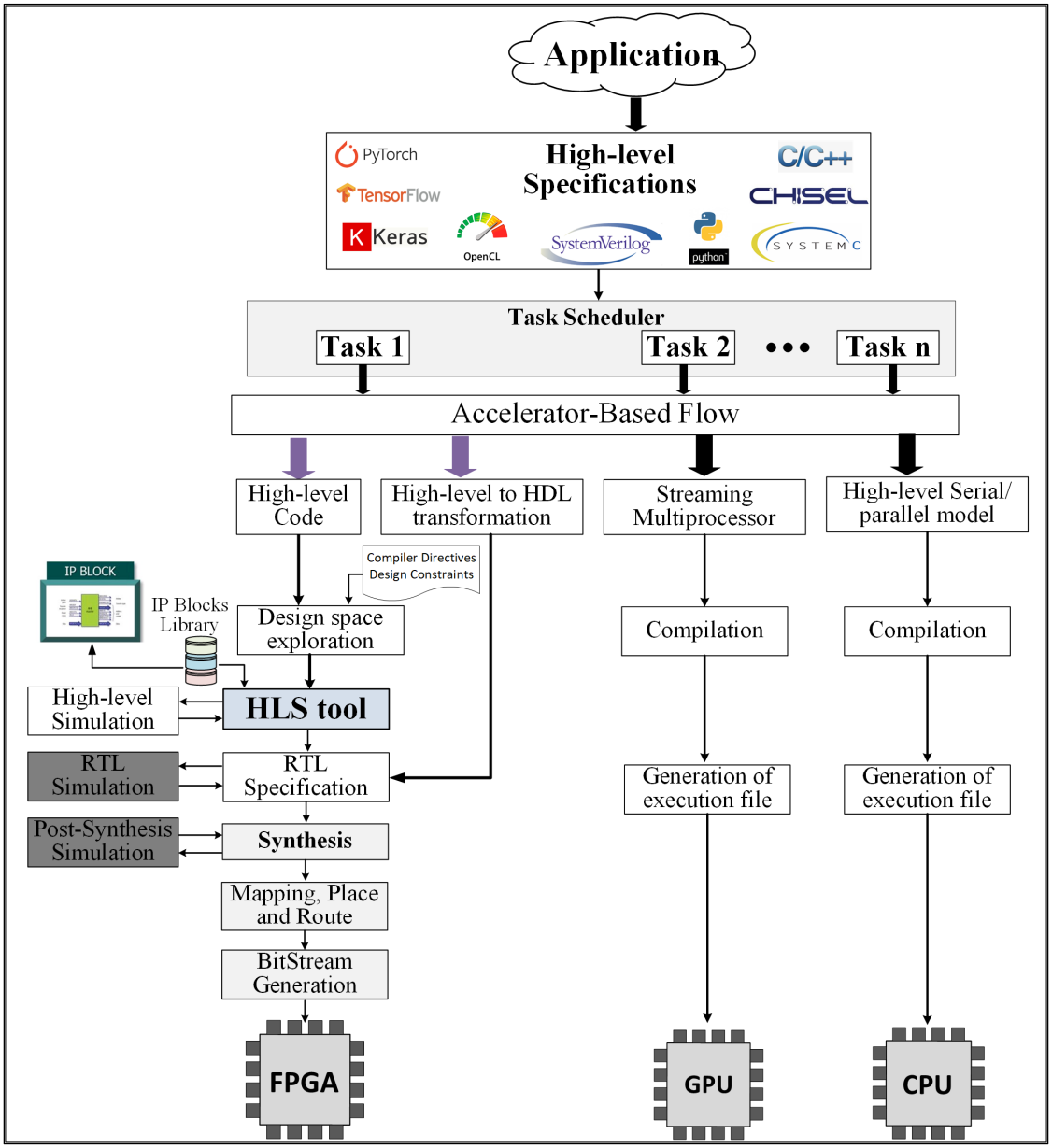


Figure 1.3 présente en partie le flot de conception des FPGA basé sur la synthèse de haut niveau. Ce flot inclut les différentes étapes de conception partant d'un code de haut niveau jusqu'à la reconfiguration du FPGA. De plus, la génération du code de haut niveau qui sert comme entrée aux outils HLS doit passer par plusieurs étapes avant d'avoir une spécification adéquate : l'exploration de l'espace de conception (*design space exploration* (*DSE*)), l'adaptation du code aux compilateurs utilisés (*compiler directives*) et finalement

l’optimisation du code. L’exploration de l’espace de conception consiste à analyser les différentes spécifications possibles pour implémenter une application.

Une fois que la spécification du haut niveau est prête à l’implémentation, l’outil HLS génère une implémentation qui sera analysée par le développeur pour s’assurer des performances fournies. En effet, les développeurs commencent avec une spécification initiale et fonctionnelle de leur application. Ensuite, ils effectuent un certain nombre d’itérations pour générer une implémentation satisfaisante qui répond aux performances désirées. Ces itérations doivent

parcourir tout le flot de conception relatif au circuit FPGA dans la

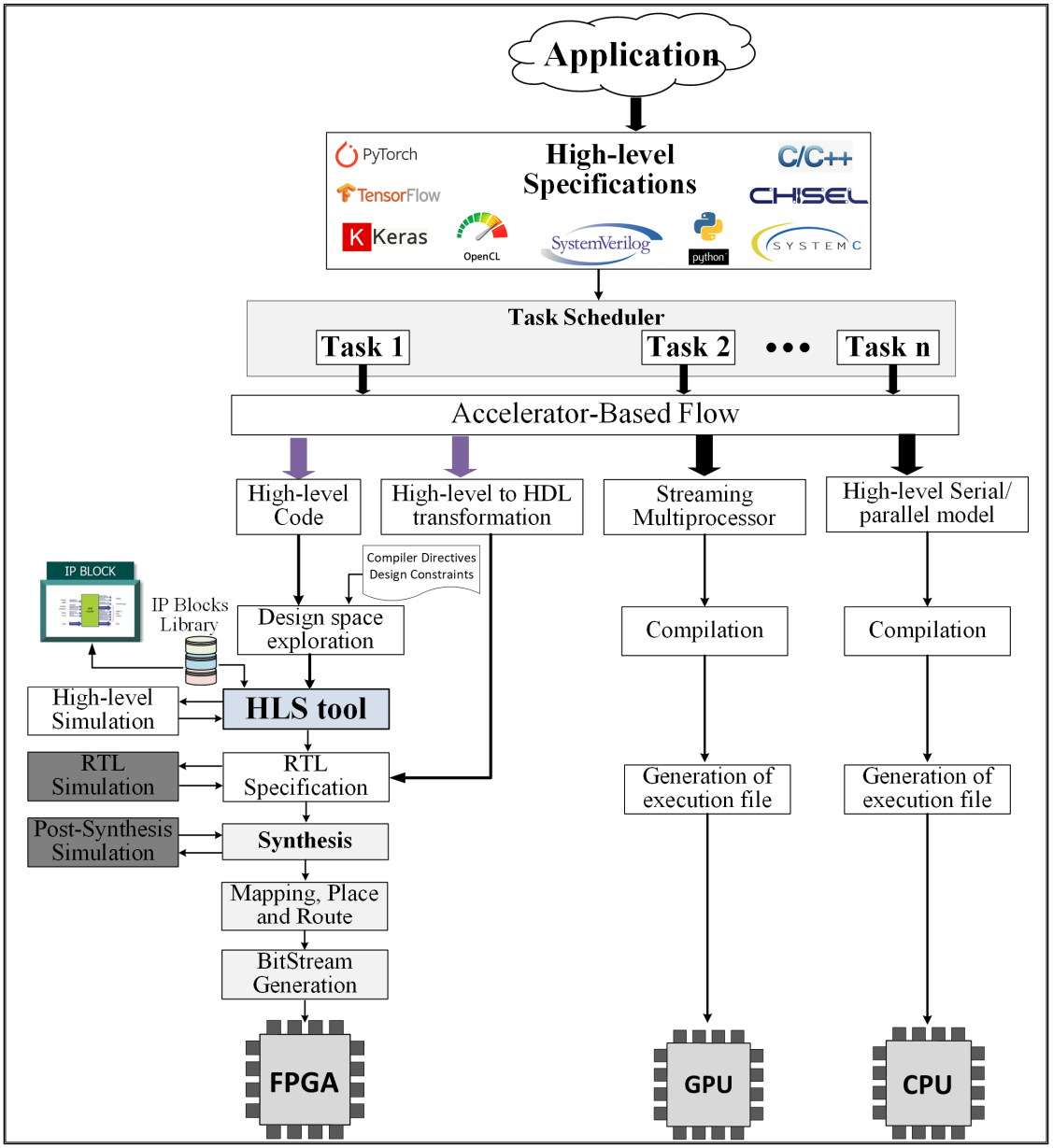


Figure 1.3, ce qui rend ces itérations à la fois très complexes et de longue durée vu la diversité des options d'implémentation pour une même fonctionnalité. Ces options d'implémentation sont en partie déterminées lors de l'étape de DSE. La section 1.2.1.3 décrit les techniques DSE.

1.2.1.3 Exploration de l'espace de conception

Le principal problème auquel les développeurs sont confrontés lors de l'utilisation d'un outil HLS est de savoir quelles sont les options à adopter à haut niveau d'abstraction qui permettent d'obtenir une meilleure implémentation en termes de performances, sachant qu'un nombre élevé de microarchitectures différentes peut être obtenu pour une seule spécification de haut niveau, tout en changeant les options de synthèse. Une microarchitecture regroupe les différents blocs logiques (exemple : *Flip Flops*) qui peuvent se trouver dans une implémentation sur FPGA.

Ainsi, l'exploration de l'espace de conception (DSE) permet de sélectionner les spécifications qui répondent le mieux aux performances désirées. L'objectif du processus de DSE n'est pas de trouver une solution unique, mais plutôt de limiter le nombre de spécifications cibles en définissant différentes options de synthèse et d'optimisation. Ces options de synthèse ou d'optimisation sont appelées dans la littérature « *knobs* » et elles peuvent être liées aux compilateurs, appelées souvent *compiler specific optimization*s, ou des transformations du code à haut niveau tel le déroulement des boucles. Dans ce contexte, plusieurs techniques d'exploration de l'espace de conception ont été proposées dans la littérature visant la synthèse de haut niveau. Ces techniques sont divisées en quatre catégorisés selon Schafer et Wang (2020). La Figure 1.4 montre ces catégories qui sont regroupées en deux classes : des techniques DSE basées sur la synthèse et d'autres sur les modèles.

Les techniques basées sur la synthèse génèrent un ensemble de *knobs* et appellent l'outil HLS afin d'évaluer l'impact de ces nouveaux *knobs* sur les performances. Cette classe regroupe les techniques heuristiques (Gawiejnowicz, 2020) et métaheuristiques (Yang, 2011) tels les algorithmes génétiques. La troisième catégorie, l'apprentissage supervisé, est un mélange entre la synthèse et la modélisation. En effet, cette catégorie de techniques utilise l'apprentissage supervisé pour générer des modèles prédictifs permettant de prévoir la surface logique et les performances sans faire la synthèse en appelant les outils HLS. Cependant, ces techniques échantillonnent d'abord l'espace de recherche en synthétisant continuellement un certain

nombre d'implémentations afin d'avoir un modèle prédictif stable. Ces différentes implémentations sont choisies en fonction d'un nouveau réglage d'options d'optimisation. Dans le travail de Jihye Kwon et Luca Carloni (2020), un modèle de réseau de neurones est proposé pour l'exploration de l'espace de conception. Ce modèle utilise l'apprentissage fait lors des implémentations antérieures faites par les outils HLS.

La quatrième catégorie (*graph analysis based*) utilise des modèles prédictifs hors ligne sous forme des graphes de flux de données (*control data flow graph* (CDFG)) pour explorer les options d'implémentations adéquates. Lors de l'utilisation de ces techniques, l'outil HLS ne sera jamais utilisé pour l'estimation de performance. En revanche, ces techniques analysent la description comportementale à explorer et construisent un CDFG pour l'extraction des options d'optimisation adéquates tel le degré de parallélisme des boucles. Ensuite, les modèles prédictifs permettent d'estimer les performances des implémentations en utilisant les informations extraites à partir des graphes. Un modèle de performance appelé FlexCL, qui utilise un CDFG, est proposé par Wang et Zhang (2018) pour analyser les performances ainsi que la puissance dissipée par un mise en œuvre des applications OpenCL sur FPGA.

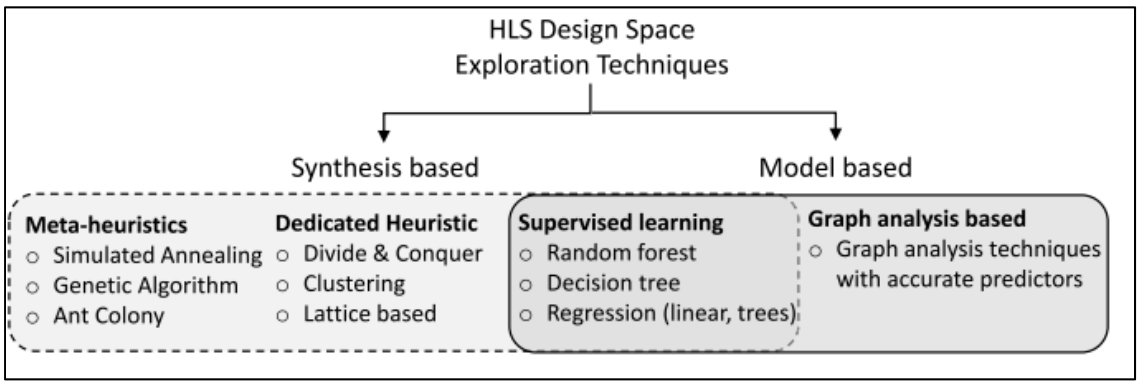


Figure 1.4 Catégories des techniques DES
Tirée de B.C. Schafer et Wang (2020)

1.2.1.4 Défis et limitations des outils HLS

Une excellente étude des outils HLS est présentée par Windh et al (2015), les auteurs ont présenté une évaluation de cinq outils HLS : Xilinx *Vivado HLS*, *Intel* (Altera) *FPGA SDK for OpenCL* (Intel, 2013), *Bluespec System Verilog* (BSV) (Arvind, 2009), *ROCCC* (Najjar, Villarreal, & Halstead, 2016) et LegUp (Canis et al., 2013) en fournissant une vue d'ensemble de leurs flots de conception, les optimisations qu'ils offrent et une analyse qualitative de leurs implémentations matérielles. L'évaluation de l'outil de Intel « *Intel FPGA SDK for OpenCL »* montre que les implémentations générées par cet outil sous forme de fichiers Verilog sont non lisibles par les développeurs. De plus, il n'existe pas de méthodes qui permettent de faire un lien entre les simulations au niveau RTL et la simulation logicielle à haut niveau. Par exemple, un signal ou un module à bas de niveau (RTL) n'a aucune correspondance à haut niveau. Une fonction à haut niveau peut être implémentée par différents modules au niveau RTL non connus et non identifiés pour l'utilisateur. De plus, l'utilisation des outils d'instrumentation ou de débogage tel ChipScope (Xilinx, 2012b) n'offre aucune compréhension à la fonctionnalité d'application lorsque cette dernière est décrite avec un langage de haut niveau. En effet, l'observation des signaux au niveau RTL est inutile puisque le développeur n'a aucune idée des modules RTL générés par les outils de synthèse de haut niveau. Cette obscurité à bas niveau complexifie l'analyse des implémentations générées. Ces limitations empêchent les développeurs d'investiguer les goulots d'étranglement dans les applications déployées.

Dans l'article "*A Survey and Evaluation of FPGA High-Level Synthesis Tools*" (Nane et al., 2016), les développeurs de trois outils HLS académiques: « Delft workbench automated reconfigurable VHDL generator » (DWARV) (Nane et al., 2012), « BAMBU » (Pilato & Ferrandi, 2013) et « LegUp » (Canis et al., 2013) ont présenté une méthodologie permettant une évaluation approfondie de performance de ces différents outils en utilisant un ensemble de bancs d'essai (benchmarks). Une comparaison avec un outil commercial est aussi présentée. Cette évaluation est basée sur trois métriques de performance : la fréquence maximale (Fmax en MHz), la latence (le nombre de cycles d'horloge nécessaires pour une exécution de l'application) et le temps de *wall-clock* (période d'horloge minimale × latence). Les résultats

de cette étude montrent que chaque outil HLS évalué peut améliorer considérablement les performances grâce à des optimisations spécifiques à chaque application ou domaine d'applications. Cependant, les développeurs logiciels doivent tenir compte que ces optimisations, ciblant les architectures matérielles reconfigurables, sont considérablement différentes à celles dédiées au logiciel (par exemple, la réorganisation des données du cache). De plus, les résultats ont montré que les outils de HLS académiques et commerciaux ne sont pas radicalement différents en termes de qualité de résultats. Cependant, les outils commerciaux offrent plus de fonctionnalités que les outils académiques.

Une revue approfondie de la littérature qui a couvert 46 articles et 118 applications associées est faite par Lahti et al. (2019) pour investiguer la différence de la qualité des résultats (quality of results; QoR) et de la productivité entre les deux flots de conception HLS et RTL. Les résultats montrent qu'en moyenne, le QoR du flot RTL est meilleur que celui des outils de HLS. Cependant, le temps de développement moyen avec les outils RTL est trois fois plus grand que celui du HLS, et un développeur obtient une productivité entre quatre et six fois plus élevée avec l'approche de conception HLS. En outre, pour aider à combler l'écart de qualité de service, cette thèse explore dans son troisième chapitre les optimisations des applications basées sur l'approche HLS et plus particulièrement le flot de conception OpenCL pour FPGA.

Tous les auteurs des travaux examinés et présentés dans cette sous-section affirment que les outils HLS ont fait des progrès significatifs permettant la configuration des FPGA avec comme entrée une spécification de l'application décrite avec des langages de programmation de haut niveau. Cependant, des modifications manuelles visant l'adaptation vers les architectures matérielles utilisées sont nécessaires pour l'optimisation ou la compilation sans erreurs. De plus, les performances sont souvent décevantes et inférieures à celles des CPU à cause des mauvaises optimisations adoptées. Bien que les outils HLS ont amené le développement avec les FPGA à un niveau d'abstraction comparable à celui des CPU pour un concepteur logiciel, leurs limitations en termes d'optimisation automatique, d'instrumentation et de débogage diminuent la productivité et présentent toujours un défi majeur face à l'unification des

méthodes de conception présentes dans les plateformes hétérogènes reconfigurables (

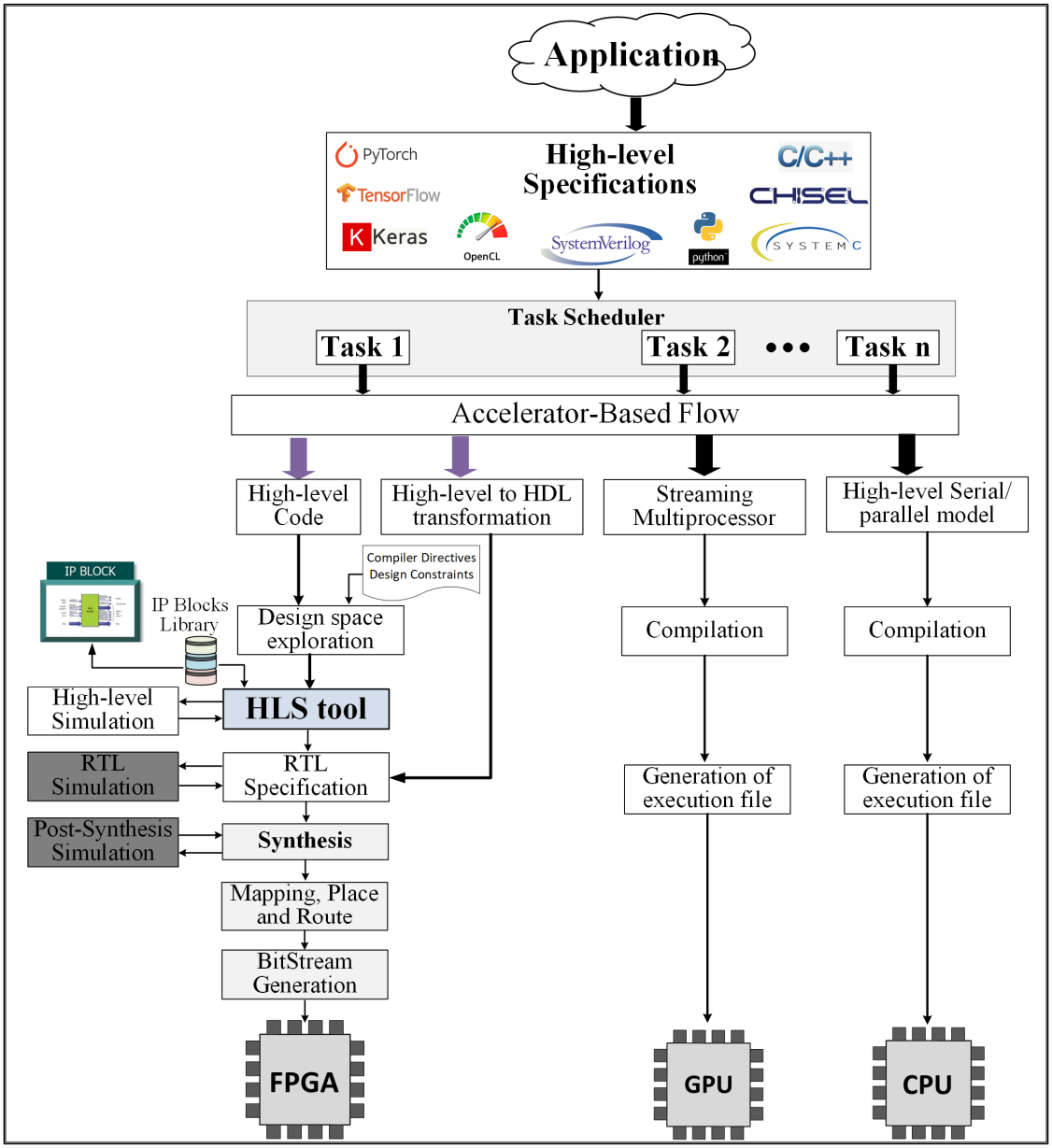


Figure 1.3).

1.2.2 Langage de programmation OpenCL

Les contributions scientifiques décrites dans cette thèse impliquent le langage OpenCL pour la description et la spécification d'applications pour les outils HLS. Cette section présente les

concepts de base d'OpenCL requis à la compréhension des contributions présentées dans cette thèse.

1.2.2.1 Structure générale d'une description d'application en OpenCL

OpenCL est un langage de programmation parallèle développé par Khronos Group pour la programmation des plateformes hétérogènes qui supportent des CPU, GPU et FPGA. Selon Aaftab (2011), une application développée avec OpenCL se compose généralement d'un hôte (CPU) et d'un ou plusieurs accélérateurs (GPU ou FPGA).

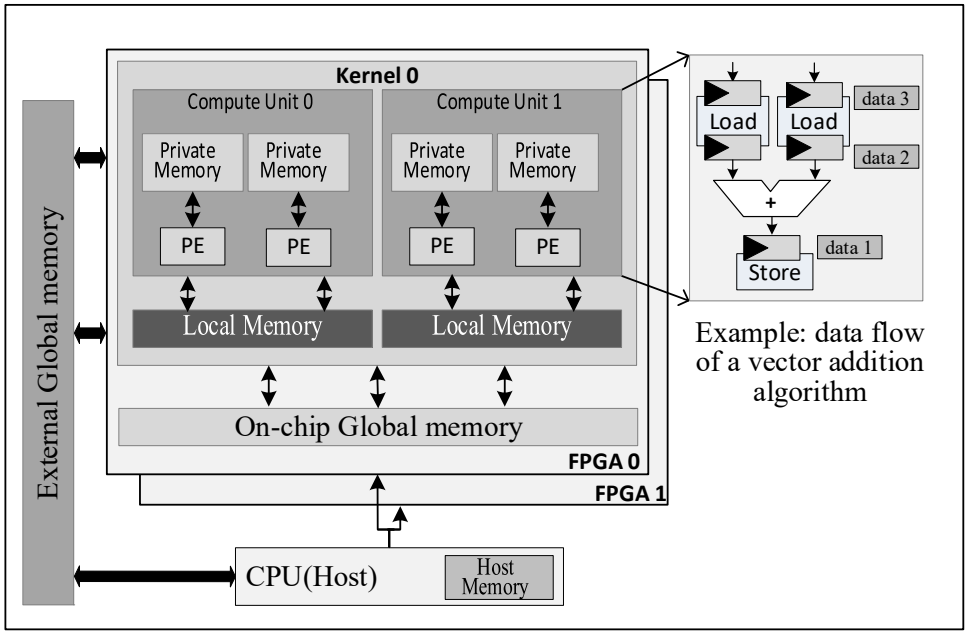


Figure 1.5 Modèle de programmation OpenCL avec FPGA
Reproduite et adaptée avec l'autorisation de Aaftab et al. (2011)

Les accélérateurs, appelés aussi *devices* sont configurés en utilisant les spécifications des noyaux OpenCL (*OpenCL kernels*). Ces spécifications sont décrites en C99, un langage dérivé du langage C. Chaque noyau est composé d'une ou plusieurs unités de calcul (*compute unit* (CU)), et chaque CU est divisée en un ou plusieurs éléments de traitement (*processing element* (PE)). Par exemple, une CU peut contenir une application d'addition de vecteurs avec trois

opérations : chargement (*load*), stockage (*store*) et addition (+). L'addition est décrite sous forme d'une structure flux de données (*dataflow*), comme le montre la Figure 1.5.

Dans une application décrite en OpenCL, plusieurs itérations relatives à une seule application peuvent s'exécuter en parallèle. Chaque exécution est appelée « *work-item* » ; un *work-item* correspond à une seule exécution du noyau OpenCL instancié dans un accélérateur. Plusieurs *work-items* peuvent être regroupés dans des groupes appelés « work-group ». Ces groupes sont exécutés simultanément sur le même accélérateur. En outre, un *work-item* peut être exécuté par un ou plusieurs PE dans le cadre d'un même groupe. De plus, OpenCL offre des fonctions pour l'interaction entre l'hôte et les accélérateurs, ainsi que pour la gestion des mémoires et des communications entre les accélérateurs intégrés dans une plateforme hétérogène (Waidyasooriya, Hariyama, & Uchiyama, 2018).

1.2.2.2 Noyaux OpenCL

Le flot typique du modèle d'exécution OpenCL se décrit comme suit : le processeur (hôte) réserve tout d'abord les espaces mémoire nécessaires pour allouer toutes les données dans sa propre mémoire, puis il prépare les entrées de chaque noyau « *kernel arguments* ». Les données sont ensuite transférées vers la mémoire de l'accélérateur via les fonctions offertes par OpenCL (citons à titre d'exemple les fonctions *clEnqueueWriteBuffer* et *clEnqueueMapBuffer*). Les noyaux de chaque accélérateur sont ainsi activés par le processeur hôte. Deux modèles d'exécution du noyau sont possibles : le modèle Single-work-item (SWI) et le modèle NDRange.

- **Modèle *NDRange***

Dans le modèle *NDRange*, plusieurs fils d'exécution, nommés *threads*, sont exécutés simultanément sur des PE. En effet, un *work-group* est généralement assigné à l'une des CU et chaque *work-item* est assigné à l'un des éléments de traitement (PE). Par défaut, l'utilisation du modèle *NDRange* obligera le compilateur à générer une seule unité de calcul, CU, mise en œuvre sur un pipeline, dont tous les *work-items* de tous les *work-groups* s'exécutent sur ce

pipeline. Si des barrières de synchronisation sont utilisées dans un noyau *NDRange*, les *threads* seront séparés par ces barrières pour synchroniser les itérations de boucle OpenCL advenant l'existence de dépendance de données entre les itérations.

- **Modèle *Single work-item***

Contrairement au modèle *NDRange*, le modèle *single work-item* suit un modèle d'exécution séquentiel dont l'ensemble des *work-items* sont exécutés par un seul pipeline. Cependant, s'il y a des boucles imbriquées dans le noyau, le compilateur peut les affecter à des pipelines séparés. La parallélisation des itérations peut être réalisée si les boucles sont déroulées, sauf s'il y a une dépendance de données.

1.2.2.3 Modèles de mémoires

OpenCL offre différents types de modèles de mémoires qui peuvent être accédés par l'hôte CPU ou par l'accélérateur. La mémoire liée au CPU est appelée *host memory* et celle liée à l'accélérateur est appelée *device memory*. La Figure 1.6 présente l'hiérarchie des différents modèles de mémoire offerts par OpenCL. Cette hiérarchie possède cinq types de mémoires.

- la mémoire de l'hôte « *host memory* » est définie comme la région de mémoire qui est accessible seulement par l'hôte ;
- la mémoire globale « *global memory* » est accessible par l'hôte et par l'accélérateur. L'accès à cette mémoire peut se faire par élément ou par paquet d'éléments ;
- la mémoire constante (*constant memory*) est une mémoire accessible par l'accélérateur en mode lecture seule ;
- la mémoire locale (*local memory*) est une mémoire interne à l'accélérateur et elle peut stocker un seul *work-group* spécifique. Les données stockées dans la mémoire locale sont partagées par les éléments du *work-group*, alors que ces données ne sont pas accessibles aux autres *work-groups* ;
- la mémoire privée (*private memory*) est une mémoire de petite taille accessible juste à un seul *work-item* et dont les données stockées ne sont pas partagées.

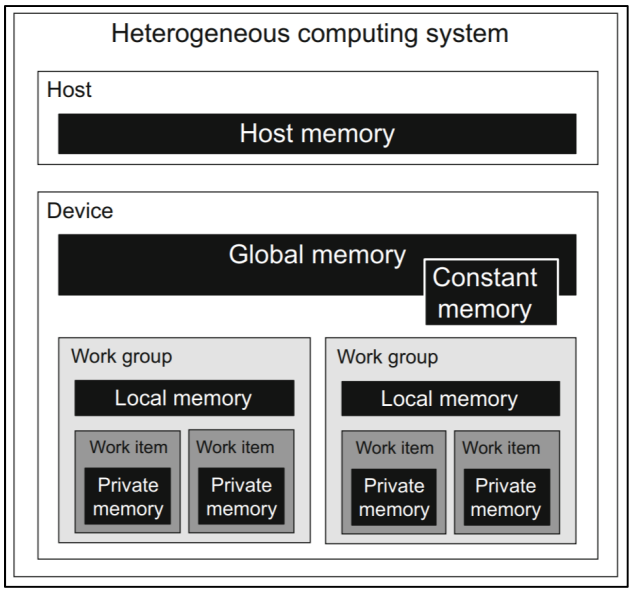


Figure 1.6 Modèles de mémoire d'OpenCL
Tirée de Waidyasooriya et al. (2018)

1.3 Méthodes d'instrumentation et d'analyse de performances sur FPGA

Comme expliqué dans la section 1.2.1.4, un des problèmes principaux des outils HLS est l'écart entre le niveau RTL et celui auquel l'application est décrite. Cet écart est causé par l'inefficacité des outils de simulation et par l'absence d'outils d'instrumentation et d'analyse à bas niveau. La capacité de ces outils ne permet pas d'exploiter au maximum le potentiel des accélérateurs FPGA. Les outils d'instrumentation en support aux outils HLS ont fait l'objet de plusieurs travaux de recherche dont différentes méthodes ont été proposées. L'instrumentation de circuits se définit dans cette thèse comme de la logique ajoutée au circuit implémenté dans un FPGA pour dévoiler son comportement interne. Ce comportement peut être observé par l'extraction de la valeur de signaux du circuit où ces derniers sont directement associés à des variables du langage de description de haut niveau. À ces valeurs peuvent aussi être attachées le moment de leur changement, principalement pour la détermination des performances tels le temps total d'exécution, la latence et l'intervalle d'initiation d'une boucle.

Cette section fait une synthèse des principales méthodes d'instrumentation proposées dans la littérature. Elle se limitera aux travaux qui ont traité le problème d'instrumentation des circuits

FPGA conçus avec les outils HLS. D'abord, la motivation qui alimente cet axe de recherche est présentée. Ensuite, les différentes méthodes d'instrumentation proposées dans la littérature sont détaillées.

Motivation de cette thèse : Limitations des outils commerciaux

Les outils HLS « Xilinx SDAccel » (Xilinx, 2018a) et « *Intel FPGA SDK for OpenCL »* (Intel, 2020a) prennent en charge la fonction printf comme une solution de débogage partielle. Cette fonction est utilisée lors d'émulation logicielle ou matérielle. L'émulation logicielle permet la vérification fonctionnelle d'un programme OpenCL sur un CPU, tandis que l'émulation matérielle permet la vérification fonctionnelle à l'aide d'un modèle logique et matériel précis de l'accélérateur utilisé. Cependant, ces émulations ne permettent pas de détecter les comportements inattendus des circuits lors de leur exécution dans les FPGA et permettent seulement la vérification et l'exactitude de la sortie du programme. Par ailleurs, l'outil « *Intel FPGA SDK for OpenCL* » fournit un rapport qui décrit les performances générales telles que les ressources logiques utilisées, le temps total d'exécution et la bande passante des mémoires externes. En revanche, ce rapport de performance ne permet pas d'identifier les goulots d'étranglement en termes de performances, telles que les décrochages du pipeline (*pipeline stall*). Le décrochage du pipeline appelé aussi « bulle » est un temps d'attente lors de l'exécution d'une instruction ou d'une opération. Le décrochage du pipeline est un problème qui est souvent présent dans les architectures matérielles des circuits conçus à base de pipeline et c'est le cas pour les spécifications OpenCL destinées aux FPGA (voir Section 1.2.2.2).

Ces limitations indiquent qu'il y a un besoin pour des instruments in situ qui permettent d'observer les données traitées par les circuits implémentées dans les FPGA, d'analyser leur déplacement et d'extraire précisément les performances temporelles des noyaux OpenCL. De ce fait, la solution à cette limitation est l'insertion d'instruments au sein des noyaux OpenCL permettant d'identifier les goulots d'étranglement ou de fournir des performances précises de chaque noyau ou fonction. Ces instruments peuvent être insérés dans le circuit à différents niveaux d'abstraction, majoritairement au niveau RTL ou à haut niveau d'abstraction.

Classification des méthodes d'instrumentation

Les méthodes d'instrumentation offertes avec les outils commerciaux ou fournies par les chercheurs pour utilisation académiques sont classifiées dans ce qui suit en quatre types :

1) méthode d'instrumentation par analyseurs logiques ;
2) méthodes d'instrumentation par modification du flot de compilation ;
3) méthodes d'instrumentation par transformation du code ;
4) méthodes d'instrumentation en utilisant les architectures virtuelles.

1.3.1 Instrumentation par analyseurs logiques (p. ex. ChipScope, SignalTap)

La visibilité à l'intérieur des FPGA est souvent offerte par les analyseurs logiques embarqués (*embedded logic analyzer* (ELA)), tels que l'analyseur SignalTap de Intel (Altera, 2013) et ChipScope de Xilinx (Xilinx, 2012b). Contrairement aux processeurs, le débogage des FPGA reste un défi surtout pour les circuits générés par les outils HLS puisque les analyseurs logiques offerts par les fournisseurs des FPGA ne peuvent être insérés qu'au niveau RTL, le niveau avec lequel les circuits sont souvent décrits avec un HDL. Ces analyseurs fournissent un support de débogage et sont utilisés pour dévoiler le comportement interne des circuits numériques pendant leur exécution dans un FPGA. En effet, les concepteurs doivent tout d'abord identifier les signaux qu'ils souhaitent observer à partir de la spécification décrite au niveau RTL, qui est généralement lisible, et ils doivent les connecter aux outils ELA. Une fois le FPGA configuré et le circuit en fonction, les ELA capturent et stockent les valeurs de signaux sélectionnés dans des mémoires. Ces mémoires sont souvent des mémoires internes aux FPGA (par exemple, des BRAM (Block RAM)).

Les ELA sont supportés par certains outils HLS, tels que les outils Xilinx SDx et SDAccel (Xilinx, 2018a), mais ils sont quasiment impossibles à utiliser pour deux principales raisons. Premièrement, la netlist générée par les outils HLS est illisible en raison de la nature générique et structurelle du circuit généré par ces outils. Les fichiers de description de matériel générés par ces outils sont nombreux et pas structurés pour être compréhensible avec des noms de signaux qui sont incompréhensibles pour un concepteur de matériel. Deuxièmement, il n'y a

aucune compatibilité entre l'identification des signaux décrits dans ces fichiers et celle des variables ou les types de données décrites dans la spécification de haut niveau. Par exemple, une seule variable décrite au code C peut correspondre à plusieurs signaux dans le circuit généré par les outils HLS. De plus, différents noms peuvent être attribués à ces signaux et ceci dépend non seulement des algorithmes métaheuristiques utilisés par l'outil HLS, mais aussi de la technologie du FPGA utilisé. Il s'est avéré difficile pour les concepteurs de faire correspondre les variables décrites à haut niveau avec les signaux de la netlist générée. Ces limitations augmentent l'intérêt des développeurs d'applications pour des instruments à haut niveau. Autrement, l'utilisation d'une telle approche permet l'insertion des instruments directement dans le code à haut niveau afin que les outils HLS puissent les intégrer dans les FPGA sans utiliser les outils ELA.

1.3.2 Instrumentation par modification du flot de compilation

La méthode d'instrumentation par modification du flot de compilation est strictement appliquée aux outils à code ouvert (*open sources*), car elle nécessite un accès aux codes sources des compilateurs. En effet, cet accès permet l'insertion d'instruments pendant l'étape de compilation en faisant une association (*mapping*) des états du circuit au code source. Cette solution a été proposée dans les travaux de Calagar et Anderson (2014) et Goeders et Wilton (2014) et elle a été appliquée à l'outil LegUp (Canis et al., 2013). Cette approche n'est pas possible avec les outils commerciaux tels que les compilateurs d'Intel ou Xilinx vu que les codes sources ne sont pas ouverts. Une technique semblable qui consiste à modifier les étapes de compilation est proposée par Bussa, Goeders et Wilton (2017) et appliquée à l'outil LegUp. Cette technique de compilation incrémentale a pour objectif de réduire le temps du débogage, ce qui permettra d'accélérer l'instrumentation des circuits générés par les outils HLS. Cependant, l'insertion des ports de débogage associés avec leurs circuits logiques augmente progressivement le temps de compilation, ce qui réduit la productivité de développement. Cette augmentation peut s'aggraver avec les compilateurs OpenCL puisque chaque étape de compilation prend plusieurs heures.

1.3.3 Instrumentation par transformation du code

En général, les instruments sont embarqués avec le circuit pour enregistrer le comportement interne des circuits lors de leurs exécutions dans les circuits FPGA tels proposés par les auteurs de ces articles (Y. K. Choi & Cong, 2017) (Hung & Wilton, 2011). L'instrumentation par transformation du code consiste à transformer la spécification de haut niveau vers un code de même niveau d'abstraction en ajoutant les ingrédients nécessaires à l'instrumentation tels les ports de débogage (J. P. Pinilla & Wilton, 2016). Dans le travail de Jose Pablo (2016), une méthodologie d'instrumentation à haut niveau d'abstraction est proposée. Ces méthodes sont appliquées aux circuits conçus avec Vivado HLS (Xilinx, 2021) et LegUp. En effet, le programme d'entrée qui est écrit en C est structuré tout d'abord sous forme d'un arbre de syntaxe abstrait (*Abstract syntax tree* ; AST). Une fois le code structuré, les instruments sont insérés par une conversion du code par un compilateur, nommé *source-to-source compiler*. Enfin, le code généré par ce compilateur est utilisé comme le code d'entrée à l'outil Vivado HLS ou LegUp pour générer la description RTL dédiée. Cette approche d'instrumentation adopte un flot d'instrumentation qui nécessite le passage par plusieurs étapes. De plus, cette technique est coûteuse en matière de consommation logique et de mémoires internes aux FPGA. Pour cette raison, les auteurs ont appliqué une technique d'optimisation de ressources appelée *array duplicate minimization* (ADM) pour réduire la consommation de ces mémoires. Par ailleurs, leur approche nécessite un outil supplémentaire pour générer la description sous-forme AST. En outre, cette technique de débogage permet d'extraire seulement les valeurs des variables définies dans le code C sans aucune information temporelle et l'absence d'informations temporelles ne permet pas d'identifier les problèmes temporels tel le décrochage de pipeline.

Une méthode semblable à celle de Jose Pablo et Wilton (2016) est publié par Monson et Hutchings (2018), (2015) et (2014). Leur méthode consiste à transformer un code C vers une spécification au même niveau d'abstraction que le C. Cette transformation a pour objectif d'ajouter des ports de débogage connectés à des expressions ou des variables dans le code C. Ces ports fournissent des points d'accès dans le code C et ouvrent un angle d'observation de

l'expression ciblée. Cette solution offre ainsi une solution de débogage des variables C à haut niveau par l'ajout des ports d'entrées/sortie dans le FPGA mais aucune information ou analyse temporelle n'est extraite. Un analyseur logique interne ou externe est aussi nécessaire pour pouvoir observer les signaux sur ces ports. Par ailleurs, une compilation et un flot supplémentaires sont aussi requis pour l'insertion des ports, ce qui augmente la complexité d'utilisation de cette méthode. De plus, cette solution est aussi coûteuse en termes d'occupation de ressources logiques et est étroitement liée aux outils HLS qui acceptent exclusivement le code C comme entrée. En conséquence, cette approche n'est pas compatible avec les outils OpenCL existants.

Dans l'article publié par Verma et al (2017), les auteurs proposent une plateforme pour le débogage et la surveillance des applications élaborées par OpenCL. Les fonctionnalités de cette plateforme sont limitées à la capture d'événements exécutés par l'unité de traitement (CPU ou FPGA) et à la détermination de leur séquencement. D'ailleurs, leur plateforme permet de détecter les décrochages que dans deux scénarios : lors des opérations de chargement ou de stockage accédant à la mémoire globale, ou dans les canaux de communications. De ce fait, cette plateforme ne fournit pas suffisamment d'informations temporelles pour permettre de caractériser les applications telles que les performances relatives aux boucles (intervalle d'initiation et latence). Aussi, elle ne permet pas d'analyser ni les causes de dégradation de performance ni les goulots d'étranglement possibles d'une application qui pourrait être décrite par un ou plusieurs noyaux OpenCL.

1.4 Accélération des algorithmes de hachage « Secure Hash Algorithm (SHA) » sur des plateformes reconfigurables

Le deuxième objectif de la thèse consiste à établir les bases pour l'amélioration des performances des accélérateurs élaborés en OpenCL ciblant les FPGA. Cette section introduit les algorithmes SHA-2 et SHA-3, utilisés dans la thèse pour mettre en œuvre les techniques d'optimisation explorées.

Compte tenu des différences architecturales majeures entre les FPGA, les CPU et les GPU, et l'absence d'outils permettant l'extraction des performances, il n'est pas évident de prédire la qualité des résultats (*quality of results* (*QoR*)) et les performances des applications élaborées en OpenCL lorsqu'ils sont implémentées sur FPGA. De plus, les optimisations typiques basées sur les architectures des CPU ou GPU ne sont pas applicables vu les différences architecturales entre les accélérateurs. Nous explorons les capacités d'optimisations des outils HLS en utilisant comme bancs d'essai les algorithmes de hachage (*Secure Hash Algorithm* (*SHA*)). Dans ce qui suit, les deux algorithmes de hachage utilisés (SHA-2 et SHA-3) sont décrits. Cette description est suffisamment détaillée pour permettre la compréhension de la troisième et quatrième contributions introduites dans la section 1.5.

Les algorithmes de hachage sont des fonctions cryptographiques largement utilisées pour assurer l'intégrité des données et l'authentification des messages. L'institut national de sciences et technologies (*National Institute of Science and Technology* (NIST)) a publié la première norme de hachage SHA-0 en 1993 sous la forme d'une norme de publication fédérale de traitement de l'information (NIST-FIPS, 1993), qui a été remplacée en 1995 par la norme SHA-1 (FIPS, 1995), puis par la famille SHA-2 (NIST, 2004) lors des révisions ultérieures de la norme SHA-1. Tandis que ces fonctions de hachage cryptographiques ont été attaquées par les pirates informatiques, le besoin de maintenir un certain niveau de sécurité a poussé le NIST à sélectionner une nouvelle fonction de hachage cryptographique par le biais d'une compétition publique. C'est lors de cette compétition que la nouvelle norme SHA-3 a vu le jour (NIST, 2015). Cette section présente l'état de l'art et les travaux présentés dans la littérature visant l'accélération des algorithmes SHA-2 et SHA-3 sur une cible FPGA. L'accélération vise à améliorer les performances de l'algorithme de hachage tels le débit (*throughput*) ou le temps total d'exécution.

1.4.1 SHA-2

L'algorithme SHA-2 est basé sur la structure de hachage *Merkle-Damgård* (MD) (Kaliski, 1990). Ces structures de hachage ont été proposées au cours des dernières décennies dont les

plus populaires sont de loin les fonctions de hachage de la famille MD4. MD4 est un algorithme de hachage développé par Ronald Rivest (1990), qui utilise des variables de 32 bits et des fonctions booléennes telles que les fonctions logiques ET, OU, XOR. La structure MD prend comme entrée un message de taille fixe et le divise en blocs de taille égale (Paar & Pelzl, 2009). Le message de sortie est ainsi calculé à l'aide d'une fonction de compression dédiée.

La famille SHA-2 comporte quatre fonctions de sortie fixe (SHA-224, SHA-256, SHA-384, SHA-512) et deux versions tronquées de la fonction SHA-512 (SHA512/224, SHA-512/256). Les versions SHA-224 et SHA-256 opèrent sur des messages de même taille (512 bits) qui est divisée sur 16 mots binaires de 32 bits chacun. Les versions SHA-384 et SHA-512 et les versions tronquées opèrent sur un bloc de 1024 bits divisé sur 16 mots de 64 bits. Cette section décrit principalement l'algorithme SHA-256 et elle présente les travaux antérieurs visant l'accélération de cet algorithme sur les plateformes intégrant des FPGA.

Présentation du SHA-256

SHA-256 génère une sortie de 256 bits à partir d'un message d'entrée d'une taille maximale égale à 2^{64} bits. SHA-256 effectue 64 cycles de hachage par bloc de message. Les principaux composants du SHA-256 sont les suivants :

- seize mots, chacun de taille 32 bits, pour stocker le message d'entrée de taille 512 bits ;
- soixante-quatre mots, chacun de taille 32 bits, nommé W=[W0…W63] ;
- huit variables, chacune de taille 32 bits, nommées a, b, c, d, e, f, g, et h ;
- deux variables temporaires nommées T1 et T2 ;
- huit variables, chacune de taille 32 bits, pour stocker la sortie de taille 256 bits.

Les six fonctions utilisées dans chaque cycle de hachage sont composées comme suit :

- les deux fonctions ∂_0 et ∂_1 sont utilisées pour calculer W et elles sont définies par les équations 1.1 et 1.2 :

$$\partial_0 (t) = \mathrm{ROTR}^7(t) \oplus \mathrm{ROTR}^{18}(t) \oplus \mathrm{SHR}^3(t) \quad (1.1)$$

$$\partial_1 (t) = \mathrm{ROTR}^{17}(t) \oplus \mathrm{ROTR}^{19}(t) \oplus \mathrm{SHR}^{10}(t) \quad (1.2)$$

La fonction $ROTR^n$ est une fonction de rotation qui pivote *n* fois à droite la variable *t*, tandis que la fonction SHR^k décale à droite le mot *t* par *k* bits.

- quatre fonctions : *Ch* (Choose), *Maj* (Majority), $\sum_0$, $\sum_1$, sont aussi utilisées pour calculer T1 and T2 et elles sont représentées par les équations 1.3, 1.4, 1.5 et 1.6 :

$$Ch\ (E_t, F_t, G_t) = (E_t \wedge F_t) \oplus (\neg E_t \wedge G_t) \quad (1.3)$$

$$Maj\ (A_t, B_t, C_t) = (A_t \wedge B_t) \oplus (A_t \wedge C_t) \oplus (B_t \wedge C_t) \quad (1.4)$$

$$\sum_0 (A_t) = ROTR^2(A_t) \oplus ROTR^{13}(A_t) \oplus ROTR^{22}(A_t) \quad (1.5)$$

$$\sum_1 (E_t) = ROTR^6(E_t) \oplus ROTR^{11}(E_t) \oplus ROTR^{25}(E_t) \quad (1.6)$$

où les symboles $\wedge$, $\neg$, et $\oplus$ représentent respectivement les opérations logiques AND, NOT et XOR. Ces opérations sont calculées bit à bit (bitwise).

Accélération du SHA-256 sur les circuits FPGA

Les algorithmes de hachage peuvent être implémentés sur CPU en utilisant une solution logicielle ou sur FPGA en utilisant une solution matérielle. Les implémentations sur FPGA offrent de meilleures performances comparées à celles sur CPU ou GPU (Wyant, Cullinan, & Frattesi, 2012). En effet, ces performances sont obtenues en appliquant de différentes techniques d'optimisation (Zeyad A. Al-Odat, Mazhar Ali, Assad Abbas, & Samee U. Khan, 2020). Parmi ces techniques, il y a l'insertion d'étages de pipeline. Cette technique est adoptée par plusieurs travaux publiés (Padhi & Chaudhari, 2017), (Tuan, Yamazaki, & Oyanagi, 2008), (R. Wu, Zhang, Wang, & Wang, 2020) tandis que d'autres travaux ont adopté des optimisations au niveau des circuits générés tels que l'optimisation des chemins critiques (Mohamed & Nadjia, 2015) ou encore la réorganisation des ressources matérielles utilisées (George S Athanasiou, Michail, Theodoridis, & Goutis, 2013). Citons à titre d'exemple l'ordonnancement des itérations de hachage proposé par Chen et Li (2020) pour réduire le chemin critique. Leur architecture du SHA-256, implémenté sur un FPGA Virtex-4, atteint un débit de 1984 Mbps. Le déroulement des boucles est aussi une technique d'optimisation appliquée au niveau algorithmique, tel que proposée par Aisopos et al (2005).

La majorité de ces travaux antérieurs ont adopté des optimisations applicables au niveau RTL en utilisant un langage de description matériel. Ces accélérations du SHA-256 sont souvent accompagnées par une évaluation de l'impact de ces optimisations sur la consommation logique et énergétique (Martino & Cilardo, 2020). Cependant, il n'est pas possible d'adopter de telles techniques quand il s'agit d'une description du haut niveau du SHA-256 telle une description OpenCL. Les circuits générés par les compilateurs OpenCL ciblant des FPGA sont structurés automatiquement par la technologie des algorithmes implémentés par ces compilateurs.

1.4.2 SHA-3

L'algorithme SHA-3 fut développé par Guido Bertoni et al. (2011), normalisé par le NIST et ses spécifications furent publiées en 2015 dans FIPS PUB (NIST, 2015).

Description de l'algorithme

Le SHA-3 offre six fonctions de hachage cryptographiques : quatre fonctions de sortie fixes (SHA-224/256/384/512) et deux fonctions de sortie extensibles (SHAKE-128 et SHAKE-256). Sa fonction de hachage principale est appelée Keccak (Bertoni, Daemen, Peeters, & Van Assche, 2009), qui est basée sur une structure appelée « éponge ». Cette structure est composée de deux phases : une première phase d'absorption dans laquelle l'entrée est absorbée. Une deuxième phase d'essorage dans laquelle la sortie est générée, telle que présentée par la Figure 1.7.

La structure éponge reçoit une entrée de taille arbitraire et génère une sortie de taille désirée. Les tailles des entrées et sorties ainsi que le niveau de sécurité de la fonction Keccak peuvent être configurés selon sur plusieurs paramètres. En effet, la fonction *Keccak* opère sur un état de taille b (b=1600 bits) = r + c, où « *r* » est le débit binaire (*bit-rate*), qui représente la quantité de bits traitée à chaque itération, et « *c* » est la capacité (Tableau 1.1Tableau 1.1 Spécification du SHA-3
Tirée de Sideris, Sanida & Dasygenis (2020)**Erreur ! Source du renvoi introuvable.**). Une

entrée doit tout d'abord passer par une étape de prétraitement appelée *padding* avant qu'elle soit traitée par SHA-3. Cette étape de prétraitement divise l'entrée en un nombre pair de blocs de même taille, nommés (M_1, ..., M_t) dans le schéma de la Figure 1.7. Ces blocs sont ainsi combinés par un traitement XOR puis traités par la fonction ***f*** durant la phase d'absorption (Figure 1.7).

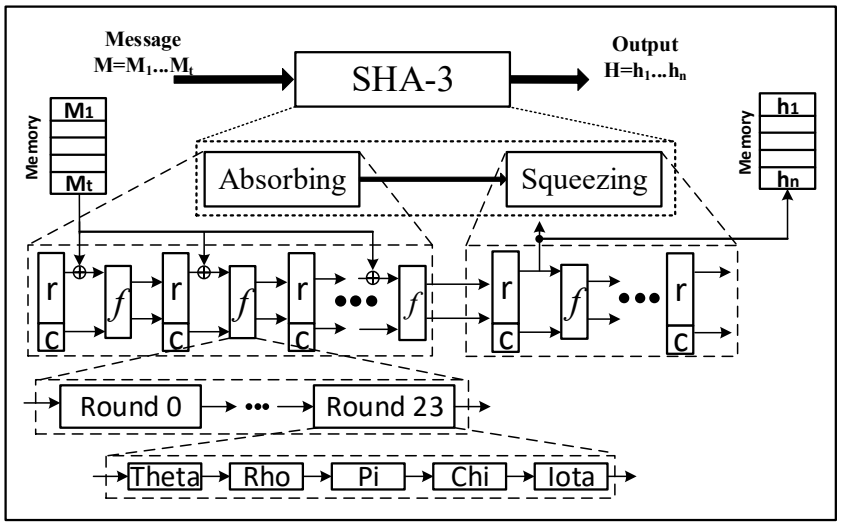


Figure 1.7 La structure du SHA-3
Reproduite et adaptée avec l'autorisation de Paar et Pelzl (2010)

Tableau 1.1 Spécification du SHA-3
Tirée de Sideris, Sanida & Dasygenis (2020)

	Taille de sortie (bits)	**r (bits)**	**c (bits)**	**b (bits)**	**Niveau de sécurité (bits)**
SHA-224	224	1152	448	1600	112
SHA-256	256	1088	512	1600	128
SHA-384	384	832	768	1600	192
SHA-512	512	576	1024	1600	256
SHAKE128	*d*	1344	256	1600	Min(*d*/2, 128)
SHAKE256	*d*	1088	512	1600	Min(*d*/2, 256)

La fonction *f* consiste en 24 itérations (rounds) dont cinq étapes de calcul sont effectuées dans chacune de ces itérations : Thêta (θ), Rho (ρ), Pi (π), Chi (χ) et Iota (ι), comme le montre la Figure 1.7. Ces étapes sont décrites par les équations (1.7) à (1.13).

- θ-step

$$C[i] = A[i,0] \oplus A[i,1] \oplus A[i,2] \oplus A[i,3] \oplus A[i,4] \qquad 0 \le i \le 4 \qquad (1.7)$$

$$D[i] = C[i-1] \oplus \text{Rotate}\,(C[i+1],1) \qquad 0 \le i \le 4 \qquad (1.8)$$

$$A'[i,j] = A[i,j] \oplus D[i] \qquad 0 \le i \le 4 \qquad (1.9)$$

- ρ-step

$$A[i,j] = \text{Rotate}\,(A'[i,j], r[i,j]) \qquad 0 \le i, j \le 4 \qquad (1.10)$$

- π-step

$$B[j, 2i+3j] = A[i,j] \qquad 0 \le i, j \le 4 \qquad (1.11)$$

- χ-step

$$A[i,j] = B[i,j] \oplus ((\neg B[i+1, j]) \wedge B[i+2, j]) \qquad 0 \le i, j \le 4 \qquad (1.12)$$

- ι-step

$$A[0,0] = A[0,0] \oplus RC[i] \qquad 0 \le i, j \le 4 \qquad (1.13)$$

Trois matrices (A, B et C) sont utilisées pour le calcul de chaque étape. A est une matrice bidimensionnelle, de taille (5x5) éléments dont chacun de ces éléments est taille 64 bits. La matrice A représente l'état *b* (*b*=1600). Les trois matrices B, C et D sont des matrices intermédiaires nécessaires au calcul des équations, comme détaillées dans Paar et Pelzl (2010). Le symbole $\oplus$ désigne l'opération XOR au niveau bit. L'opération *Rotate* fait une rotation cyclique qui déplace le bit de position « *i* » vers la position « *i+r* » (Latif, Rao, Mahboob, & Aziz, 2012).

La première étape Thêta (θ) (Figure 1.7) manipule la matrice A[*i*,*j*] et les deux matrices unidimensionnelles C [*i*] et D [*i*] selon les équations 1.7 à 1.9. La séquence des deux étapes ρ et π calculent la matrice B à partir de la matrice d'état A tandis que dans l'étape Chi, A [*i*,*j*] est régénéré encore une fois selon l'équation (1.12). Lors du calcul à la dernière étape (Iota), une constante RC(*i*) est ajoutée au premier élément de la matrice A. Une valeur spécifique RC(*i*) est attribuée dans chaque itération. Une fois les 24 itérations exécutées, la sortie de hachage H (*digest message*) est générée en lui associant les nouveaux éléments de la matrice A.

Accélération du SHA-3 sur FPGA

Plusieurs travaux antérieurs ont traité le problème d'accélération et d'amélioration des performances du SHA-3 en proposant des solutions visant son implémentation sur FPGA. Ces travaux peuvent être classifiés en deux catégories : une accélération du SHA-3 au niveau du matériel en modifiant l'architecture RTL, en utilisant un langage matériel et une accélération au niveau algorithmique en utilisant un outil HLS. L'accélération de l'algorithme SHA-3 sur FPGA en utilisant un langage matériel a été traité par la majorité des travaux publiés (Sideris et al., 2020), (Fatma Kahri, Mestiri, Bouallegue, & Machhout, 2016), (G. S. Athanasiou, Makkas, & Theodoridis, 2014), (Arribas, 2019), (Wong, Haj-Yahya, Sau, & Chattopadhyay, 2018), (Assad, Elotmani, Fettach, & Tragha, 2019) et (Zeyad A Al-Odat, Mazhar Ali, Assad Abbas, & Samee U Khan, 2020). Tous ces travaux proposent des implémentations accompagnées d'une série d'optimisations qui se limitent essentiellement à l'ajout d'étages de pipeline entre les opérations ou les étapes de calcul et certainement proposent l'optimisation des circuits arithmétiques utilisés par le SHA-3. Toutes ces techniques d'optimisation proposées nécessitent non seulement une compréhension approfondie de l'architecture matérielle implémentée, mais aussi du circuit FPGA utilisé ainsi que les cellules logiques qui l'offrent. De plus, toutes ces modifications sont faites au niveau RTL, ce qui constitue un obstacle à leur adoption dans une approche de conception en utilisant un outil HLS et un langage tel que l'OpenCL puisque l'architecture au niveau RTL est générée par le compilateur.

Des implémentations du SHA-3 basées sur la conception conjointe matérielle/logicielle exploitant un système sur puce (SoC), le FPGA Zynq de Xilinx (Rajagopalan, Boppana, Dutta, Taylor, & Wittig, 2011), ont été proposées par Farahmand et al. (2016) et Jacinto et al. (2017). Bien que ces deux travaux utilisent un outil HLS et une spécification du SHA-3 décrite en langage C, les résultats rapportés montrent une amélioration des performances du SHA-3. En revanche, leurs solutions dépendent exclusivement de l'architecture du SoC, le FPGA Zynq de Xilinx, et elles exploitent un processeur ARM intégré dans le circuit FPGA. L'approche de conception conjointe exploitant un processeur ARM n'est portable que sur des FPGA de même type et ne peut pas être utilisée avec d'autres plateformes.

1.5 Contributions de la thèse : Méthodes d'instrumentation et d'optimisation des accélérateurs sur FPGA élaborés en OpenCL

L'analyse des travaux antérieurs sur la synthèse de haut niveau (section 1.2.1) et les méthodes d'instrumentation (section 1.3) révèle que les caractéristiques et les limites de l'instrumentation des circuits produits par les outils HLS dépendent de deux principaux facteurs. Le premier concerne le niveau d'abstraction de la spécification du circuit à instrumenter puisque les instruments insérés doivent être adaptés à ce même niveau d'abstraction. Par exemple, l'instrumentation au niveau RTL ne fournit pas des analyses de performances lorsque les circuits sont générés avec les outils HLS. Ainsi, l'instrumentation par les analyseurs logiques ne permet pas d'avoir une visibilité des performances sein des accélérateurs élaborés en OpenCL. Le second facteur est la technologie et la nature des outils HLS. En effet, l'instrumentation par modification du flot de compilation nécessite un accès à la technologie *front-end* ou *back-end* des compilateurs et dans ce cas il faut que les outils soient totalement ou partiellement à code ouvert. Cependant, les outils commerciaux sont totalement à code fermé et le flot fournit ne peut pas être modifié ou adapté. Vu que les méthodes citées précédemment ne sont pas applicables avec le flot de conception OpenCL pour FPGA, cette thèse propose une nouvelle méthode d'instrumentation par transformation du code en ajoutant des instruments directement au niveau OpenCL.

Rappelons tout d'abord que l'objectif principal de cette thèse est d'améliorer la productivité de développement des applications élaborées en OpenCL mise en œuvre sur des plateformes hétérogènes reconfigurables. Plus précisément, les deux objectifs secondaires suivants :

1) améliorer l'observation et la mesure d'événements temporels, tel le décrochage du pipeline, dans les accélérateurs de type FPGA élaborés en OpenCL ;
2) établir les bases pour l'amélioration des performances des accélérateurs de type FPGA pour des applications décrites en OpenCL et vers la conception d'un outil d'optimisation automatisé.

Pour répondre à ces objectifs, deux directions principales sont abordées par cette thèse : Premièrement, l'instrumentation des accélérateurs élaborés en OpenCL et générés avec des outils HLS en visant la proposition d'une solution permettant l'instrumentation et le débogage de ces accélérateurs. Deuxièmement, cette thèse contribue principalement à l'automatisation des optimisations qui pourront être appliquées par les outils HLS pour FPGA (spécifiquement les outils qui acceptent OpenCL comme *INTEL FPGA SDK for OpenCL*) par la proposition de techniques d'optimisation visant des accélérateurs élaborés en OpenCL et la mise en œuvre des plateformes hétérogènes de type CPU+FPGA. Ce manuscrit décrit les contributions suivantes :

1) une preuve de concept d'un circuit permettant l'analyse in situ de performances temporelles d'un noyau OpenCL implémenté sur un FPGA ;
2) une nouvelle plateforme d'instrumentation pour extraire les performances temporelles des noyaux OpenCL ;
3) exploration de techniques d'optimisation en OpenCL et évaluation leurs impacts lorsqu'utilisées pour améliorer les performances de l'algorithme SHA-2 sur FPGA ;
4) une implémentation efficace sur FPGA d'un coprocesseur SHA-3 élaboré en OpenCL. Cette implémentation implique l'exploration et l'évaluation des différents choix de modèles d'exécution OpenCL. Ces diverses solutions appliquent aussi un ensemble de techniques d'optimisation autres que celles proposées pour SHA-2.

Les contributions présentées dans cette thèse ont mené à la publication d'un article de revue et deux articles présentés dans des conférences IEEE. Un deuxième article de revue est aussi soumis. Plus précisément, ces quatre articles peuvent être décrits selon les objectifs et classifiées comme suit :

1) Une nouvelle plateforme d'instrumentation pour les applications élaborées en OpenCL mises en œuvre sur une plateforme de type FPGA+CPU (objectif (1)).

- comme première contribution, une solution pour l'analyse in situ de performances temporelles d'un noyau OpenCL implémenté sur FPGA est proposée. Cette solution consiste à insérer dans le noyau des circuits de moniteur qui permettent de mesurer les performances temporelles du noyau en question. Ce moniteur in situ est inséré au sein du

noyau. Pour ce faire, deux moniteurs de mêmes fonctionnalités sont développés et leurs performances de mesure sont comparées : un premier moniteur est développé à haut niveau d'abstraction avec OpenCL tandis que le deuxième est développé au niveau RTL en Verilog. Les résultats montrent que le moniteur développé au niveau RTL est plus précis et consomme moins de ressources logiques sur FPGA. Une approche de conception conjointe matérielle/logicielle a permis l'insertion de ce moniteur (celui qui est développé au niveau RTL) au niveau OpenCL. Ce dernier moniteur permet aux développeurs d'application sans expertise avec les flots de conception matériels d'extraire les performances avec la précision du cycle d'horloge du FPGA. Cette contribution a mené à la publication de l'article intitulé « ***Toward In-System Monitoring of OpenCL-Based Designs on FPGA »*** présenté en 2019 à la conférence ISCAS (Bensalem, Blaquière, & Savaria, 2019);

- comme deuxième contribution, une nouvelle plateforme d'instrumentation pour extraire les performances temporelles des noyaux OpenCL est proposée. Pour ce faire, une modélisation de performance et d'insertion d'instruments est tout d'abord proposée pour identifier le nombre d'instruments utilisés pour l'instrumentation des noyaux. Lors de l'utilisation de la plateforme, des instruments sont insérés directement dans le code OpenCL et sont synthétisés par la suite avec le noyau tel que présenté par la première contribution. La plateforme permet d'identifier les noyaux critiques et d'enregistrer en temps réel la valeur de toute variable utilisée au sein des noyaux OpenCL. Une évaluation de la plateforme proposée est effectuée sur un grand ensemble d'applications. Les résultats obtenus montrent que la plateforme consomme entre 1,5 et six fois moins de ressources que celles rapportées dans les meilleurs travaux de recherche déjà publiés. Cette contribution a fait l'objet d'un article de revue IEEE Access intitulé « ***In-FPGA Instrumentation Framework for OpenCL-Based Designs*** » (Bensalem, Blaquière, & Savaria, 2020).

2) Évaluation des techniques d'optimisation pour améliorer la performance des accélérateurs élaborés en OpenCL (objectif (2)).

- comme troisième contribution, des techniques d'optimisation basées sur OpenCL sont proposées et évaluées par l'implémentation d'algorithmes sur FPGA. Pour ce faire, ces

techniques sont appliquées pour accélérer différentes versions de l'algorithme SHA-256. Huit implémentations qui appliquent une série de techniques d'optimisation au niveau OpenCL ont été développées pour étudier l'impact de chaque technique sur la vitesse, la fréquence d'opération et l'utilisation des ressources du FPGA. Par conséquent, les expérimentations rapportées permettent de montrer l'efficacité des techniques d'optimisation proposées pour l'amélioration des performances de l'accélérateur SHA-256 conçus avec OpenCL. Les résultats ont montré que l'implémentation OpenCL de SHA-256 offre un débit de traitement de données de 3.97 Gpbs qui est 4.3 fois plus élevé que le meilleur débit reporté dans les travaux publiés dans la littérature et implémentées par des outils HLS. Cette contribution a été publiée à la conférence ISCAS 2021, intitulé « ***Acceleration of the Secure Hash Algorithm-256 (SHA-256) on an FPGA-CPU Cluster Using OpenCL*** » (Bensalem, Blaquière, & Savaria, 2021).

- dans la quatrième contribution, la même méthodologie que celle de la troisième contribution est utilisée, mais pour une implémentation d'un coprocesseur SHA-3 élaboré avec OpenCL. Cette implémentation applique un plus grand ensemble de techniques d'optimisation, en particulier le modèle d'exécution multi-noyaux. En effet, sept implémentations élaborées avec OpenCL exploitant ces nouvelles techniques sont mises en œuvre sur une plateforme FPGA. Les résultats de performance de chaque implémentation, tels que la vitesse et le débit, sont ensuite présentés. En conséquence, l'implémentation la plus performante du coprocesseur SHA-3 est finalement proposée. Les résultats ont montré que l'implémentation OpenCL de SHA-3 offre un débit de traitement de données de 22.36 Gbps qui est 2 fois plus élevé que les meilleures implémentations publiées dans la littérature et faites avec des outils HLS Cette contribution a mené à la soumission d'un article de revue IEEE intitulé « ***An Efficient OpenCL-Based Implementation of a SHA-3 Co-Processor on an FPGA-Centric Platform*** ».

1.6 Conclusion

Ce chapitre a fait une revue de littérature critique concernant les axes de recherche de cette thèse. Cette revue a identifié les limites et lacunes des environnements et des approches HLS

existantes pour optimiser la performance d'applications à l'aide d'accélérateur sur FPGA : en particulier l'absence d'instruments facile à utiliser par l'utilisateur et précis au cycle d'horloge du FPGA ainsi que la complexité associée à l'exploitation des techniques d'optimisation disponible dans ces environnements. Ce chapitre de revue a ainsi positionné les travaux et contributions apportés par cette thèse.

CHAPITRE 2

MÉTHODOLOGIE ET PLATEFORME D'INSTRUMENTATION PROPOSÉE POUR DES ACCÉLÉRATEURS SUR FPGA ÉLABORÉE EN OPENCL

Ce chapitre présente les deux premières contributions de cette thèse, dont une méthode d'instrumentation, la preuve de concept d'un instrument ainsi que la plateforme d'instrumentation. La section 2.2 comprend une validation de la preuve de concept de l'instrument, appelé moniteur, une description du flot de conception mis en œuvre ainsi qu'une comparaison des résultats avec les travaux antérieurs. La section 2.3 expose la plateforme d'instrumentation proposée. En premier lieu, les étapes requises pour l'instrumentation sont décrites, suivies des modèles de performance supportés par la plateforme. Ces modèles permettent entre autres d'identifier le nombre d'instruments à insérer dans le noyau OpenCL pour mesurer certaines performances temporelles. En deuxième lieu, les capacités et la validité de cette plateforme sont montrées par la présentation de résultats de mesures faites sur 50 variantes d'applications instrumentées. Finalement, ces résultats sont comparés avec l'état de l'art puis d'en tirer une conclusion relative à cette contribution.

2.1 Problématiques et défis

L'approche HLS vise à donner au développeur logiciel une vision abstraite du matériel au détriment d'une méconnaissance des détails de la logique implémentée. La principale difficulté pour un développeur logiciel est d'optimiser la performance de son application avec cette méconnaissance des unités de calcul et des mémoires inférées. En outre, les circuits générés par les outils HLS sont ainsi non lisibles, ce qui rend l'étape d'identification des parties critiques du circuit dans le FPGA très difficile. En effet, ces circuits sont totalement différents de ceux décrits au niveau OpenCL, tant au niveau de leur structure qu'au niveau des variables et des fonctions qu'ils contiennent. Par conséquent, la compréhension de ces circuits générés devient très difficile même par les experts en langage matériel.

L'absence d'un outil efficace d'instrumentation ou de débogage rend l'identification des goulots d'étranglement quasi impossible dans les implémentations matérielles à base d'OpenCL. Les performances générées par les compilateurs OpenCL actuels sont générales, comme la bande passante de transfert sur le bus PCI ou le temps total d'exécution calculé avec le CPU. De plus, les outils de simulation offerts par les fournisseurs de FPGA permettent une simulation matérielle des circuits générés sans aucun lien ou identification avec le code OpenCL. Par exemple, les valeurs observées d'un signal n'ont aucune signification dans le code OpenCL. Une variable définie dans le noyau OpenCL peut correspondre à plusieurs signaux au niveau RTL et son contenu peut être stocké dans plusieurs types de mémoires choisis par le compilateur lors de la synthèse. Le type de ces mémoires est déterminé selon le mode de déclaration de la variable dans le code OpenCL. De plus, les performances détaillées comme celle des boucles (*II* ou latence), le temps d'exécution associé à chaque élément dans le noyau OpenCL comme les canaux de communication et les unités de calcul dans un noyau sont complètement invisibles aux développeurs. Pour ces raisons, l'insertion d'instruments au niveau OpenCL sans intrusion dans l'outil HLS et en faisant abstraction du circuit généré dans le FPGA présente une méthode prometteuse pour l'identification des parties critiques du code du noyau OpenCL. Le principal défi est de développer une solution qui permette d'extraire précisément les performances temporelles du noyau comme celles des boucles et d'identifier les bogues fonctionnels et les goulots d'étranglement.

2.2 Développement d'un moniteur au niveau RTL

Cette section présente la méthode d'instrumentation par l'intégration d'un moniteur développé au niveau RTL et intégré dans le noyau OpenCL pour fournir les caractéristiques temporelles des opérations exécutés dans le noyau et leurs temps critiques. Les performances de deux noyaux OpenCL *NDRange* et *single work-item* sont caractérisés avec cette méthode d'instrumentation.

2.2.1 Concept général d'instrumentation des noyaux OpenCL

L'instrumentation des noyaux OpenCL consiste à insérer des circuits qui permettent d'extraire les performances d'un code. Citons à titre d'exemple, les données relatives à un évènement, la valeur d'une variable ou les performances d'une section du code dans le noyau OpenCL. La Figure 2.1 illustre le concept général de l'instrumentation où une fonction appelée *instrument* (*inputs*) est insérée dans le noyau OpenCL pour mesurer les performances désirées. Les données sont stockées dans une mémoire dans le noyau, identifiée par la variable OpenCL *Mport* dans cet exemple, et transférées vers la mémoire globale du CPU, externe au FPGA. Le CPU peut afficher ou transférer ces données pour un post traitement pour calculer des performances temporelles, après la lecture des mémoires globales qui y sont attachées. Deux méthodes de développement d'instruments sont possibles : avec le langage OpenCL seulement ou avec les langages OpenCL et matériel au niveau RTL. Cette dernière méthode est possible vu que le flot OpenCL pour FPGA supporte la conception conjointe matérielle/logicielle. Dans ces méthodes, les instruments sont insérés dans le noyau OpenCL à instrumenter. L'infrastructure logicielle qui permet l'intégration d'instruments dans un noyau OpenCL avec la méthode de conception conjointe matérielle/logicielle est décrite dans la section 2.2.2. Cette méthode a été appliquée à un simple circuit multiplicateur accumulateur (MAC) comme étude de cas dont les résultats sont résumés à la section 2.2.3.

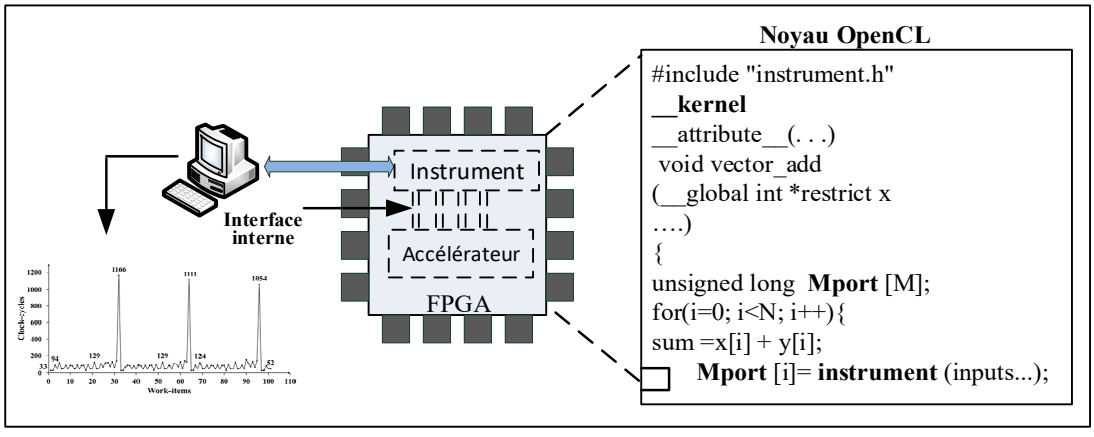


Figure 2.1 Vue d'ensemble de l'intégration d'un moniteur (instrument) dans un FPGA élaboré en OpenCL

2.2.2 Intégration d'un moniteur développé au niveau RTL

2.2.2.1 Architecture RTL du moniteur

L'architecture du moniteur implémenté est présentée par la Figure 2.2, composée d'un compteur, appelé *Timer*, et de la logique nécessaire pour l'interface interne au FPGA. Cette logique d'interface est expliquée dans la section 2.2.2.2. Le *Timer* est connecté en amont et en aval avec une interface de communication interne propre aux FPGA d'Intel nommée *Avalon Streaming Interface* « Avalon-ST » (Intel, 2021a). Ce moniteur est décrit en Verilog et son intégration dans une librairie usager exige la création d'autres modules complémentaires nécessaires à son interfaçage avec le noyau OpenCL. Ces modules sont détaillés dans la section 2.2.2.2. L'entrée du circuit moniteur, appelée « inputs » dans la Figure 2.2, est l'entrée à instrumenter tandis que les signaux appelés « Controls » sont des signaux de l'interface interne au FPGA qui permettent le contrôle de données transmises par le moniteur.

Les données extraites par les instruments sont stockées dans des tampons de mémoires, appelées *trace buffers*. Ces *trace buffers* peuvent être implémentés dans des mémoires internes au circuit FPGA, tel que présenté dans l'architecture de l'instrument illustrée à la Figure 2.2, ou dans des mémoires externes au FPGA. Le développeur peut choisir de stocker les données extraites dans une mémoire externe lorsque la taille de données est supérieure à celle des mémoires disponibles dans le circuit FPGA.

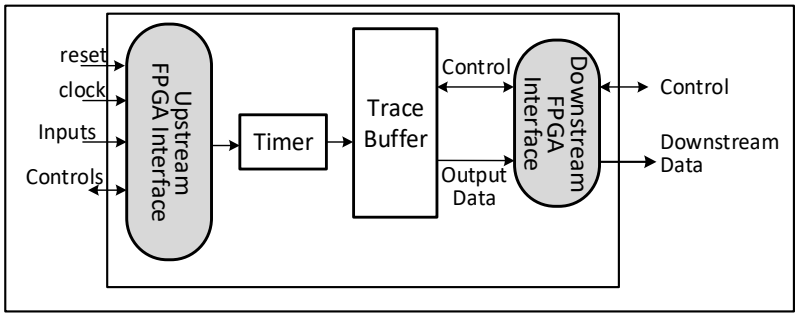


Figure 2.2 L'architecture RTL du moniteur

2.2.2.2 Interface de communication entre le moniteur et le noyau OpenCL

La communication entre le moniteur (développé au niveau RTL) et le noyau OpenCL est prise en charge par les interfaces internes du FPGA. Ces interfaces dépendent de type du FPGA utilisé. Les deux interfaces internes majoritairement utilisées sont l'interface AXI dans les FPGA de Xilinx (Xilinx, 2012a) et l'interface *Avalon-ST* (Intel, 2021a) dans les FPGA d'Intel. La communication en amont et aval entre le circuit RTL et le noyau instrumenté permet de contrôler le processus de capture des données (Figure 2.3). L'interface Avalon-ST reçoit comme entrées les arguments de la fonction d'instrumentation, qui sont spécifiés dans le noyau OpenCL. Ces flux d'entrées sont contrôlés par les signaux de contrôle spécifique à l'interface *Avalon-ST*, « ivalid », « iready », « ovalid » et « oready ». Ces signaux de contrôle sont utilisés pour sélectionner et valider les données à capturer et à transmettre. La transmission de données par l'interface est détaillée dans le document de référence de Intel (Intel, 2021a).

Des modules nécessaires à l'interfaçage entre le moniteur et le noyau avec le noyau OpenCL sont aussi développés. Ces modules sont présentés par un fichier XML (*eXtensible Markup Language*) et une bibliothèque logicielle (*header file*) avec une extension « .h ». Cette dernière contient le nom de la fonction d'instrumentation qui doit être appelée dans le noyau OpenCL. Citons à titre d'exemple, le nom « *instrument* (*inputs*) » dans le code de la Figure 2.1.

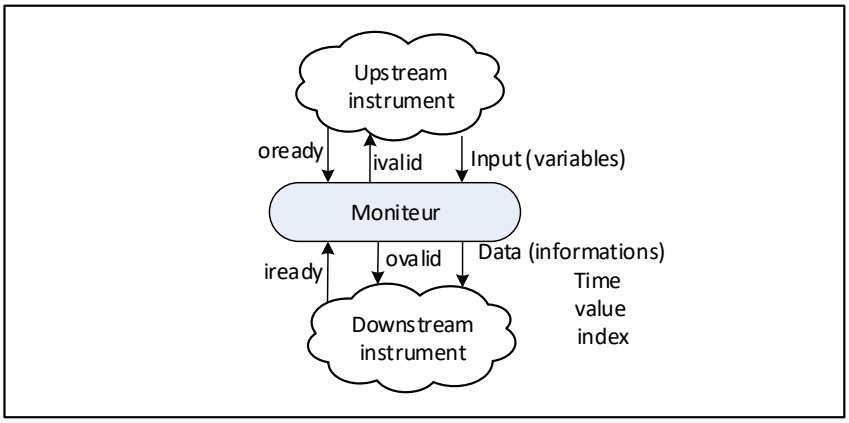


Figure 2.3 L'interaction entre le moniteur et le noyau OpenCL
Reproduite et adaptée avec l'autorisation de Intel (2020a)

2.2.2.3 Flot de conception d'OpenCL pour FPGA

Le flot de conception OpenCL pour les FPGA d'Intel est utilisé pour les implémentations faites dans ce travail (Figure 2.4). Ce flot est propre à l'outil *Intel FPGA SDK for OpenCL* et il est compatible avec la version du modèle OpenCL 1.0 et certaines des fonctionnalités des versions 1.1 et 1.2 (Sean, 2013). Le compilateur d'Intel compile le code du noyau (.cl) pour fournir le fichier de configuration des FPGA (.aocx). Le code logiciel destiné à l'hôte (un CPU) est compilé en utilisant le compilateur GCC sous Linux ou Microsoft Visual Studio et il génère un fichier exécutable (.exe). L'ensemble des fichiers créés pour l'instrumentation est appelé bibliothèque d'instrumentation (appelée *Monitor library* dans la Figure 2.4). Cette bibliothèque est définie par un fichier XML, un fichier d'en-tête logicielle (.h) et des fichiers Verilog ou VHDL. Une fois compilée, la bibliothèque d'instrumentation est ensuite liée au noyau OpenCL, tel que présenté par la Figure 2.4.

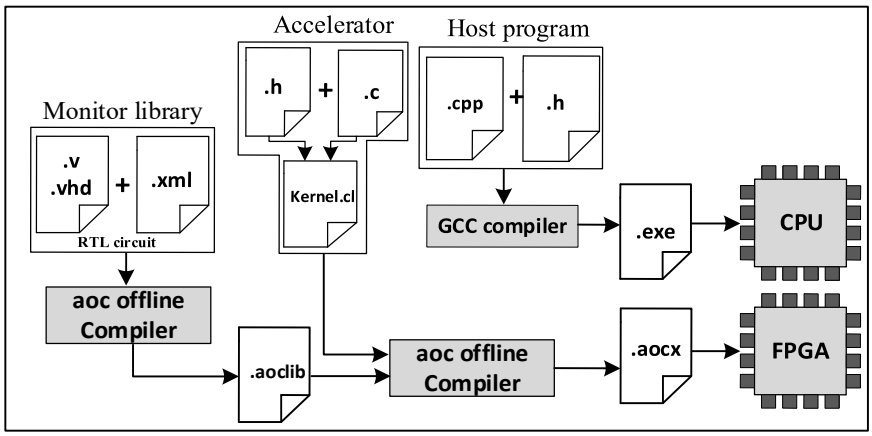


Figure 2.4 Flot de conception du *Intel FPGA SDK for OpenCL*
Reproduite et adaptée avec l'autorisation de Intel (2020a)

2.2.2.4 Plateforme CPU+FPGA

Rappelons que la carte FPGA Nallatech 510T (Nallatech, 2021) est utilisée dans ce travail pour valider le moniteur ainsi que la plateforme d'instrumentation proposée (décrite par la Section 2.2). La plateforme CPU+FPGA est illustrée par la Figure 1.2 (page 12), qui comprend deux FPGA Arria 10-1150 GX et quatre bancs de mémoire de 4 Go par FPGA (32 Go au total)

installés sur une machine Dell à l'aide d'une interface PCIe (3.0). L'outil *Intel FPGA SDK for OpenCL* v17.1 est utilisé pour les implémentations faites dans les travaux présentés dans cette thèse.

2.2.3 Étude de cas : Multiplicateur-accumulateur (MAC)

Dans la section précédente, l'architecture du moniteur, l'interface de la communication entre le moniteur et le noyau OpenCL, le flot de conception ainsi que la plateforme FPGA utilisée ont été décrits. Dans la présente section, une étude de cas permettant l'implémentation, la validation du moniteur et les résultats de mesure sont présentées.

2.2.3.1 Architecture et implémentation du MAC

Un multiplicateur accumulateur appelé MAC est utilisé comme banc d'essai afin de valider l'implémentation du moniteur décrit à la Figure 2.2. Le MAC contient trois opérations : de stockage (*store*), de chargement (*load*) et de multiplication-accumulation. La Figure 2.5 illustre l'architecture du MAC : (a) sans moniteur et (b) avec 4 moniteurs intégrés M 1 à M 4. Les moniteurs M1 à M2 enregistrent les temps d'exécutions pour les deux opérations *Load*. Les temps fournis par M2 et M3 permettent de calculer la latence de l'opération de multiplication tandis que les moniteurs M3 et M4 sont utilisés pour calculer la latence de l'opération accumulation. Les codes des noyaux OpenCL des MAC sans et avec moniteurs sont présentés à la Figure 2.6. Le compilateur OpenCL pour FPGA compile les fichiers des noyaux OpenCL (. cl) et génère les fichiers des configurations des FPGA (.aocx) (Figure 2.4). L'implémentation du MAC consiste à développer deux parties : une partie matérielle qui décrit l'architecture du moniteur en Verilog et une partie logicielle écrite en C++ et exécutée par le processeur. Cette partie logicielle permet de lire les données extraites par le moniteur et faire le posttraitement des données brutes pour fournir les temps d'exécution des opérations désirés et les latences calculées. Les performances calculées dans cette étude de cas sont la latence de l'opération *load*, l'intervalle d'initiation *II* (*Initiation Interval*) de la boucle (Figure 2.6.(b)) et les décrochages (*stalls*).

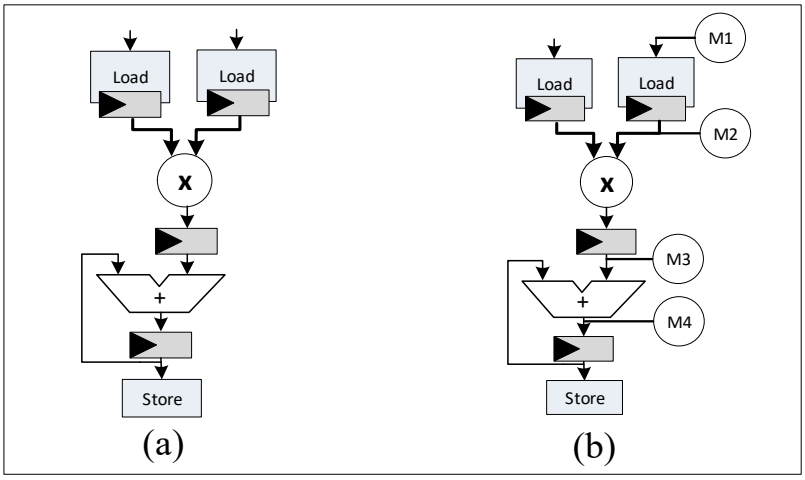


Figure 2.5 Intégration des moniteurs dans le noyau OpenCL pipeliné du multiplicateur-accumulateur MAC. (a) MAC sans moniteur (b) MAC avec 4 moniteurs

```
// kernel.cl:multiply-accumulate
// vector application example without
// monitor
__kernel void vecMAC (....)
{
int a, b, sum, mul;
for (int idx=0, idx<N, idx++) {
      a=x[idx];
      b=y[idx];
      mul= a*b;
      sum = sum+mul;
      z[idx]=sum;  }
```

(a)

```
// kernel.cl : multiply-accumulate vector
// application example with the insertion
// of four monitors
#include “monitor_lib.h”
__kernel void vecMAC (....)
#define [k]= [1024]; int a, b, sum,
mul;
unsigned long M1[k], M2[k], M3[k],
M4[k];
for (int idx=0, idx<N, idx++) {
     M1[idx] = monitor (idx, a);
     a=x[idx];
     M2[idx]= monitor (idx, a);
     b=y[idx];
     mul= a*b;
     M3[idx] = monitor (idx, mul);
     sum = sum+mul;
     M4[idx]= monitor (idx, sum);
     z[idx] = sum; }
```

(b)

Figure 2.6 Noyau OpenCL du circuit MAC. (a) Noyau sans moniteur. (b) avec moniteur

2.2.3.2 Résultats

Le MAC a été validé en implémentant deux modèles de noyaux : un noyau de type NDRange et un deuxième de type *single work-item* (les types des noyaux sont expliqués dans la

section 1.2.2). De plus, différents nombres de moniteurs ont été insérés pour étudier l'impact d'ajout de moniteurs dans les noyaux OpenCL. Quatre versions de noyaux ont été étudiées, avec 2, 4, 6 et 8 moniteurs. Tous les noyaux OpenCL ont été développés avec la même configuration : une seule SIMD (*single instruction multiple data*) et une seule unité de calcul (*compute unit*). Le posttraitement des données extraites par les moniteurs permet de déterminer différentes caractéristiques temporelles à la précision d'horloge du FPGA dont les temps d'exécution des opérations, la latence et le II des boucles, ainsi que les nombres de décrochages (*stalls*). Les opérations *load* et *store* peuvent causer des décrochages dans les pipelines lorsqu'ils accèdent à une mémoire globale. Le moniteur proposé permet de détecter quand ces décrochages se produisent avec la résolution précise de l'horloge du FPGA.

Les performances temporelles sont extraites pour les deux modèles : NDRange et *single work-item*. La Figure 2.7 présente le temps de chargement de l'entrée « a » où en général 20 cycles d'horloge sont mesurés entre chaque nouvelle entrée « a ». Deux *stalls* sont aussi observés de 74 cycles et 96 cycles lors de chargement des 100 premières entrées. De plus, les latences reportées des opérations « mul » et « sum » sont d'un seul cycle d'horloge lors de l'exécution du modèle *single work-item*. Par conséquent, le nombre de cycles requis pour générer la sortie « sum » est de 22 cycles d'horloge lorsqu'il n'y pas de décrochage. Cependant, les instruments reportent de mauvaises performances lors de l'exécution du modèle NDRange. En effet, le *II* reporté de la boucle est instable où 3000 cycles sont nécessaires pour le traitement des premières données, comme montre la Figure 2.8. Les performances de *II* sont améliorées à mesure que le flux des entrées progresse où le *II* tombe à 7 cycles d'horloge à la 63e entrée (Figure 2.8). Le *II* est la différence du temps mesuré par le même moniteur et pour deux éléments successifs (équation 2.1, page 57). Nous remarquons que le modèle *single work-item* est stable et plus performant où seulement 20 cycles nécessaires pour le chargement de données et 22 cycles requis pour générer la sortie « sum ». Par contre, 3000 cycles sont nécessaires seulement pour le chargement d'une nouvelle donnée « a » lors de l'exécution du modèle NDRange dans le FPGA. Il est important de mentionner que les performances du modèle NDRange se dégradent avec l'augmentation du nombre des données d'entrées traitées.

Finalement, le moniteur RTL proposé dans ce travail avec notre implémentation du moniteur décrit en OpenCL est comparé à celui précédemment publié (Verma et al., 2017). Ce dernier est décrit en OpenCL et basé sur un compteur décrit dans un noyau séparé. Un mécanisme de communication entre les noyaux appelé *pipes* est utilisé pour alimenter le noyau du MAC par les valeurs des compteurs généré par le noyau qui décrit le moniteur.

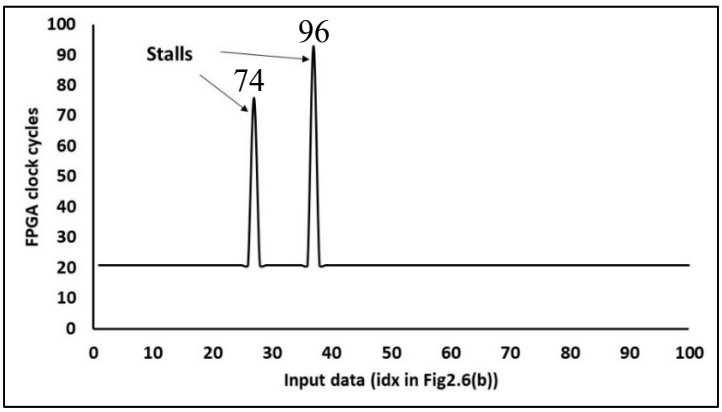


Figure 2.7 Courbe de performance temporelles de l'opération *load* intégré dans le noyau MAC de type single work-item décrit à la figure 2.6(b)

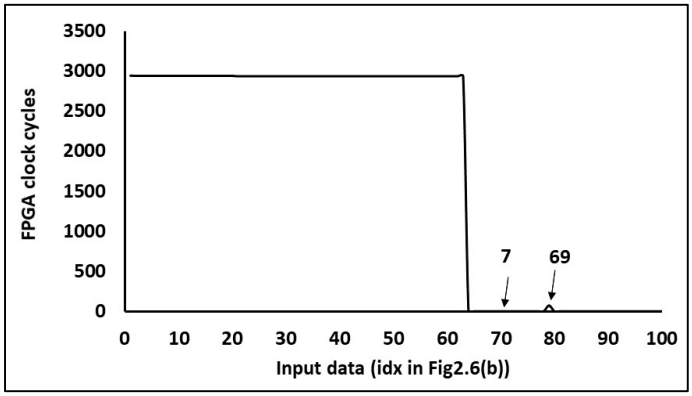


Figure 2.8 Courbe de II généré en exécutant le noyau NDRange du MAC décrit à la figure 2.6(b)

L'impact de l'insertion du moniteur proposé, développé en Verilog, sur le temps total d'exécution des noyaux est évalué. Le Tableau 2.1 montre les temps d'exécution de différents nombres d'entrées traitées allant de 1000 (1k) à 10 millions entrées dont chacune est de taille

32 bits. La colonne 6 montre que le temps d'exécution d'un noyau avec 10 millions d'entrées traitées avec 8 moniteurs augmente de 5% par rapport au noyau MAC sans moniteur (74.65 ms comparées à 74.25 ms). En général, ce résultat montre que le temps d'exécution du noyau augmente légèrement avec le nombre de moniteurs insérés lorsqu'une grande quantité de données sont traitées. Ce résultat montre que plusieurs moniteurs peuvent être insérés sans avoir un grand impact sur le temps d'exécution du noyau.

Tableau 2.1 Temps d'exécution du noyau *single work-item* (ms)

	Nombre de moniteurs par noyau				
Nb. de données d'entrée	0	2	4	6	8
1k	0.262	0.28	0.34	0.39	0.41
10k	1.61	2.01	2.11	2.22	2.29
100k	4.22	4.24	4.27	4.3	4.35
1M	7.54	7.6	7.61	7.62	7.69
10M	74.25	74.32	74.35	74.36	74.65

2.2.3.3 Comparaison avec un moniteur OpenCL

La consommation de ressources logique sur FPGA par les noyaux est résumée dans le Tableau 2.2. Ces résultats montrent que les taux d'utilisations des ressources logiques des noyaux intégrant des moniteurs développés en Verilog sont significativement plus petits qu'avec les moniteurs développés en OpenCL. En effet, le noyau d'un MAC avec 8 moniteurs développés en Verilog cause une surcharge de 31%, 6 % et 23% de FF, ALUT et d'éléments logiques, respectivement, par rapport à celui sans moniteur, ce qui est nettement inférieur à la surcharge de 194 %, 146 % et 135 % lorsque 8 moniteurs de type OpenCL sont instanciés dans le même MAC. Le résultat à retenir de ce tableau est que l'insertion des moniteurs développés au niveau RTL réduit significativement l'utilisation de ressources du FPGA par rapport à ceux développés en OpenCL. Cependant, les taux d'utilisation de FF et de RAM augmentent de 21% et 22% à 31% et 24 % respectivement, lorsque 8 moniteurs sont insérés. Le Tableau 2.3 compare les fonctionnalités de notre moniteur à ceux publiés par Verma et al (Verma et al., 2017) et Intel (Intel, 2020a).

Tableau 2.2 Ressources utilisées sur FPGA avec différents nombres de moniteurs intégrés (% est l'augmentation par rapport au noyau sans moniteurs)

	Nombre de Moniteurs par noyau	Flip-Flop	ALUT	Éléments logiques	RAM	FREQ (MHz)
	0	5253	3508	4280	42	333.3
Notre moniteur	2	21 %	8 %	10.5%	22 %	326.7
	4	24 %	4 %	16%	28 %	320.8
	6	27 %	12 %	20.6%	16 %	318.2
	8	31 %	6 %	23%	24 %	312.9
Moniteur OpenCL (Verma et al., 2017)	2	75 %	42 %	45%	18 %	316.7
	4	119 %	87 %	78%	34 %	294.9
	6	147 %	121 %	105.8%	44 %	282.7
	8	194 %	146 %	135%	77 %	271.5

Tableau 2.3 Comparaison des fonctionnalités des moniteurs

Fonctionnalités	Notre moniteur	(Verma et al., 2017)	(Intel, 2020a)
Détection des *stalls* de pipeline	Oui	Partielle	Non
Collecte de données tracées	Oui	Oui	Non
Calcul de II	Oui	Possible	Partielle
Mesure de la latence	Oui	Oui	Partielle
Possibilité de sélection des données à capturer	Oui	Possible	Non

2.2.3.4 Discussion

Dans cette première partie du chapitre, l'instrumentation de deux modèles de noyaux single work-item et NDRange ainsi que les deux types de moniteurs (le premier développé en Verilog et l'autre en OpenCL seulement) ont été évalués. Quatre différents noyaux, avec deux, quatre, six et huit moniteurs, ont été testés afin d'évaluer l'impact sur les performances des noyaux après l'insertion de moniteurs. Les résultats ont permis de montrer qu'un noyau de type single work-item fournit de meilleures performances temporelles par rapport à un noyau de type NDRange, en particulier au niveau de *II* des boucles, qui est égale à un seul cycle d'horloge (extrait par M4) alors qu'il est beaucoup plus grand pour le noyau NDRange. De plus, le moniteur développé en Verilog permet de mesurer les performances temporelles à la précision de l'horloge du FPGA. En outre, l'insertion de ce type de moniteur décrit au niveau RTL

permet facilement de gérer les ressources et les interfaces internes au FPGA, ainsi que de sélectionner les données à capturer et à les transmettre, ce qui n'est pas possible avec une description OpenCL des moniteurs. Par ailleurs, et tel qu'attendu, le moniteur développé au niveau RTL consomme moins de ressources logiques par rapport à celui développé seulement en OpenCL. La conclusion à retenir est que le moniteur développé en Verilog est avantageux à utiliser pour l'instrumentation des accélérateurs de type FPGA élaborés en OpenCL. Finalement, les résultats montrent que le moniteur proposé précis au cycle d'horloge du FPGA permet aux développeurs de diagnostiquer les problèmes qui peuvent dégrader les performances du noyau tels les décrochages et les mauvaises performances des boucles (*II*).

2.3 Plateforme d'instrumentation pour les accélérateurs de type FPGA élaborés en OpenCL

Cette partie décrit une plateforme d'instrumentation qui met à profit le moniteur décrit à la section 2.2. Cette plateforme fait l'objet de la deuxième contribution proposée par cette thèse qui est en cohérence avec l'objectif (1) présenté dans l'introduction de ce manuscrit à la page 6. La plateforme proposée est présentée selon les sous-sections intitulées :

- présentation de la vue d'ensemble d'un noyau avec instruments (section 2.3.1) ;
- illustration d'étapes d'instrumentation (section 2.3.2) ;
- modélisation des métriques de performance à extraire en fonction de type de la structure du noyau OpenCL implémentée (section 2.3.3) ;
- implémentation de la plateforme d'instrumentation qui est basée sur une architecture d'un instrument développé au niveau RTL, présentée dans la section 2.3.4 ;
- validation du fonctionnement de la plateforme avec les deux structures de noyaux ainsi qu'une validation en utilisant un ensemble d'applications (sections 2.3.5) ;
- évaluation et comparaison des résultats (section 2.3.6).

2.3.1 Vue d'ensemble d'un noyau avec instruments

L'instrumentation des noyaux consiste à insérer des instruments (nommés moniteurs à la section 2.2) en appelant la fonction d'instrumentation *instrument* ( ) au sein des noyaux. Le

développement de cette fonction est décrit dans la section 2.3.4.3. Un noyau OpenCL, peut contenir plusieurs unités de calcul (*compute unit* (CU)), appelées « *computation* (*Ci*) » dans la Figure 2.9. Cette figure montre la vue conceptuelle d'un noyau OpenCL avec six instruments insérés I1, I2, I3, I4, I5 et I6 et trois unités de calcul (C1, C2 et C3). Ces unités peuvent contenir un ensemble d'opérations arithmétiques, des opérations de chargement (*load*) ou de stockage (*store*). L'instrumentation des noyaux OpenCL permet d'extraire des informations d'exécution relatives à ces unités de calcul. Par exemple, l'unité C1 peut être instrumentée en utilisant l'instrument I1 ou les deux instruments I1 et I2. En effet, la différence entre les valeurs de synchronisation renvoyées par I1 et I2 peut déterminer la latence de C1 pour chaque entrée exécutée, égale au temps de traitement Tc1. La différence entre deux valeurs temporelles successives renvoyées par I2 permet de déterminer le *II* de C1. Les instruments I1 et I6 sont ainsi utilisés pour déterminer les temps de traitement de chaque unité de calcul (Tc1, Tc2, Tc3), les latences et le *II* du noyau au complet.

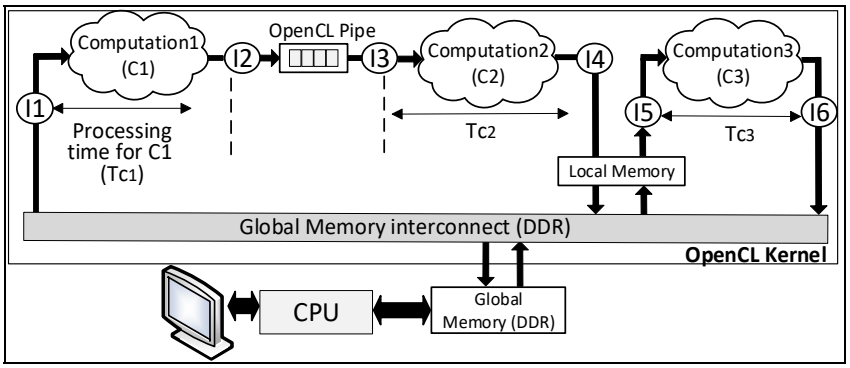


Figure 2.9 Vue d'ensemble d'un kernel avec instruments

2.3.2 Étapes d'instrumentation d'un noyau OpenCL

La méthode d'instrumentation proposée est basée sur une approche de conception conjointe OpenCL-Verilog, tel que détaillé à la section 2.2. Cette méthode de conception est dite conjointe OpenCL-Verilog car elle permet d'instrumenter des applications élaborées en OpenCL en insérant des instruments développés en Verilog dans les noyaux OpenCL. Les instruments sont développés avec Verilog puisque les expérimentations faites dans la section 2.2 ont montré que l'insertion des instruments développés en langage matériel est plus

avantageuse que ceux développés avec OpenCL. L'instrumentation des noyaux se fait en plusieurs étapes (Figure 2.10). À l'étape S1, les instruments sont insérés dans le code du noyau OpenCL. Le nouveau code du noyau est ensuite compilé et les FPGA sont configurés. À ce stade, le circuit résultant fonctionne sur le FPGA et à partir duquel les valeurs de variables OpenCL instrumentées avec les données temporelles en cycles d'horloge du FPGA sont enregistrées (S2). Ces données sont ensuite transférées vers le processeur (CPU) pour un post-traitement afin de générer les métriques de performances ciblées (S3). Finalement, le développeur analyse ces performances extraites, et peut alors décider d'optimiser certaines sections de code, d'ajouter ou supprimer certains instruments (S4), comme le montre la Figure 2.10.

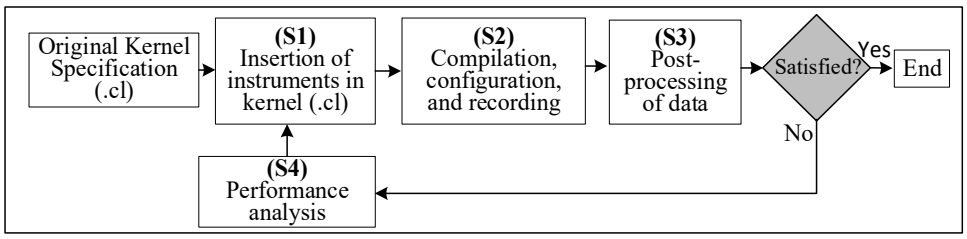


Figure 2.10 Étapes d'instrumentation d'un noyau OpenCL

2.3.3 Modélisation de performances des noyaux OpenCL

Une application OpenCL peut être conçue en utilisant une structure à noyau unique ou une structure à multiples noyaux. Les performances fournies par ces deux structures sur FPGA sont différentes pour une même application. La fréquence, le débit et les ressources logiques utilisées peuvent être très différents. Pour cette raison, les modèles de performance de deux structures sont présentés dans cette section. Dans ce qui suit, les modèles de performance d'une application OpenCL conçue avec un seul ou plusieurs noyaux OpenCL sont détaillés.

2.3.3.1 Modèle de performance d'un seul noyau

Un seul noyau OpenCL peut contenir de nombreuses unités de calcul (C), implémentées souvent dans un FPGA sous forme de pipeline. Ces unités sont instrumentées pour déterminer

la latence (*L*) et le *II* de diverses boucles ou pipelines, comme présentés à la Figure 2.11. Ces mesures sont exprimées en cycles d'horloge du FPGA à l'exception du temps d'exécution (*Tsk*), qui est exprimé en secondes. Les instruments insérés extraient le temps d'exécution de chaque élément traité (*work-item* (*wi*)) lors de l'exécution du noyau OpenCL sur le FPGA.

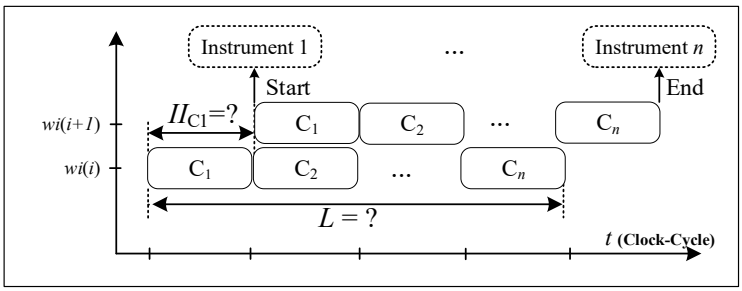


Figure 2.11 Métriques de performance d'un seul noyau

Le nombre d'instruments à insérer dépend des performances à extraire. Par exemple, un seul instrument est nécessaire pour extraire le *II* d'un pipeline ou d'une boucle ciblée et un minimum de deux instruments est nécessaire pour mesurer la latence du noyau. Ces deux métriques sont calculées selon les équations 2.1 et 2.2:

$$II_{ij} = t_j\,(wi\,(i{+}1)) - t_j\,(wi(i)) \tag{2.1}$$

$$L = t_n\,(wi(i)) - t_1\,(wi(i)) \tag{2.2}$$

où $t\,(wi\,(i))$ représente le temps mesuré de l'exécution pour l'élément $wi\,(i)$ et il est exprimé en cycles d'horloge du FPGA. Le II_{ij} est défini comme la différence entre deux temps d'exécution de deux éléments traités successivement par l'unité de calcul Cj (Figure 2.11). Le calcul de ces deux métriques de performance (*II* et *L*) relatif à chaque élément traité par le noyau OpenCL permet de détecter les décrochages de pipeline et d'identifier les causes de ces décrochages, ce qui facilite l'optimisation des temps d'exécutions. En effet, si le nombre de cycles reportés par les instruments relatifs au traitement de chaque entrée n'est pas stable, cela signifie qu'il a des déficiences ou décrochages dans le noyau implémenté. Par ailleurs, une implémentation idéale d'une unité de calcul ou opération quelconque possède un *II* d'un seul cycle d'horloge, ce qui signifie qu'un élément est traité à chaque cycle d'horloge par le noyau.

Alternativement, il est généralement difficile d'obtenir un II=1 surtout dans les opérations de calculs ou les boucles complexes ; il est donc crucial d'insérer des instruments capables de mesurer avec précision le II cible, et de détecter les opérations qui provoquent des déficiences ou des décrochages dans les pipelines.

Pour ce faire, nous proposons d'insérer des instruments entre chaque opération ou chaque unité de calcul pour permettre le calcul des *II*, de diagnostiquer les décrochages susceptibles de dégrader les performances ainsi que de tracer les chemins de ces décrochages à travers les pipelines implémentés dans le noyau OpenCL. Rappelons que le noyau OpenCL est implémenté dans le circuit FPGA sous forme d'un pipeline dont les entrées sont traitées par ce pipeline d'opérations. La Figure 2.11 montre le modèle d'un seul noyau OpenCL contenant *n* unités de calcul. Le nombre total de cycles d'horloge requis pour traiter toutes les entrées est défini par T_{sk} et il est donné par l'équation 2.3:

$$T_{sk} = L + II \times (N_{\mathrm{WI}} - 1) + T_{stalls} \tag{2.3}$$

où N_{WI} désigne le nombre d'entrées traitées par le noyau OpenCL. T_{stalls} désigne la somme des durées des décrochages exprimées en cycles d'horloge. Ce temps est difficile à prévoir théoriquement, car il dépend fortement de l'architecture matérielle dont les interfaces de communication, de la taille des entrées traitées ainsi que du type et de la technologie des mémoires utilisées dans le HPC. Le temps T_{stalls} permet donc d'être calculé par les instruments insérés selon l'équation 2.3.

La latence critique d'un noyau (L) désigne le nombre minimal de cycles d'horloge nécessaires pour produire une sortie lors de l'exécution du noyau, qui est égale à la latence maximale spécifique aux unités de calcul du même noyau (L_{ci}). Elle est définie par l'équation 2.4 :

$$L = \mathrm{Max}\,(L_{\mathrm{C1}}, L_{\mathrm{C2}}, \ldots, L_{\mathrm{Cn}}) \tag{2.4}$$

Le temps de traitement total (T_{pr}) requis pour traiter tous les éléments est exprimé en secondes et il est défini par l'équation 2.5, où F_{FPGA} désigne la fréquence d'horloge du noyau implémenté sur FPGA.

$$T_{pr}=\frac{T_{sk}}{F_{FPGA}} \tag{2.5}$$

2.3.3.2 Modèle de performance d'une structure multi-noyaux

Le principal défi lors de l'instrumentation des structures multi-noyaux est que des goulots d'étranglement (p. ex. les décrochages) peuvent se produire simultanément dans plusieurs noyaux. De plus, la dégradation des performances peut se produire en raison des canaux de communication entre les noyaux, appelés *pipes*. Par exemple, un mauvais dimensionnement des profondeurs de ces canaux de communication peut dégrader considérablement la synchronisation entre les noyaux et cause des décrochages. En effet, nous proposons d'insérer des instruments qui sont associés aux entrées et aux sorties de chaque noyau pour caractériser dynamiquement les structures multi-noyaux, tels que présentés par la Figure 2.12. Cela permettra de suivre les dégradations de performances des noyaux, principalement les décrochages. Le modèle présenté par la **Erreur ! Source du renvoi introuvable.** est donc proposé pour détecter les décrochages et suivre leur propagation à travers les différents noyaux. En effet, supposons qu'un nombre *n* d'instruments (*I* = [I1, I2, I3,..., In]) sont insérés dans les noyaux, et qu'un ensemble de *m* éléments (*work-item*) (*WI* = [wi_1, wi_2, wi_3,..., wi_m]) sont traités par ces noyaux.

Le temps d'exécution extrait par chaque instrument relatif à un élément donné *wi* peut être représenté par la matrice T_{nm} suivante :

$$T_{nm}=\begin{pmatrix} t_{11} & \cdots & t_{1m} \\ \vdots & \ddots & \vdots \\ t_{n1} & \cdots & t_{nm} \end{pmatrix} \tag{2.6}$$

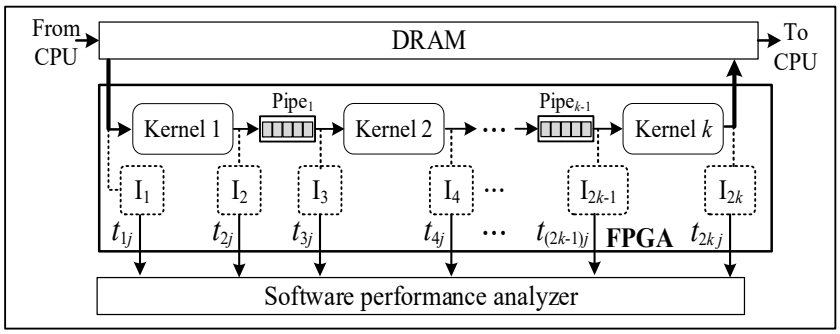


Figure 2.12 Modélisation d'une structure multi-noyaux

Une matrice $\Gamma_{n(m-1)}$ qui représente tous les *II* est calculée en utilisant l'équation 2.1 et elle est définie par l'équation 2.7:

$$\Gamma_{n(m-1)} = \begin{pmatrix} t_{12} & \cdots & t_{1m} \\ \vdots & \ddots & \vdots \\ t_{n2} & \cdots & t_{nm} \end{pmatrix} - T_{n(m-1)} = \begin{pmatrix} II_{11} & \cdots & II_{1(m-1)} \\ \vdots & \ddots & \vdots \\ II_{n1} & \cdots & II_{n(m-1)} \end{pmatrix} \quad (2.7)$$

Un noyau sans décrochage implique que $II_{nm} = II_{n(m+1)}$ pour toutes les valeurs de n et m. Un décrochage est alors détecté lorsque $II_{nm} \neq II_{n(m+1)}$, et le nombre de cycles de décrochage reporté par l'instrument numéro n lors du traitement des éléments m+1 et m est déterminé en calculant la différence $II_{n(m+1)} - II_{nm}$. Cette différence peut être appliquée à tous les éléments d'entrée et est exprimée sous la forme d'une matrice de différence nommée $\Delta_{n(m-2)}$ définie par l'équation 2.8:

$$\Delta_{n(m-2)} = \begin{pmatrix} II_{12} & \cdots & II_{1(m-1)} \\ \vdots & \ddots & \vdots \\ II_{n2} & \cdots & II_{n(m-1)} \end{pmatrix} - II_{n(m-2)} = \begin{pmatrix} \delta_{11} & \cdots & \delta_{1m} \\ \vdots & \ddots & \vdots \\ \delta_{n1} & \cdots & \delta_{nm} \end{pmatrix} \quad (2.8)$$

Un décrochage est détecté quand un élément de la matrice $\Delta_{n(m-2)}$ est non-nulle ($\delta_{ij} \neq 0$) et $\Delta_{n(m-2)}$ est une matrice nulle en l'absence de décrochage. Le nombre de décrochages peut être différent d'un noyau à l'autre, lorsque plusieurs noyaux produisent des décrochages simultanément. À titre d'exemple, basé sur la Figure 2.12, si les instruments I2, I3 et I4 produisent une matrice $\Delta_{n(m-2)}$ avec $\delta_{24} = \delta_{34} = 15$ et $\delta_{44} = 35$, cela signifie qu'il y a 15 cycles de

décrochages causés par *kernel* 1 et 20 cycles causés par *kernel* 2. Les noyaux et les entrées qui provoquent les décrochages sont ainsi reportés. Par conséquent, les développeurs peuvent se focaliser à optimiser les noyaux reportés, soient ceux qui causent ces décrochages, pour améliorer les temps d'exécution.

La matrice de latence, relative à une structure multi-noyaux (Figure 2.12), est représentée par l'équation 2.9. Cette matrice fournit la latence des unités de calcul en extrayant le nombre de cycles mesurés par deux instruments consécutifs pour la même entrée traitée.

$$L_{(n-1)m} = \begin{pmatrix} t_{21} & \cdots & t_{2m} \\ \vdots & \ddots & \vdots \\ t_{n1} & \cdots & t_{nm} \end{pmatrix} - \begin{pmatrix} t_{11} & \cdots & t_{1m} \\ \vdots & \ddots & \vdots \\ t_{(n-1)1} & \cdots & t_{(n-1)m} \end{pmatrix} = \begin{pmatrix} l_{11} & \cdots & l_{1m} \\ \vdots & \ddots & \vdots \\ l_{(n-1)1} & \cdots & l_{(n-1)m} \end{pmatrix} \qquad (2.9)$$

2.3.4 Implémentation de la plateforme d'instrumentation

Après avoir présenté les modèles pour la détermination des performances, cette section formule plutôt les étapes de développement et d'implémentation de la plateforme d'instrumentation. L'implémentation de la plateforme nécessite la création d'une bibliothèque d'instrumentation. D'abord, les fichiers sources requis pour le développement de l'instrument proposé ainsi que la bibliothèque sont détaillés (section 2.3.4). Cette bibliothèque décrit la partie matérielle de l'instrument. Ensuite, la partie logicielle de la plateforme est brièvement décrite (section 2.3.4.2). Rappelons que la plateforme présentée par la Figure 1.2 (page 12) est utilisée pour les implémentations.

2.3.4.1 Partie matérielle : Architecture RTL de l'instrument

L'instrument dérivé de la version décrite à la section 2.2 (Figure 2.2) est composé de trois modules et son architecture est présentée par la Figure 2.13. L'échantillonnage de données dans ces trois modules se fait à chaque nouvelle valeur d'index (p. ex. idx dans la Figure 2.6 (b)). En effet, le premier module *monitor* est un moniteur constitué d'un compteur binaire incrémenté à la cadence de l'horloge du FPGA et échantillonné à chaque nouvelle valeur d'index. Le deuxième module est appelé *untimed probe* est une sonde non chronométrée qui

enregistre la valeur de n'importe quelle variable à l'intérieur du noyau OpenCL sans aucune information temporelle. Le module *untimed probe* est utilisé pour le débogage des variables au sein des noyaux OpenCL. Le troisième module est une sonde chronométrée nommée *timed probe*, qui est une combinaison de deux premiers modules et fournit ainsi la valeur de la variable observée avec le temps de capture, exprimée à la précision du cycle d'horloge du FPGA. Seules les variables observées et associées avec leurs temps de capture sont échantillonnées et envoyées à la mémoire « *trace buffer* ». La variable observée est spécifiée par le développeur avec les autres paramètres lors de l'appel de la fonction d'instrumentation dans le noyau. Le module *timed probe* est utile lors de débogage des problèmes de synchronisation puisqu'on peut savoir le temps de traitement ou de transfert de chaque variable au cycle près. Chaque module de cet instrument peut être activé avec une entrée de sélection (*sel*) spécifiée par le développeur lors de l'appel de la fonction d'instrumentation, comme illustré à la Figure 2.13. La description Verilog de l'instrument contient également la logique responsable à la gestion de l'interface entre le noyau et l'instrument tel qu'expliqué dans la section 2.2.2.2. Le pseudo-code de la description Verilog est présenté par la Figure 2.14.

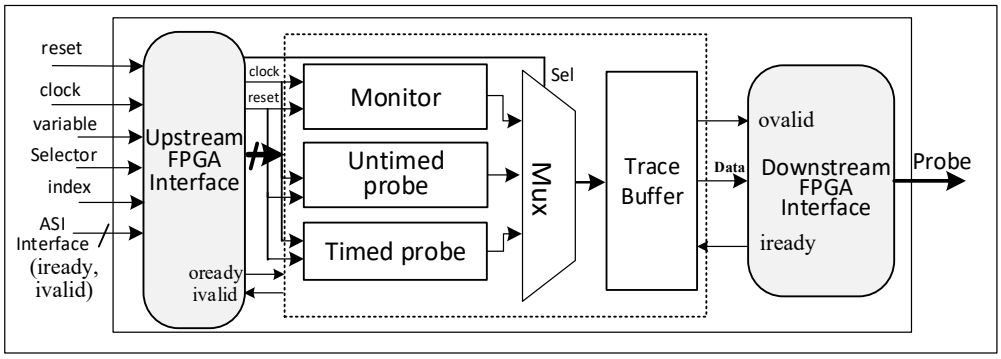


Figure 2.13 Architecture de l'instrument proposé

Chaque instrument inséré enregistre un ensemble de données, dans une collection de mémoires appelées *trace buffers*. Si la taille de ces données est relativement petite, il est possible et plus efficace de le stocker dans la mémoire interne à l'FPGA pour réduire la latence de la capture. Sinon, une mémoire externe peut être utilisée pour stocker les données si la taille est trop volumineuse. Une fois les données stockées, le CPU lit et traite toutes ces données pour fournir

les informations souhaitées aux développeurs (par exemple, la latence et les valeurs de variables extraites).

2.3.4.2 Partie logicielle

La partie logicielle de la plateforme contient plusieurs fonctions qui sont exécutées par le CPU, y compris les fonctions de la définition de la plateforme et des accélérateurs (les FPGA), les fonctions de lecture et d'écriture de/vers les mémoires. Ces fonctions permettent de transférer les données capturées par les instruments de/vers les mémoires globales. Ces mémoires sont accessibles au CPU, ce qui lui permet de faire le post-traitement des données extraites par les instruments. Ce post-traitement fait partie de la partie logicielle développée et il permet d'extraire la valeur des éléments capturés et les performances temporelles telles que la latence et II. D'autres fonctions sont aussi incluses dans la partie logicielle, sans s'y limiter, comme les fonctions de création de tampons de mémoires « *clCreateBuffer*», configuration des noyaux et exécution du noyau sur le FPGA.

2.3.4.3 Implémentation de la bibliothèque d'instrumentation

L'implémentation de l'instrument proposé consiste à créer une bibliothèque d'instrumentation composée de trois fichiers incluant le fichier Verilog et un fichier XML comme mentionné dans la sous-section 2.2.2. Ces fichiers sont présentés par la Figure 2.14. Le signal de sortie « probe » de l'instrument (la ligne 8 de la Figure 2.14) est de taille 64 bits, il renvoie les données échantillonnées définies aux lignes 31 à 33 selon le type d'instrument sélectionné par l'entrée de sélection « sel » (associé à *selector* à la Figure 2.13). Cette entrée de sélection est activée par le développeur au niveau OpenCL pour activer le type d'instrumentation désiré, comme décrit aux lignes 27 à 29 du code de la Figure 2.14. Les lignes 19 à 24 mettent en œuvre le processus de synchronisation des entrées de l'instrument avec l'interface interne du FPGA, comme l'exige l'interface Avalon-ST. Cette interface est implémentée automatiquement par le compilateur *Intel FPGA SDK for OpenCL*. Ses ports d'entrée et de sortie (par exemple, ivalid, oready) sont spécifiés dans les fichiers Verilog et XML, comme indiqué aux lignes 3, 7 et 43.

```
// Header .h file
1 unsigned long instrument (sel,index,var);
// Pseudocode of the Verilog file: instrument.v
2 module instrument
3  (input clock, resetn, ivalid, iready,
4 startofpacket, endofpacket,
5 input [31 : 0] enable, sel,
6 input [15 : 0] index, input_var,
7 output  oready, ovalid,
8 output  [63 : 0] probe);
9 assign probe=data;
10 assign i_inputs = {index,sel,input_var} ;
// local wires and signals declarations are omitted.
11 always @(posedge clock or negedge resetn)
12 begin
13   if( ~(resetn))
14     counter_time <= 32'h0;
15   else if (counter_time ==32'hFFFFFFFF)
16     counter_time <= 32'h0;
17   else
18     counter_time <=(counter_time + 32'h1);end
19 always @(posedge clock or negedge resetn)
20 begin
21   if( ~(resetn))
22     o_inputs <= WIDTH'h0;  //WIDTH =16 or 32
23   else if (ivalid)
24     o_inputs <= i_inputs; end
25 always @*
26 begin
27  monitor = {counter_time,o_index};
28  untimed_probe = {input_var,o_index};
29  timed_probe={counter_time,o_index,o_input_var};
30  case (o_sel)
31    32'h0 : data = monitor;
32    32'h1 :data = untimed_probe;
33    default : data = timed_probe;
34  endcase end
35 endmodule
//XML file for instrument engine
36 <EFI_SPEC>
37 name="instrument" module="instrument">
38 <ATTRIBUTES>
39   <IS_STALL_FREE value="yes"/>
40   <EXPECTED_LATENCY value="1"/>
41   <HAS_SIDE_EFFECTS value="no"/>
42 <INTERFACE>
43   <AVALON port="list of Avalon signals"/>
44   <INPUT port="list of input signals"/>
45   <OUTPUT port="list of output signals"/>
46 <C_MODEL>
47 <FILE name="instrument.cl"/>
48 <FILE name="instrument.v"/>
```

Figure 2.14 Le pseudocode des fichiers de description de l'instrumentation (fichiers Verilog, XML et .h)

2.3.5 Instrumentation et résultats

Cette section a pour objectif de valider la plateforme d'instrumentation avec un algorithme MAC codé en OpenCL selon une structure à un seul noyau et une structure multi-noyaux, décrit aux sections 2.3.5.1 et 2.3.5.2 respectivement. Enfin, l'impact des instruments insérés sur l'utilisation des ressources sur FPGA est évalué dans les deux cas instrumentés.

2.3.5.1 Instrumentation d'un seul noyau

Un algorithme MAC qui contient des opérations d'addition, de multiplication, de chargement (*load*) et de stockage (*store*) est décrit dans un seul noyau. Cet algorithme intègre aussi 4 instruments, comme illustré à la Figure 2.15. Les variables I1 à I4 représentent les *trace buffers* qui sont déclarées dans le noyau en tant que variables locales (lignes 11, 12, 14 et 16). Ces variables locales sont interprétées par le compilateur OpenCL comme des mémoires locales (BRAM). La taille de ces mémoires est principalement déterminée par le compilateur en fonction des données extraites par chaque fonction d'instrumentation « instrument ( ) », qui est égale à 64 bits dans ce cas. Puisque ces mémoires sont interprétées par le compilateur comme BRAM, leurs profondeurs sont aussi gérées par le compilateur. Ce choix dépend principalement de la technologie FPGA utilisée, dans laquelle les BRAM peuvent avoir différentes tailles. Chaque donnée échantillonnée est ensuite transférée vers la mémoire globale qui est une mémoire externe accessible par le CPU, comme indiqué à la ligne 19 de la Figure 2.15.

Deux modèles d'exécution sont instrumentés en utilisant l'algorithme présenté à la Figure 2.15: le modèle NDRange et single work-item (SWI). Rappelons que le modèle NDRange implémente explicitement le parallélisme de calcul au niveau des données en divisant la taille de données à traiter en groupes d'éléments. Le noyau est interprété comme un SWI lorsque le paramètre WLsize (ligne 3, Figure 2.15) est égal à 1 et comme un NDRange s'il est différent de 1. Les instruments utilisés sont implémentés en mode *timed probe* pour calculer les performances définies à la section 2.3.3.1.

```
1 //kernel.cl
2 #include "instrument.h"
3__attribute__((reqd_work_group_size(WLsize,1,1)))
__kernel
4  void single-kernel (
5  __global const int *restrict x,
6  __global const int *restrict x,
7  __global const int *restrict II1,
// Input & output declarations omitted
8    int N){
// Local variable declaration omitted
9  for(i=0; i<N; i++){
10   a = x[i];b = y[i]; // load operation
11   I1   = instrument (2,i,a);
12   I2   = instrument (2,i,b);
13   add  = a+b; //addition operation
14   I3   = instrument (2,i,add);
15   mul  = add*a; //multiplication operation
16   I4   = instrument (2,i,mul);
17   z[i] = mul; // store operation
// Uncomment for the NDRange model
18   //barrier( CLK_GLOBAL_MEM_FENCE);
19   II1[i] = I1;
// Remaining code omitted } }
```

Figure 2.15 Pseudo-code de l'algorithme présenté sous forme d'un seul noyau

Ces performances incluent la latence de la sortie « mul », par rapport à l'entrée de données, le *II* de la boucle ainsi que d'identifier les décrochages. En effet, les instruments I1 et I2 capturent les moments où les entrées "a" et "b" sont chargées tandis que I3 et I4 permettent de calculer la latence des opérations d'addition et de multiplication. De plus, le noyau présenté par la Figure 2.15 est testé avec des données d'entrées de différentes tailles afin de suivre son comportement lors de son exécution sur FPGA. La latence et le II, sont ensuite calculées selon les équations 2.1 et 2.2. Le *II* est mesuré par I4 sur 50 000 éléments (*work-items*) (une taille de 200 Kilo octets (Ko)), sa courbe est présentée par la Figure 2.16. La courbe de II montre qu'un élément est traité à chaque cycle d'horloge lors de l'exécution du modèle SWI. Cependant, cette courbe montre clairement huit décrochages dont la durée varie de 94 cycles d'horloge à 286 cycles d'horloge.

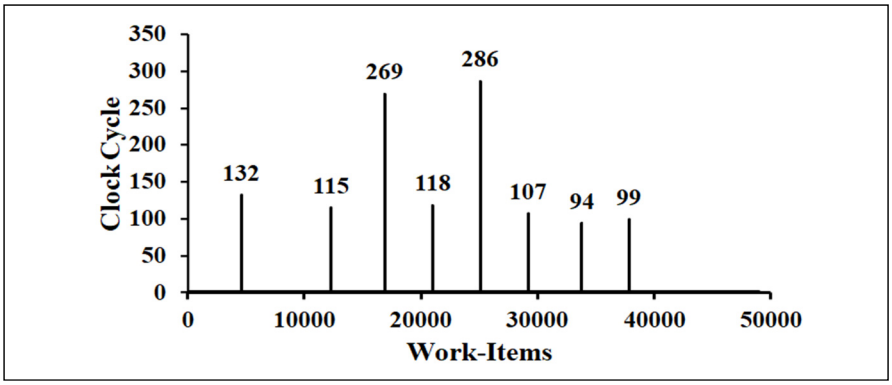


Figure 2.16 II mesuré par l'instrument I4 dans le modèle d'exécution SWI (WLsize=1, N=50K)

Le modèle NDRange est simulé en exécutant les 50 000 éléments (WLsize = 50 K). Le comportement enregistré lors de l'exécution de ce modèle est présenté par la Figure 2.17. Cette figure illustre la courbe de *II* avec et sans la fonction barrière (*barrier*) (ligne 18, Figure 2.15). La fonction barrière est utilisée dans le modèle NDRange pour la synchronisation de flux d'éléments à travers les opérations. L'exécution enregistrée du modèle NDRange avec barrière est présentée par la Figure 2.17 (b). Contrairement à celui du modèle SWI (Figure 2.17 (a)), l'exécution du modèle NDRange possède un débit instable de l'opération de multiplication, mesurée par l'instrument I4, comme il est illustré à la Figure 2.17 (a). La Figure 2.17 (a) présente seulement les II des 100 premiers éléments traités.

La courbe II du modèle NDRange commence avec une valeur de 33 cycles d'horloge et augmente à des milliers de cycles d'horloge en la présence de décrochages. Le II de ce modèle, mesuré par I4, est clairement beaucoup moins performant par rapport à celui du modèle SWI. Il est aussi important de mentionner que les instruments I1 et I2 ont enregistrés le même comportement, pour les opérations de chargement (*load*) et de stockage (*store*). Cette instabilité au niveau de II est expliquée par l'inefficience du transfert de données du CPU vers le circuit FPGA dans le modèle NDRange. Le II calculé en exécutant le modèle NDRange avec la fonction barrière (Figure 2.17 (b)) vaut 50000 cycles pour la plupart des 50000 éléments traités. De plus, la courbe montre également plusieurs états de décrochage avec des valeurs comprises entre 2 (II=50002) et 32 (II=50032) cycles lorsqu'une taille de 200 Ko de données est traitée (Figure 2.17 (b)).

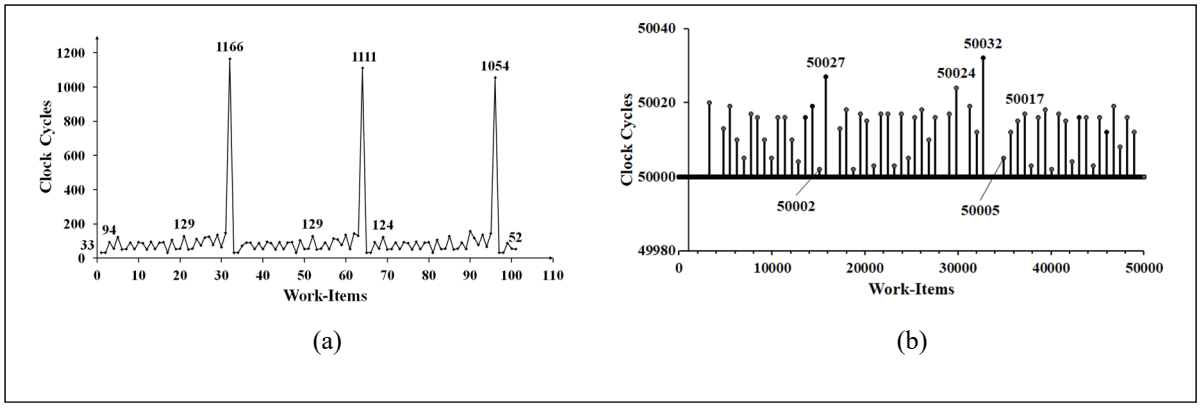


Figure 2.17 II des modèles d'exécution NDRange (WLsize=N=50k)
(a) sans barrière (les 100 éléments sont présentés),
(b) avec barrière (50 k éléments)

Les performances temporelles sont calculées en se basant sur les équations 2.3 et 2.4 pour les deux modèles SWI et NDRange sans barrières, et elles sont résumées au Tableau 2.4. Ces plateformes montrent que le modèle d'exécution SWI nécessite 21,2 fois moins de cycles que le modèle NDRange pour l'exécution de 50000 éléments. Cette mauvaise performance de NDRange par rapport au SWI est principalement due au grand nombre de décrochages observés dans l'implémentation de NDRange et qui contribuent jusqu'à 1975% du temps d'exécution optimal T_{sk}, qui devrait être 50000 cycles d'horloge pour les 50K éléments. Cependant, ces deux modèles maintiennent une latence minimale de 3 cycles d'horloge pour l'opération de multiplication, calculée comme la différence entre I3 et I4 pour la même entrée traitée. Ces résultats d'instrumentation aident non seulement à identifier le modèle d'exécution approprié, mais aussi à confirmer que le choix du modèle d'exécution pour une cible FPGA a un impact significatif sur les performances des applications élaborées en OpenCL. Ces résultats confirment que le choix du modèle OpenCL doit être pris en compte lors du développement d'une application ciblant un circuit FPGA.

Les ressources utilisées lors de l'implémentation de deux modèles d'exécution (Figure 2.15) sur FPGA sont résumées au Tableau 2.5. La consommation logique supplémentaire due à l'insertion des instruments est évaluée à partir des rapports des synthèses générés par les compilateurs. Les rapports de synthèse montrent que la consommation logique est similaire pour les modèles NDRange et SWI et ils montrent aussi que la surcharge de consommation

due à l'instanciation des instruments reste acceptable. En effet, la surcharge pour le modèle NDRange est de 0,72% pour *logic element* (LE), de 0,61% pour les *flip flop* (FF) et de 1% pour les *adaptive logic table* (ALUT). Ces taux de surcharge logique sont mesurés par rapport à l'implémentation sans instruments, ce qui confirme que la plateforme d'instrumentation a un faible impact sur la consommation des ressources sur FPGA. De plus, la fréquence de fonctionnement des implémentations sur FPGA diminue légèrement (de 287 MHz à 280 MHz) après l'insertion de quatre instruments dans le modèle d'exécution SWI.

Tableau 2.4 Performances temporelles exprimées en cycles d'horloge du FPGA et mesurées avec 50 k éléments (charge de 200 Ko) de deux modèles d'exécution

Modèle d'exécution	**T_{sk}**	**L_{min}**	**F_{FPGA} (MHz)**	**Stalls (%)**
SWI (Figure 2.16)	51260	3	287	2.1%
NDRange sans barrière (Figure 2.17 (a))	1087630	3	259	1975%
Ratio	21.2	1	1.08	914.4

Tableau 2.5 Ressources logiques utilisées par les noyaux avec 4 instruments

Nombre d'instruments	**Modèle d'exécution**	**FF**	**ALUT**	**LE**	**RAM**	**FREQ (MHz)**
0	SWI	77490 (5.17%)*	40389 (5.39%)	47972 (11%)	278	287
	NDRange	77827 (5.20%)	40690 (5.43%)	48223 (11.28%)	278	270
4	SWI	86791 (5.8%)	47957 (6.41%)	(53161) (12%)	312	280
	NDRange	86961 (5.81%)	48216 (6.44%)	53343 (12%)	312	258
Surcharge de 4 instruments	SWI	0.63%**	1.02%	1%	12%	-2.4%
	NDRange	0.61%	1.01%	0.72%	12%	-4.4%

*Ressources utilisées par rapport à celles disponibles dans le FPGA
**Ressources utilisées par le modèle d'exécution par rapport celles sans instrument

Tableau 2.6 Temps d'exécution (ms) des noyaux SWI (Figure 2.15)

	Nombre d'instruments par noyau				
Charge (*workload*)	0	2	4	6	8
5 Ko	0.24	0.26	0.29	0.33	0.34
40 Ko	1.26	1.41	1.59	1.72	1.81
100 Ko	3.52	3.59	3.63	3.69	3.72
4 Mo	7.61	7.64	7.68	7.71	7.72
40 Mo	75.48	75.49	75.50	75.51	75.52

En outre, des expérimentations sont aussi effectuées pour caractériser l'impact de l'insertion de différents nombres d'instruments sur les temps d'exécution total du noyau. Les résultats présentés au *Ressources utilisées par rapport à celles disponibles dans le FPGA
**Ressources utilisées par le modèle d'exécution par rapport celles sans instrument

Tableau 2.6 montrent que lorsque la taille des éléments présentés à l'entrée augmente, le temps d'exécution du noyau n'augmente que légèrement et atteint un maximum d'augmentation de 0,04 ms lorsque 40 Mo de données sont traitées. Étant donné que les applications de haute performance traitent généralement une grande quantité d'éléments (Vanderbauwhede & Benkrid, 2013), ce résultat confirme que l'impact des instruments sur le temps total d'exécution demeure très faible.

2.3.5.2 Instrumentation d'une structure multi-noyaux

L'algorithme MAC présenté sous forme d'un seul noyau à la Figure 2.15 est divisé en trois noyaux afin de former une structure multi-noyaux : le premier noyau pour l'opération de chargement des variables a et b ; le deuxième pour les opérations d'addition et de multiplication et le dernier noyau pour l'opération de stockage. Six instruments sont ajoutés pour caractériser la structure multi-noyaux. Le code des noyaux avec instruments est présenté à la Figure 2.18. Ces instruments sont insérés pour fournir les métriques de performance décrites à la section 2.3.3.2. Les performances temporelles telles que le II, la matrice de

différence $\Delta_{n(m\text{-}2)}$ sont extraites et illustré à la Figure 2.19. Les instruments permettent de détecter les décrochages soit à l'intérieur des noyaux, soit dans les canaux de communications entre les noyaux (*pipes*). En effet, des décrochages sont également observés dans le deuxième noyau de la structure multi-noyaux présentée à la Figure 2.18. Ce noyau contient les opérations de multiplication et d'addition dans lesquelles les décrochages peuvent provenir du premier noyau. Plus précisément, ces décrochages se produisent lorsque les canaux (Load_A, Load_B) sont vides ou lorsque le canal Mul_OUT est plein. Dans un tel cas, une nouvelle profondeur du canal serait nécessaire pour améliorer les performances du modèle.

```
1__Kernel void load_Data (
2 //Pipes and Global input/output declaration)
3 {
4 // Local variable declaration
5   for(i=0; i<N; i++){
6     a = x[i];
7     b = y[i];
8     I1[i] = instrument (2,i,a);
9     I2[i] = instrument (2,i,b);
10     write_pipe( Load_A, &a );
11     write_pipe( Load_B, &b ); } }
12__Kernel void inFPGA operations (pipes) {
13   for(i=0; i<N; i++)   {
14      read_pipe(Load_A,&Ain);
15      read_pipe(Load_B,&Bin);
16      I3[i] = instrument (2,i,Ain);
17      I4[i] = instrument (2,i,Bin);
18      Sum   = Ain+Bin;
19      Mul   = Bin*Sum;
20      I5[i] = instrument (2,i,Mul);
21      write_pipe( Mul_OUT, &Mul );   }  }
22__Kernel void store_data (pipes, outputs ) {
23 for(int i=0; i<N; i++){
24  read_pipe(Mul_OUT, &last_val);
25  I6[i] = instrument (2,i,last_val);
26  z[i]  = last_val;}     }
```

Figure 2.18 Structure multi-noyaux avec six instruments insérés

La Figure 2.19 montre les valeurs des matrices Γ_{68} and Δ_{67} mesurées selon les équations 2.7 et 2.8. Le *II* mesuré par l'instrument I5 est égale à un cycle d'horloge. Ce *II* est relatif aux opérations d'addition et de multiplication et démontre que le fonctionnement de ces opérations est optimal et qu'aucun décrochage n'est détecté. La latence de ces mêmes opérations est aussi

égale à 1 cycle d'horloge, ce qui présente une performance maximale du pipeline. Cette implémentation montre un *II* stable d'un seul cycle comparé à la courbe de *II* de l'implémentation à un seul noyau de type NDRange (qui est une courbe instable) (Figure 2.17). Les performances de cette structure sont similaires à celui du noyau SWI, mais avec une fréquence plus élevée (352 MHz, une augmentation de 13%). Il est également important de mentionner que la plateforme d'instrumentation occupe une faible surface logique. Le Tableau 2.7 résume le taux d'utilisation des ressources logique dans le FPGA : 1,29% de ALUT, 1,34% de LE et 0,73% de FF. Le tableau indique également que la plateforme d'instrumentation cause une réduction de 4,54% de la fréquence d'opération par rapport à une implémentation sans instruments.

$$\Gamma_{68} = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 2 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 2 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \qquad \Delta_{67} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Figure 2.19 Matrices de II (Γ_{68}) et la matrice de différence Δ_{67}, calculés selon les équations (2.8) and (2.9) respectivement, pour l'implémentation multi-noyaux de l'algorithme MAC

Tableau 2.7 Surcharge de ressources utilisées sur FPGA de la structure multi-noyaux avec 6 instruments insérés (Figure 2.18)

Nombre d'instruments /noyau	**FF**	**ALUT**	**LE**	**FREQ (MHz)**
0	80352 (5.37%)	43162 (5.77%)	49604 (11.61%)	352.6
6	91329 (6.1%)	52789 (7.06%)	55322 (12.95%)	336.58
Surcharge de 6 instruments	0.73%	1.29%	1.34%	-4.54 %

2.3.6 Évaluation de la plateforme d'instrumentation

Après la caractérisation de deux structures OpenCL, des applications de différents domaines sont utilisées pour étudier l'évolutivité et l'applicabilité de la plateforme d'instrumentation proposée. Ces applications font partie de la suite d'applications CHO (Ndu, Navaridas, & Luján, 2015), des applications proposées par Intel (Intel, 2019) et des bancs d'essai appelés Rodinia (Che et al., 2009). Ils incluent, sans se limiter, des applications arithmétiques telles que « dfadd », « dfmul », « dfsin » et d'autres algorithmes utilisés dans la cryptographie telles que « sha » et « blowfish ». Le Tableau 2.8 résume la consommation logique sur FPGA de ces applications sans instrument inséré. Un nombre différent d'instruments (10, 20 et 50 instruments) sont ainsi insérés dans chacune de ces applications, afin d'étudier l'évolutivité de la plateforme. L'augmentation de la consommation de ressources logique par instrument sont illustrés aux Figure 2.20, Figure 2.21 et Figure 2.22 où les taux de consommation maximale en termes de ALUT, FF et LE par instrument sont respectivement de 0,25%, 0,22% et 0,31%. Ces résultats montrent que les taux de consommation sont inversement proportionnels aux nombres d'instruments insérés. C'est-à-dire que les taux d'occupation sont plus grands par instrument lorsque moins d'instruments sont insérés. Cependant, les taux moyens convergent vers un pourcentage relativement stable lorsqu'un nombre important d'instruments sont insérés.

Tableau 2.8 Quantité de ressources logiques utilisées sur FPGA pour les 17 benchmarks sans insertion d'instruments

	Benchmark	ALUT	FF	LE	RAM	Freq (MHz)
1	mips	43441	82624	50449	356	252.3
2	kmeans	46492	84110	52123	525	270.2
3	gsm	52816	88114	54900	657	115.8
4	adpcm	61975	105273	63742	373	225.9
5	hotspot3d	67797	104203	69722	657	205
6	fd3d	79372	125814	74626	335	206.5
7	fft1D	93141	161411	88694	322	264.1
8	dfmul	95739	124516	83404	551	232.8
9	sha	96700	126838	88218	1163	168.4
10	blowfish	109858	172347	92815	826	135.4
11	cfd	120119	194048	108747	392	200.8

12	jpeg	142155	222400	118994	1300	171.1
13	srad	143616	223210	122300	1063	220.3
14	dfsin	176717	286011	151017	1286	126.4
15	lud	194698	292677	160469	673	218.8
16	dfadd	198038	215882	166064	745	220.7
17	Quicksort	257080	336544	211812	1312	180.4

Notez que l'insertion d'instruments peut également entraîner dans certains cas une diminution de la quantité de ressources logiques (FF ou ALUT). Ce cas est présent dans l'application 9 (-0,01% pour FF) comme montre la Figure 2.21. Cela est parfois possible, car l'insertion d'instruments peut forcer le compilateur de trouver de nouveaux chemins d'optimisation, à partir desquels le compilateur arrive avec une meilleure implémentation. Cependant, cette diminution du nombre de FF correspond aussi à une augmentation du nombre d'ALUT (0,06% pour ALUT, comme le montre la Figure 2.20). L'insertion d'instruments augmente également le nombre de mémoires utilisées sur FPGA « BRAM », les BRAM sont nécessaires pour l'enregistrement des données capturées. Le nombre maximal reporté de mémoire utilisé dans le Arria 10 est de 114 BRAM lorsque 50 instruments sont insérés ; présentant 4,2% du nombre de BRAM disponible sur le FPGA Arria 10. En fait, les résultats expérimentaux de l'instrumentation de 17 applications montrent que le nombre moyen de BRAM utilisés par instrument est de 1.5, 2.35, 2.45 et 2.22 lorsque 2, 10, 20 et 50 instruments sont insérés, respectivement. Il est important de mentionner aussi que les instruments insérés ne consomment pas de blocs DSP. Finalement, nous pouvons conclure que la consommation logique est relativement faible et que la plateforme d'instrumentation peut être appliquée à une grande variété d'applications.

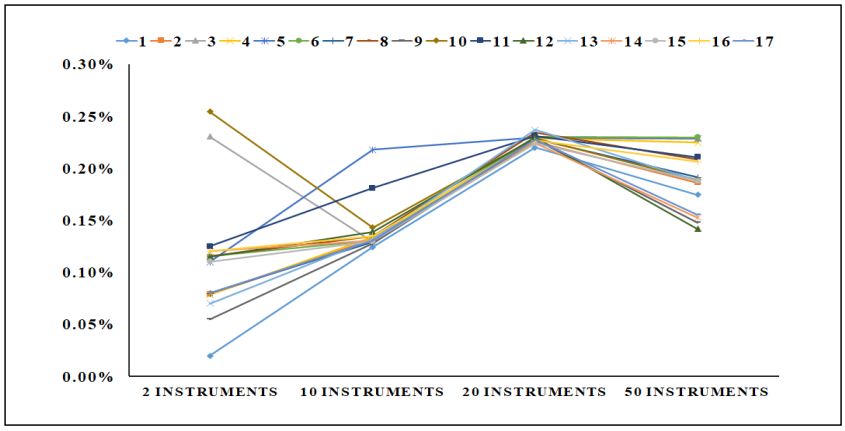


Figure 2.20 Taux d'occupation de ALUT sur FPGA pour les 17 applications

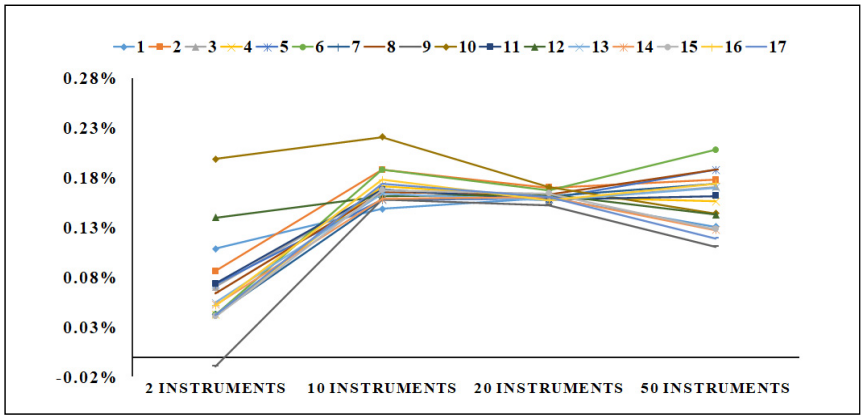


Figure 2.21 Taux d'occupation de FF sur FPGA pour les 17 applications

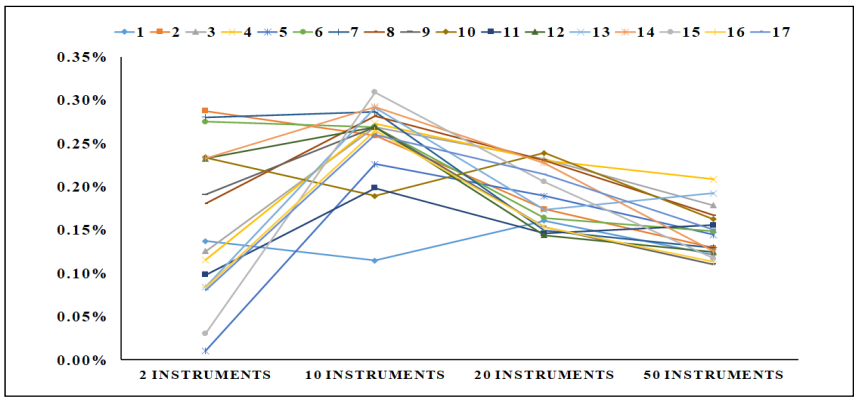


Figure 2.22 Taux d'occupation de LE sur FPGA pour les 17 applications

La fréquence d'opération des noyaux diminue avec l'augmentation du nombre d'instruments insérés. Cependant, la fréquence peut augmenter dans certaines applications lorsque les instruments sont insérés. Par exemple, la fréquence du MIPS (1ere application du Tableau 2.8) augmente de 2,39% lorsque 2 instruments sont insérés, comme le montre la Figure 2.23. Deux raisons expliquent cette amélioration. Tout d'abord, l'ajout de circuits d'instrumentation peut pousser le compilateur à trouver de meilleures solutions de placement et routage sur FPGA. Deuxièmement, le circuit avec des instruments intégrés peut être mieux structuré au niveau RTL que le circuit sans instruments. Cela peut conduire à des implémentations plus efficaces dans le FPGA. La Figure 2.23 résume les impacts de la plateforme d'instrumentation sur les fréquences d'opération de 17 applications. La perte maximale de fréquence reportée est de 4,5% (application 5). Ce résultat démontre que la plateforme d'instrumentation affecte légèrement les performances des noyaux OpenCL lors de l'instrumentation.

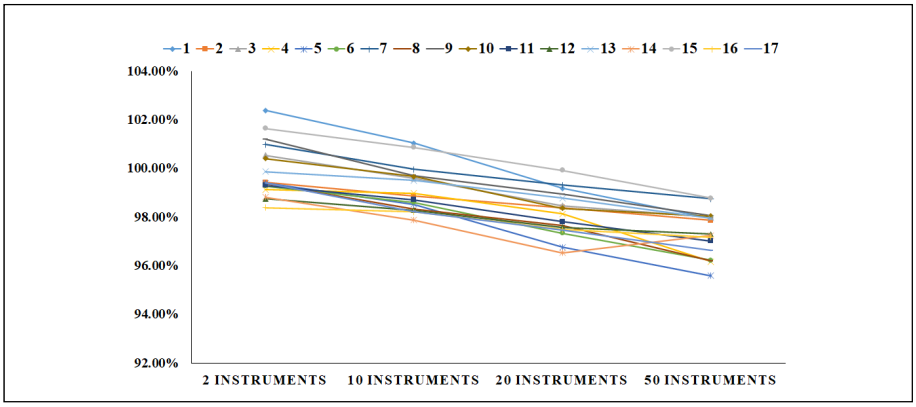


Figure 2.23 Variation de la fréquence de fonctionnement
pour les 17 applications

2.3.7 Discussion

Dans ce chapitre, nous avons proposé une plateforme d'instrumentation qui offre un support aux développeurs pour améliorer la visibilité et l'observabilité dans les applications élaborées en OpenCL mise en œuvre sur des plateformes hétérogènes reconfigurables. En ajoutant des instruments intégrés aux noyaux OpenCL, la plateforme offre aux développeurs la possibilité d'analyser précisément les performances temporelles des applications OpenCL compilées pour une cible FPGA. Cela fait de notre travail le premier à considérer les restrictions

d'instrumentation des noyaux OpenCL conçus pour les FPGAs. Notre solution proposée prend également en compte les défis de débogage dans l'outil *Intel FPGA SDK for OpenCL* tout en surmontant l'inefficacité des analyseurs logiques intégrés pris en charge par les outils HLS. L'instrumentation des différentes structures des noyaux confirme que les performances dépendent fortement du modèle d'exécution choisi. Enfin, les résultats montrent également que les taux de la surcharge maximale de consommation logique en termes de ALUT, de FF et de LE ne dépassent pas 0,25 %, 0,22% et 0,31% par instrument. D'autres résultats de comparaison sont donnés dans la section 4.2 (chapitre 4). Ces résultats caractérisent les performances et les fonctionnalités offertes par cette plateforme d'instrumentation afin de les comparer aux travaux publiés dans la littérature.

CHAPITRE 3

OPTIMISATION D'APPLICATIONS ÉLABORÉES EN OPENCL MISE EN ŒUVRE SUR DES PLATEFORMES HÉTÉROGÈNES RECONFIGURABLES

Ce chapitre présente les troisième et quatrième contribution de la thèse. Il vise à évaluer l'impact des techniques d'optimisation appliquées aux algorithmes SHA-2 et SHA-3 (introduits à la section 1.4), qui sont élaborées en OpenCL. Une exploration de l'espace de conception nous a permis d'identifier les techniques d'optimisation qui peuvent être appliquées à ces algorithmes, décrites dans la section 3.1. Nous détaillons ainsi les différentes implémentations OpenCL proposées et présentons les résultats expérimentaux pour les algorithmes SHA-2 (SHA-256) et SHA-3 aux sections 3.2 et 3.3 respectivement. Ce chapitre se base sur les concepts OpenCL introduits dans la section 1.2.2 du chapitre 1.

3.1 Les techniques d'optimisation des noyaux OpenCL pour une cible FPGA

Dans cette section, une série de techniques d'optimisation et de transformation du code sont introduites pour évaluer l'impact de ces techniques sur les applications élaborées en OpenCL et implémentées sur FPGA. D'abord, la motivation derrière l'étude et l'application des techniques d'optimisation et de transformation du code à haut niveau sont présentées. Ensuite, différentes techniques sont détaillées avec leurs directives spécifiques. Précisons que toutes les techniques d'optimisation présentées sont implémentées à l'aide de l'outil *Intel FPGA SDK for OpenCL*.

3.1.1 Motivation

Malgré les avantages qu'offre le langage OpenCL au niveau de la programmabilité, il n'est pas évident d'obtenir des implémentations efficaces sur FPGA sans l'application de techniques d'optimisation (Sanaullah, Patel, & Herbordt, 2018). Les applications élaborées en OpenCL offrent souvent des performances moins bonnes que les spécifications décrites en HDL ou même des spécifications de haut niveau fonctionnant sur des GPU ou des CPU. Par conséquent,

la compréhension de l'impact des transformations du code sur les performances est primordiale pour obtenir des implémentations de hautes performances. Ces transformations du code sont étudiées depuis des décennies pour l'implémentation des applications purement logicielles (Chellappa, Franchetti, & Püschel, 2008). D'autres travaux publiés ont étudié ces transformations pour les implémentations sur GPU (Lim, Norris, & Malony, 2017) (Muralidharan, Roy, Hall, Garland, & Rai, 2016) (Grauer-Gray, Xu, Searles, Ayalasomayajula, & Cavazos, 2012). Avec l'émergence des circuits FPGA dans les plateformes hétérogènes reconfigurables, une méthodologie d'optimisation utilisant les transformations de code est nécessaire pour un déploiement efficace sur FPGA.

3.1.2 Modèles d'exécution des noyaux OpenCL

Pour la compréhension de ce qui suit, rappelons qu'OpenCL offre deux modèles de noyaux, NDRange et *Single work-item* (SWI), tel que présenté dans la section 1.2.2.2. Dans le modèle NDRange, la charge de travail (*work load*) est divisée en des groupes d'éléments « work-group », ces groupes sont généralement assignés à des unités de calcul spécifiées dans les noyaux. Dans le modèle *Single work-item*, le noyau est implémenté en forme de pipeline où les entrées sont exécutées en série dans ce pipeline. Le compilateur génère un étage de pipeline pour chaque ensemble de boucles imbriquées dans le noyau, s'il y a lieu. Chacun de ces modèles est sujet à des optimisations spécifiques et la compréhension de l'interprétation des compilateurs OpenCL vers FPGA de ces modèles est primordiale pour pouvoir appliquer les optimisations adéquates à chaque modèle.

3.1.3 Optimisation par l'insertion des mémoires locales

Les latences produites par les accès aux mémoires globaux, qui sont généralement externes aux accélérateurs, sont beaucoup plus importantes que les latences de calcul. Par conséquent, les performances des noyaux sont fortement dépendantes de performances des liens de communications et de transactions entre les mémoires et les opérations au sein des FPGA. L'insertion d'une mémoire locale est une technique d'optimisation qui permet d'éviter ou de limiter les accès à la mémoire globale en exploitant les mémoires internes à l'FPGA. En effet,

les opérations ou les boucles à l'intérieur du noyau OpenCL font appel aux mémoires externes une seule fois pour le transfert des données vers la mémoire locale au lieu de faire des transactions pour chaque itération afin de recevoir une nouvelle entrée. En transférant temporairement les données de la mémoire globale (*global mem*ory), vers la mémoire locale (*local mem*ory), le nombre de transactions sera réduit significativement et cela peut améliorer considérablement la latence du noyau. Cette technique est appliquée à l'algorithme SHA-2 (section 3.2.3) et SHA-3 (section 3.3.1).

3.1.4 Optimisation des opérations de chargement et de stockage (load/store)

Les opérations de chargement et de stockage (*load*/*store*) peuvent limiter les performances lorsqu'il y a des dépendances de données en mémoires, surtout lorsqu'elles sont implémentées dans les boucles. En effet, le compilateur peut inférer de la logique supplémentaire qui cause une surcharge logique pour contrôler les fausses dépendances à la mémoire et cette logique supplémentaire peut dégrader la performance de la boucle. La directive *#pragma ivdep*, qui est compatible avec le compilateur *Intel FPGA SDK for OpenCL*, est utilisée pour empêcher le compilateur à supposer de fausses dépendances aux mémoires (Intel, 2020a). Cette directive pousse ainsi le compilateur à ne pas inférer la logique supplémentaire entre les opérations de chargement et de stockage existantes dans les boucles. La suppression de la logique supplémentaire peut réduire la latence et l'intervalle d'initiation des boucles. Cette directive ne s'applique qu'aux noyaux de type *Single work-item*. Cette technique est appliquée à l'algorithme SHA-3 (section 3.3.1).

3.1.5 Optimisations des boucles

Les optimisations sont appliquées aux boucles sous forme de transformations ou de directives spécifiques (*pragma*) insérées avant leur description. Le Tableau 3.1 résume les directives qui peuvent être appliquées aux boucles (Intel, 2020a).

Tableau 3.1 Directives permettant les transformations des boucles

Les directives	Transformations visées
#pragma unroll	Déroulement des boucles
#pragma loop_coalesce	Fusionnement de boucles imbriquées en une seule boucle
#pragma ii « n »	Imposition d'un II spécifique de '*n*' pour le pipelinage de la boucle
#pragma disable_loop_pipelining	Désactivation du pipeline au sein de la boucle
#pragma max_concurrency	Limitation de la concurrence des opérations dans la boucle. Le pragma *max-concurrency* s'applique au noyau du type SWI.
#pragma loop_fuse	Fusionnement de boucles adjacentes en une seule boucle
#pragma nofusion	Évitement de la fusion des boucles adjacentes

3.1.5.1 Pipelinage des boucles

Le pipelinage des boucles (*loop pipelining*) est une technique d'optimisation essentielle et efficace pour les implémentations sur FPGA. Le pipelinage des boucles améliore la latence (*L*) du noyau et l'intervalle d'initiation (*II*). Un pipeline optimal au niveau temporel offre une performance d'un seul cycle d'horloge pour chaque itération de la boucle et il peut être appliqué en insérant la directive « #pragma ii 1 » avant la boucle concernée, dans un noyau OpenCL. La Figure 3.1 illustre le concept du pipeline d'une boucle. Les deux métriques principales pour évaluer les performances d'une boucle sont la latence et l'intervalle d'initiation (*II*). Le *II* optimal doit être égale à un seul cycle tel que montrée dans la Figure 3.1. Cette technique est appliquée à l'algorithme SHA-2 et SHA-3.

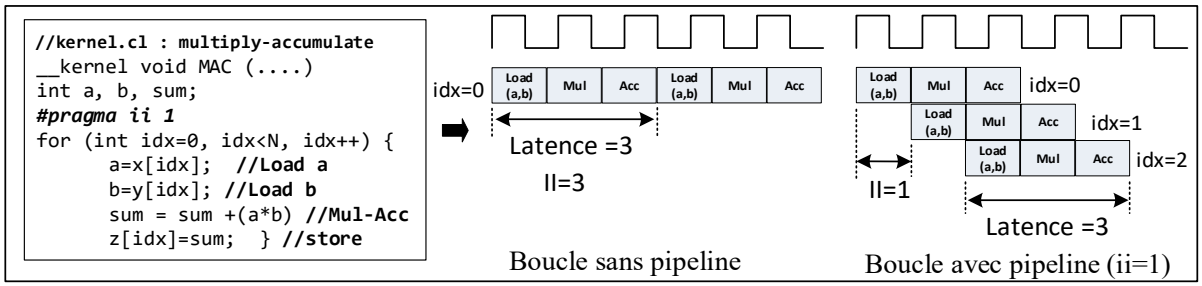


Figure 3.1 Concept de la technique *loop pipelining*

3.1.5.2 Fractionnement des boucles (*Loop splitting*)

Le pipeline optimal de la boucle (II=1) est très difficile à atteindre lorsqu'il y a des dépendances aux mémoires dans une boucle. Dans ce cas, les compilateurs OpenCL n'arrivent pas toujours à optimiser les performances du pipeline surtout pour une cible FPGA dont l'architecture est flexible et n'est pas fixe tels les GPU. Il est alors préférable d'utiliser le fractionnement de la boucle comme technique d'optimisation pour séparer les opérations qui ont des dépendances aux autres opérations du noyau. Les compilateurs OpenCL peuvent ainsi optimiser séparément les boucles dans le noyau et par conséquent maximiser leurs performances. La performance de cette technique varie selon l'algorithme et la complexité des opérations ainsi que le compilateur utilisé. Cette technique est appliquée à l'algorithme SHA-3.

3.1.5.3 Déroulement des boucles

Le déroulement des boucles est une optimisation permettant la parallélisation du traitement des données. Le déroulement des boucles se fait par l'insertion de la directive *pragma unroll* avant la boucle ciblée. Cette appellation « *pragma unroll »* est propre au compilateur d'Intel, par exemple le terme *pragma HLS unroll* est utilisé pour la technologie Xilinx. La directive *pragma unroll* transforme les boucles en créant plusieurs copies d'unités de calcul dans le circuit RTL généré, ce qui permet aux calculs de se produire en parallèle (Figure 3.2).

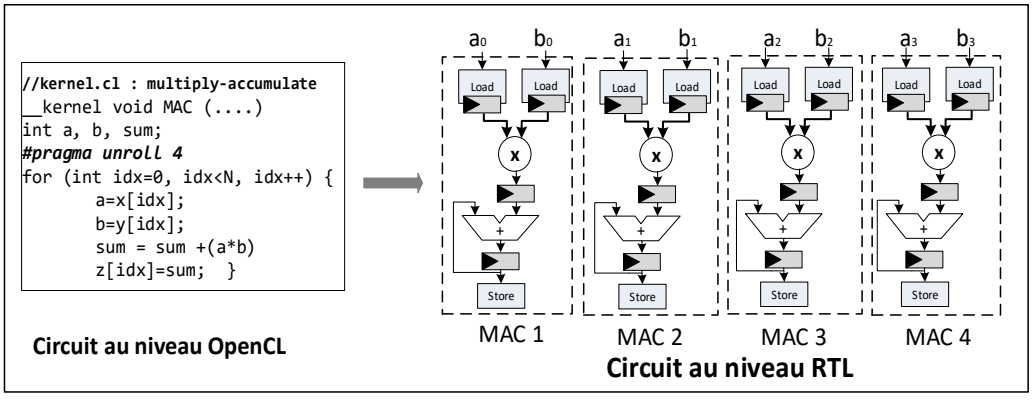


Figure 3.2 Concept du déroulement d'une boucle sur des architectures matérielles

Le déroulement peut être total ou partiel. Le déroulement partiel est préférable lorsque le déroulement total pourrait causer une surcharge de l'occupation logique dans le circuit FPGA. Par exemple, la Figure 3.2 montre le concept de déroulement d'une boucle qui contient une opération d'addition. Le circuit RTL généré se multiplie en suivant le facteur de déroulement, ce qui augmente le parallélisme des opérations. Cette technique est appliquée aux deux algorithmes SHA-2 et SHA-3.

3.1.5.4 L'optimisation des boucles imbriquées

Les boucles imbriquées existent souvent dans les codes de haut de niveau tel que C++ ou OpenCL. Ces boucles causent une surcharge logique supplémentaire sur les FPGA lors de la synthèse du code de haut niveau. Cette surcharge peut se multiplier avec la grandeur de la profondeur du nid de la boucle. Les compilateurs OpenCL pour FPGAs infèrent des registres et des blocs logiques supplémentaires tels que les Flip flops et les BRAM pour stocker l'état des différentes variables interne à la boucle imbriquée (Zohouri, 2018). La directive d'optimisation « #pragma loop_coalesce » est une directive spécifique au compilateur OpenCL pour FPGA de Intel. Cette directive permet au compilateur OpenCL de fusionner les boucles imbriquées en une seule boucle sans affecter la fonctionnalité. Cette réduction permet d'éviter la surcharge logique supplémentaire qui est nécessaire pour contrôler les boucles imbriquées. Néanmoins, cette directive a certaines limitations et le compilateur n'arrive pas toujours à fusionner les boucles surtout quand la condition de sortie de la boucle ne peut pas être déterminée.

3.1.6 Directives spécifiques « *attribute* »

Les directives spécifiques telles les *attribute* sont des optimisations propres à chaque compilateur et elles sont insérées dans les noyaux OpenCL pour limiter les options de synthèse. En effet, elles dirigent les compilateurs vers une synthèse plus spécifique et optimisée. Le Tableau 3.2 résume les directives essentielles au compilateur *Intel FPGA SDK for OpenCL* et explique leur effet sur la synthèse. Par exemple, la directive « attribute ((reqd_work_group _size(x,y,z)) » permet de déterminer le modèle d'exécution généré lors de la synthèse.

Tableau 3.2 Les directives spécifiques au compilateur de Intel pour OpenCL

Directives	Significations
__attribute((reqd_work_group_size(x,y,z))	Spécifier les charges du travail des noyaux, si (x,y,z) = (1, 1, 1), le noyau sera interprété lors de la synthèse comme un modèle SWI.
__attribute__((blocking))	Cette directive est utilisée pour les canaux de communication « pipes » bloquante ou non bloquante.
__attribute__max_work_group_size	Limiter le nombre de groupes. Les groupes modélisent la charge du travail du noyau.
__attribute__((num_simd_work_items(x)))	Spécifier le nombre de SIMD (Single instruction, multiple data).
__attribute__((num_compute_units(x)))	Spécifier le nombre d'unité de calcul.

3.2 Optimisation de l'algorithme SHA-2 (SHA-256)

3.2.1 Version de base du SHA-256

SHA-256 traite des messages d'entrée de 512 bits (M) et génère un message de sortie de 256 bits (*Hash*) (Theis & Wong, 2017), comme présenté à la section 1.4.1. Le pseudo-code du noyau OpenCL du SHA-256 (SHA-2) est présenté à la Figure 3.3. D'abord, un prétraitement de l'entrée est nécessaire (non présenté à la Figure 3.3) pour générer des blocs de message de 512 bits à partir des messages d'entrées de taille arbitraire en ajoutant un seul bit '1' suivit de plusieurs '0' jusqu'à ce que sa longueur atteigne un multiple de 512 bits. Ensuite, les calculs des fonctions qui constituent le SHA-256 débutent. En effet, huit variables (A_t à H_t) sont initialisées avec des valeurs fixes (lignes 1-2 de la Figure 3.3) et elles sont mises à jour à chaque itération de boucle (lignes 11-12). Deuxièmement, 64 itérations de hachage sont effectuées sur chaque bloc de message de 512 bits pour générer 64 mots (W) d'une taille de 32 bits chacun (lignes 4-7). Troisièmement, W et les quatre fonctions *Ch* (Choose), *Maj* (Majority), $\sum_0$, $\sum_1$, sont aussi utilisées pour calculer T1 and T2 (lines 9-10) selon les équations 1.3, 1.4, 1.5 et 1.6. Les valeurs de hachage intermédiaires sont enfin mises à jour en les additionnant avec les variables A à H (lignes 13-15). Le message de sortie est finalement généré et stocké dans la mémoire globale comme indiqué à la ligne 17.

```
Input : n 512-bit message blocks M = [M0,…, Mn]
Output : digest message : Hash
Hash initial values H[8]=[h0, h1,…, h7]
Initialization of K matrix: K[64]= [K0, … , K63]
Initialization of the working variables
1  At = h0;  Bt = h1;  Ct = h2;  Dt = h3;
2  Et = h4;  Ft = h5;  Gt = h6;  Ht = h7;
3  for each block in M do                          // Loop 1
4     for  (t = 0; t < 16; t++) do                 // Loop 1.1
5        Wt ← Mt
6        for  (t = 16; t < 64; t++) do             // Loop 1.2
7           Wt ← ∂0 Wt-15 + ∂1 Wt-2+ Wt-16 + Wt-7
8     for (t = 0; t < 64; t++) do                  // Loop 1.3
9        T1t = ∑1 (Et) + Ch (Et; Ft;Gt) + Kt +Wt + Ht
10       T2t = ∑0 (At) + Maj (At;Bt;Ct)
11       At= T1t + T2t;    Bt = At;   Ft= Et;     Dt= Ct;
12       Ct = Bt;          Et= Dt+T1t;            Gt = Ft;      Ht = Gt;
13    H[0] += At;  H[3] += Dt;   H[6] += Gt ;
14    H[1] += Bt;  H[4] += Et ;  H[7] += Ht ;
15    H[2] += Ct;  H[5] += Ft ;
16 for  (i = 0; t < 8; t++)   do                   // Loop 2
17    Hash (i) ← H(i)
```

Figure 3.3 Pseudo-code de la version de base du SHA-256

3.2.2 Métriques de performance

Pour évaluer les implémentations des algorithmes SHA-2 et SHA-3, les métriques suivantes sont typiquement prises en considération :

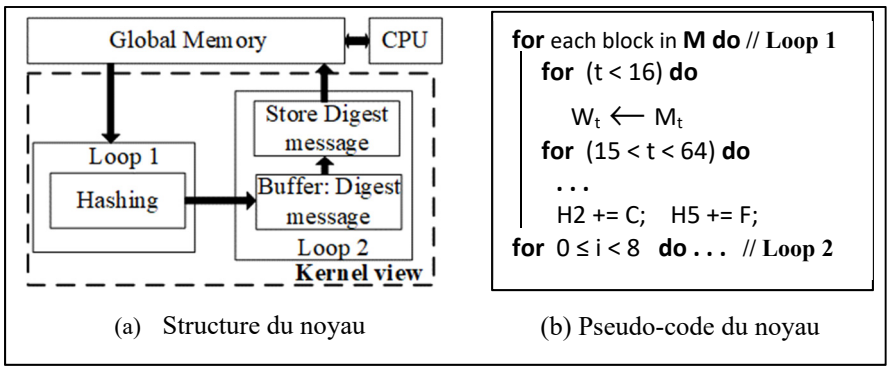
- **débit** : le nombre de bits traités par unité du temps (seconde) ;
- **accélération (speed up)** : le temps d'exécution de la version non optimisée divisé par le nouveau temps d'exécution (le temps de la version optimisée) ;
- **fréquence** : la fréquence d'opération du circuit sur FPGA qui est déterminée lors de la synthèse ;
- **nombre de ressources logiques** : le nombre des cellules logiques (p. ex. bascules, ALUT, RAM) utilisées par le circuit implémenté.

3.2.3 Les différentes implémentations des noyaux du SHA-256

Cette section décrit les différentes spécifications OpenCL créées pour évaluer les techniques d'optimisation décrites dans la section 3.1. Ces évaluations visent à déterminer si les techniques améliorent les performances de la version de base de l'algorithme SHA-256 (Figure 3.3Figure 3.4). Ces techniques comprennent l'insertion de mémoires locales, le fractionnement, le déroulement et le pipelinage des boucles. Les différents noyaux OpenCL créés sont résumés au Tableau 3.3. Ces différentes versions de noyau et leur implémentation en OpenCL sont décrites dans ce qui suit.

Version VB : Version de base

L'implémentation de base de l'algorithme SHA-256 est illustrée par la structure présentée par la Figure 3.4. Les blocs de messages d'entrées sont transmis par le CPU de la mémoire globale au FPGA et ils sont traités directement selon les opérations définies dans la boucle 1 (*loop 1*) (Figure 3.3 et Figure 3.4(b)). Le message de sortie est stocké dans une mémoire locale au FPGA avant d'être transféré vers la mémoire globale. Enfin, le CPU lit ce message de la mémoire globale pour le rendre visible aux développeurs pour une lecture ou un posttraitement.



(a) Structure du noyau (b) Pseudo-code du noyau

Figure 3.4 Implémentation du SHA-256 tel que décrite par Figure 3.3

Version V : Insertion d'une mémoire locale

Dans la version V, une mémoire locale est insérée « localmem » pour réduire le nombre de transactions lentes de la mémoire globale à chaque itération de calcul SHA-256 comme le

montre la Figure 3.5. Cette technique permet alors de réduire la latence (*L*) et d'améliorer le *II* (*Initiation interval*) d'une boucle en réduisant le nombre de décrochages dus aux accès à la mémoire globale externe.

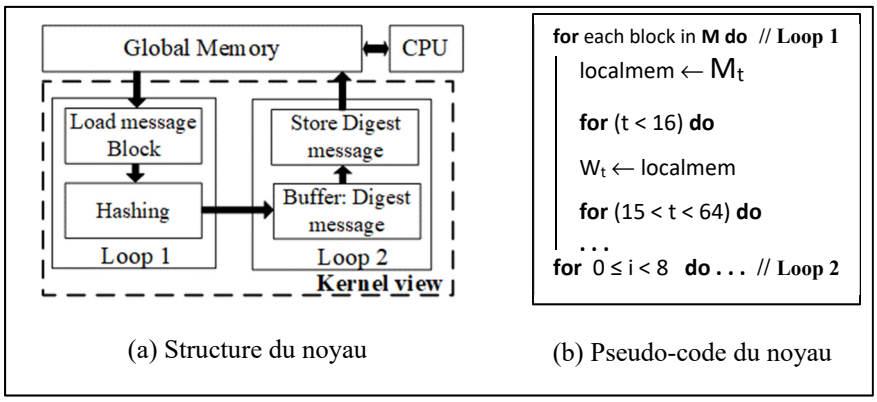


Figure 3.5 Noyau SHA-256 avec insertion d'une mémoire locale

Version VS : fractionnement des boucles

Dans cette version, la première boucle est divisée en deux pour séparer la dépendance de la mémoire globale et le calcul au sein du noyau. La Figure 3.6 présente la vue globale du noyau, dans laquelle le transfert de la mémoire globale vers la mémoire locale est séparé du calcul de hachage (les 64 itérations). Trois boucles sont ensuite créées à l'intérieur du noyau du SHA-256.

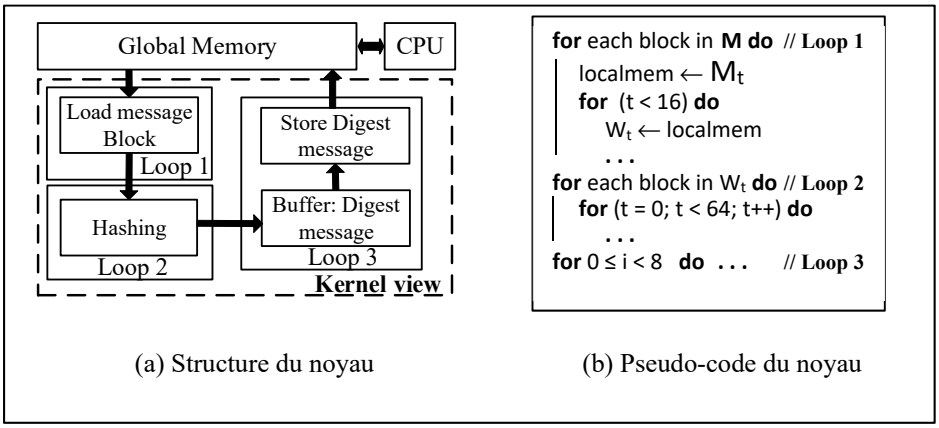


Figure 3.6 Structure du noyau avec fractionnement de la boucle

Version VP : boucle avec pipeline

Le mode pipeline est activé dans une boucle en insérant la directive # pragma ii 1 avant la boucle. Cette directive est compatible avec le compilateur *Intel FPGA SDK for OpenCL*. Cette directive est activée sur les boucles 1.1, 1.2, 1.3 et 2 du code à la Figure 3.3.

Version VU : déroulement de la boucle

Le déroulement est activé avec un degré spécifique et il est appliqué en ajoutant le « pragma unroll <p> » où p désigne le nombre d'itérations parallèles. Les boucles 1.1, 1.2 et 1.3 du code à la Figure 3.3 sont entièrement déroulées lors de l'application de la technique de déroulement.

3.2.4 Implémentations et résultats

Neuf noyaux de SHA-256 sont créés : la version de base (VB) et huit autres versions qui combinent les différentes techniques d'optimisation décrites dans la section 3.1, comme résume le Tableau 3.3. Les neuf noyaux SHA-256 ont été compilés et implémentés en modèle SWI. L'impact de chaque combinaison de techniques d'optimisation sur les performances des noyaux est évalué. Les débits et les fréquences maximales obtenues par les différentes implémentations sont présentés au Tableau 3.4. Le débit et la fréquence sont évalués en utilisant treize différentes tailles d'éléments aux entrées allant de 1 Ko à 2 Mo. Le débit est calculé en divisant la taille des éléments traités par le temps d'exécution du noyau.

Tableau 3.3 Les neuf versions du SHA-256 auxquels s'appliquent les techniques d'optimisation

Versions	***Local memory***	***Splitting* (S)**	***Unrolling* (U)**	***Pipelining* (P)**
VB				
V	✔			
VP	✔			✔
VU	✔		✔	
VUP	✔		✔	✔
VS	✔	✔		
VSP	✔	✔		✔
VSU	✔	✔	✔	
VSUP	✔	✔	✔	✔

Les résultats expérimentaux montrent que l'insertion d'une mémoire locale dans la version de base (version VB) améliore le débit d'un facteur de 6.7 par rapport au noyau VB sans mémoire local (1160 Mbps/173 Mbps). De plus, les techniques d'optimisation de la version VSUP (fractionnement de la boucle combinée avec pipeline et déroulement) améliorent le débit d'un facteur de 22,9 par rapport à la version de base VB (3973 Mbps comparé à 173 Mbps). Il est aussi important de mentionner que l'implémentation VSUP offre le plus haut débit mesuré parmi toutes les versions. Comme prévu, la fréquence d'horloge la plus élevée a été obtenue lorsque le fractionnement de boucle et le pipelining de boucle sont appliqués (VSP). Cela s'explique par l'insertion appropriée de registres à plusieurs étages dans les implémentations sur FPGA. L'amélioration des performances observées par l'application de ces techniques démontre l'efficacité des techniques d'optimisation proposées.

Tableau 3.4 Performances pour 2 Mo d'éléments

	Débit (T) ($T/T_{baseline}$) (Mbps)	**Fréquence du FPGA (MHz)**	**LE (%)**	**Surcharge logique (*logic overhead*) (%)**
VB	173 (1×)	237	13	N/A
V	1160 (6.7×)	238	13	0
VP	1165 (6.7×)	239	14	1
VU	3380 (19.5×)	170.8	16	3
VUP	3787 (21.9×)	170.8	16	3
VS	1533 (8.86×)	241.6	12	-1
VSP	1570 (9×)	241.2	12	-1
VSU	3360 (19.4×)	179.4	16	3
VSUP	3973 (22.9×)	179.4	16	3

Les taux d'accélération sont définis comme le rapport du temps d'exécution de la version de base (sans optimisation) par le temps d'exécution de la nouvelle implémentation avec les optimisations. Ces taux sont résumés à la Figure 3.7. Les implémentations VSUP et VSU sont les plus rapides, i.e. avec les plus petits temps d'exécution. L'implémentation VSUP offre une accélération de 90× par rapport à l'implémentation de base VB. Cette amélioration des performances est possible au prix d'une plus grande utilisation des ressources du FPGA.

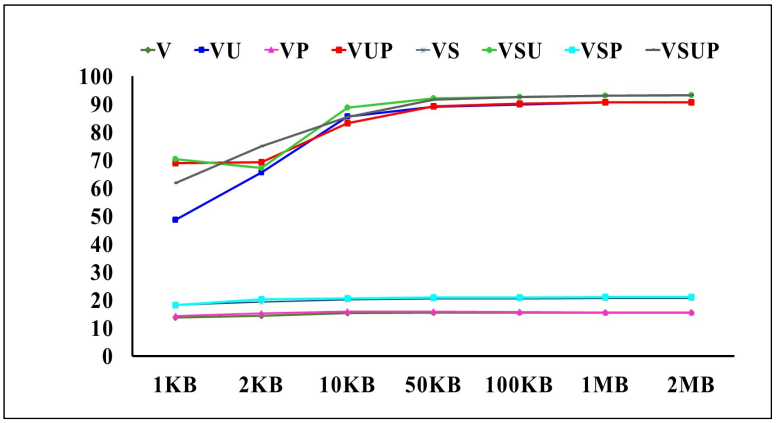


Figure 3.7 Taux d'accélération des implémentations optimisées par rapport à la version de base

L'implémentation VSUP consomme plus de ressources que les autres, en raison de déroulement de boucle appliquée au noyau. L'utilisation des ressources en termes de ALUT, de FF et de LE a été évaluée tel que rapportée à la Figure 3.8. Les noyaux qui consomment le plus de ressources sont ceux obtenus lors de l'application de l'optimisation du déroulement de boucle (VU, VUP, VSU et VSUP). Les résultats présentés à la Figure 3.8 montrent que le fractionnement de boucle diminue l'utilisation des ressources logiques, ce qui confirme que le compilateur OpenCL pour FPGA génère une surcharge logique lorsqu'une boucle imbriquée existe dans le noyau. En effet, l'insertion de mémoires locales internes diminue la surcharge logique nécessaire pour contrôler les dépendances des données dans les boucles et les conflits d'accès aux mémoires globales, mais elle augmente également le nombre requis de mémoires internes du FPGA (BRAM). Le nombre de BRAM supplémentaire dépend de la taille des mémoires locales déclarées dans le noyau.

Les résultats expérimentaux montrent qu'un noyau avec 10 Kilo-octets de mémoire locale réservée consomme 13 BRAM de plus dans le FPGA Arria 10 par rapport au noyau sans mémoire locale tel que celui de la version de base (VB). La surcharge maximale reportée (par rapport au noyau de base) est relativement faible et représente 3% des ressources disponibles sur Arria 10 (Tableau 3.4). Cela confirme que les techniques d'optimisation appliquées ont un faible impact sur la ressource logique. Finalement, l'implémentation qui offre les meilleures

performances est la version VSUP, qui cumule trois techniques d'optimisation : l'insertion de mémoire locale, boucle avec pipeline et le déroulement de boucle.

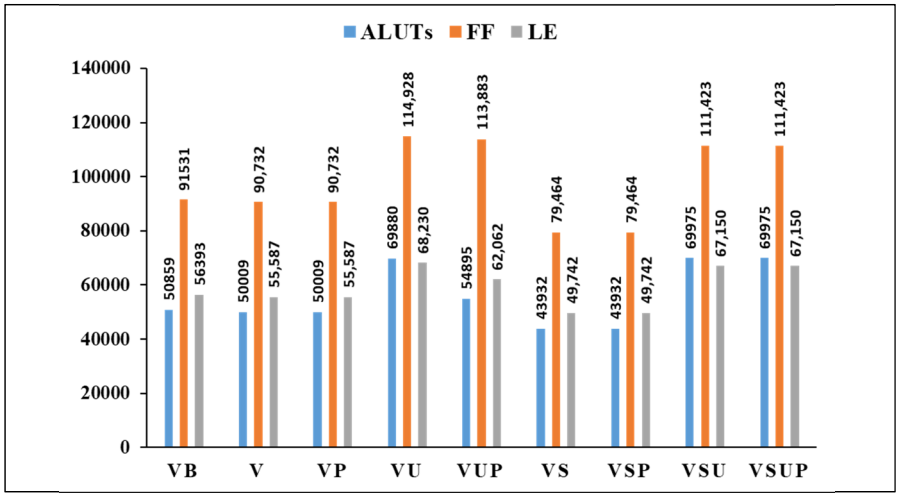


Figure 3.8 Nombre de ressources utilisées par les différentes implémentations du SHA-256

3.3 Optimisation de l'algorithme SHA-3

La structure typique de l'algorithme SHA-3 contient deux principaux modules : un module de prétraitement (*padding*) et la fonction de hachage principale du SHA-3 « Keccak », comme illustrée par la Figure 1.7 (section 1.4.2 du chapitre 1). Le *padding* prend comme entrées des messages de taille arbitraire pour produire des blocs de message de taille fixe M = [M1,..., Mt]. Le Tableau 1.1 au chapitre 1 (page 41) résume les spécifications de quatre algorithmes évalués dans ce travail de la famille SHA-3 classés suivant leur taille de sorties (SHA-224, SHA-256, SHA-384 et SHA-512).

3.3.1 Les différentes versions du noyau SHA-3

Le pseudocode de la description OpenCL du SHA-3 est présenté à la Figure 3.9 où la fonction Keccak est implémentée dans une seule boucle sous forme d'un seul noyau. Chaque itération de cette boucle reçoit un bloc de message (M1 à Mt) qui est d'abord chargé dans un tableau A (ligne 5). La taille de ces blocs de messages d'entrée est fixée en fonction de la fonction de

hachage ciblée. La boucle 1.2 (ligne 6) effectue les 24 itérations, dans lesquels les 5 étapes (Thêta, Rho, Pi, Chi et Iota) sont calculées (ligne 7 à 11). Chaque étape fonctionne sur les trois tableaux A, B et C, et se compose d'une boucle de 5 itérations. Enfin, la sortie du hachage est générée par la boucle 2 (ligne 13). Dans ce travail, l'exploration du pseudo-code est d'abord effectuée pour identifier les techniques d'optimisation qui peuvent être appliquées au coprocesseur SHA-3. Les différentes techniques d'optimisation appliquées sont basées sur ce pseudo-code. Ces techniques sont détaillées au Tableau 3.5 et sont basées sur l'environnement OpenCL pour FPGA de Intel. Sept noyaux sont ainsi créés, chacun de ces noyaux applique une combinaison différente de ces techniques.

Version VB

La version VB est l'implémentation de base de SHA-3. Cette version de base n'a pas bénéficié d'aucune technique d'optimisation et le pseudo-code est présenté à la Figure 3.9.

```
// Input : message blocks = M = [M1,…, Mt]
// Output : hashed message = H = [h1, …, hn]
// Block_size : 1152, 1088, 832 or 576
// Iota matrix initialization: RC = [RC0 ,…, RC23]
// Working variables initialization: A, B, C and D.

1  __kernel void sha3 (__global unsigned M,
                       __global unsigned long H,
                       unsigned Nb_of_Block)
2  {
3    for each block in M do       // Loop 1
4      for k = 1 to (block_size/8) do // Loop 1.1
5        A[k] ← load(M [k], block size)
6      for ROUND = 0 TO 23 do     // Loop 1.2
7        A ← theta (A,C,D)       // Loops 1.2.1/2/3
8        A ← rho (A)
9        B ← pi (A)
10       A ← chi (B)              // Loop 1.2.4
11       A ← iota (A,RC)
12   for i = 0 to 24 do           // Loop 2
13     H[i] ← A[i]
14 }
```

Figure 3.9 Pseudocode de l'implémentation de base de SHA-3

Version VU

La version VU est une implémentation qui déroule les boucles présentes dans le noyau. La boucle 1.2 est partiellement déroulée d'un facteur de 2 tandis que les boucles 1.2.1 à 1.2.4 sont entièrement déroulées.

Version VC

Cette version est une implémentation qui fusionne les boucles imbriquées qui ont accès aux mémoires externes en insérant l'option « #pragma loop_coalesce », à la boucle 1.

Version VI

L'implémentation VI applique l'option « ivdep » qui est une optimisation spécifique au compilateur Intel OpenCL utilisée dans un modèle de noyau *Single work-item* lorsqu'une dépendance de boucle à la mémoire globale limite les performances du noyau. Le pragma ivdep améliore les performances de la boucle tel que la latence en empêchant les opérations de chargement inutiles. Le noyau VI applique ivdep à la boucle 1.1 pour atténuer la dépendance d'accès à la mémoire. Ceci est particulièrement important lorsque les messages d'entrée sont stockés dans la mémoire globale comme spécifié à la ligne 1 (Figure 3.9).

Version VLU

Cette version insère une mémoire locale qui est utilisée à la place de la mémoire globale pour supprimer le retard lié à un accès plus lent à la mémoire externe. La mémoire locale est insérée en remplaçant le type de mémoire __global par le type __local (Ligne 1, Figure 3.9). Le noyau VLU utilise ainsi des données stockées dans les BRAM plutôt que dans les mémoires externes au FPGA. L'optimisation du déroulement de boucle est également appliquée dans cette version telle qu'appliquée à la version VU.

Version VDM

Cette version flux de données est un modèle qui implémente l'algorithme SHA-3 sous forme de multi-noyaux interconnectés par des canaux de communication. Ce modèle permet aussi de réduire le trafic entre un FPGA et la mémoire externe au FPGA en découplant l'accès aux

mémoires aux opérations de calcul faites au sein du noyau. L'implémentation multi-noyaux proposée à la Figure 3.10 contient un premier noyau qui charge le message d'après les messages, un deuxième qui effectue les 24 itérations, et un troisième qui génère la sortie.

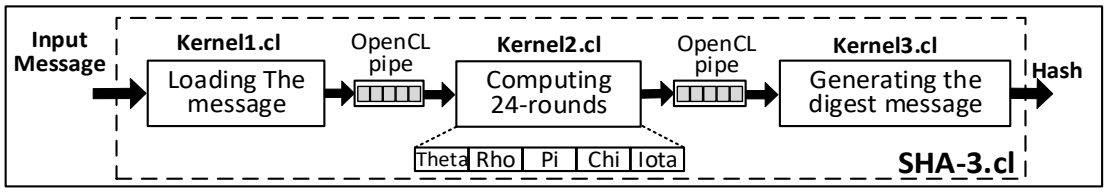


Figure 3.10 Structure multi-noyaux du SHA-3

Tableau 3.5 Les différentes versions du noyau du SHA-3

Versions	Technique d'optimisations appliquées	Type
VB	Version de base sans optimisation (Figure 3.9)	N/A
VU	Déroulement des boucles (U): (boucle 1.2.1 to 1.2.4)	Optimisation des boucles
VC	Loop Coalesce (C) (boucle 1)	
VI	Ivdep pragma (I) (boucle 1)	Optimisation spécifique au compilateur
VIU	Déroulement des boucles + Ivdep pragma (IU)	Combinaison de technique
VDM	Flot de données de multi-noyaux (DM)	Model d'exécution
VLU	Utilisation de mémoire locale + Déroulement des boucles	Combinaison de technique

3.3.2 Implémentations et résultats

Les noyaux répertoriés au Tableau 3.5 se sont avérés être les meilleures combinaisons possibles de techniques d'optimisation appliquée à l'algorithme SHA-3 en utilisant le flot de conception de Intel. Ces versions de noyau ont été implémentées en tant que noyaux de type *Single work-item* en utilisant le flot basé sur le compilateur *Intel FPGA SDK for OpenCL*.
Les résultats de performance sont obtenus en exécutant les différents noyaux avec plusieurs tailles d'éléments (*Data size*) allant de 100 octets à 1 Mo. Les données sont stockées dans la mémoire globale et transférées au FPGA sauf pour le noyau VLU, où les données sont stockées dans la mémoire locale à l'FPGA.

L'impact des techniques d'optimisation est évalué en mesurant le débit et l'accélération apportés par les différentes implémentations. Le débit (*throughput*) est calculé en divisant la taille des données traitées par le temps d'exécution du noyau. Les différents débits mesurés sont présentés à la Figure 3.11. Ces résultats montrent que le débit augmente avec la taille des éléments pour atteindre une valeur maximale stable, ce maximum est généralement limité par la bande passante du bus PCI-express. L'implémentation de base (VB) a un débit de 70 Mbit/s, tandis que les versions VI, VC, VU et VIU offrent un débit de 185 Mbit/s, 266 Mbit/s, 4,6 Gbit/s et 4,72 Gbit/s respectivement. La version multi-noyaux (VDM) est l'implémentation qui possède le débit le plus élevé (12,2 Gbit/s) parmi ceux qui utilisent des mémoires externes, ce qui représente une accélération de 169 fois par rapport à l'implémentation de base VB (Tableau 3.5). Ce résultat montre que les canaux de communication utilisés dans la version VDM augmentent l'efficacité de l'implémentation du SHA-3. Alternativement, le débit de la version VLU (non représenté sur la Figure 3.11) est évalué en utilisant des ensembles de données de taille 10 Ko à 100 Ko et le résultat montre qu'il est indépendant de la taille d'entrée, car les entrées sont stockées dans les mémoires internes au FPGA (BRAM). Le noyau VLU atteint un débit de 22,36 Gbit/s, ce qui représente une accélération de 310x par rapport au noyau VB non optimisé et le plus grand parmi toutes les versions implémentées (Tableau 3.6).

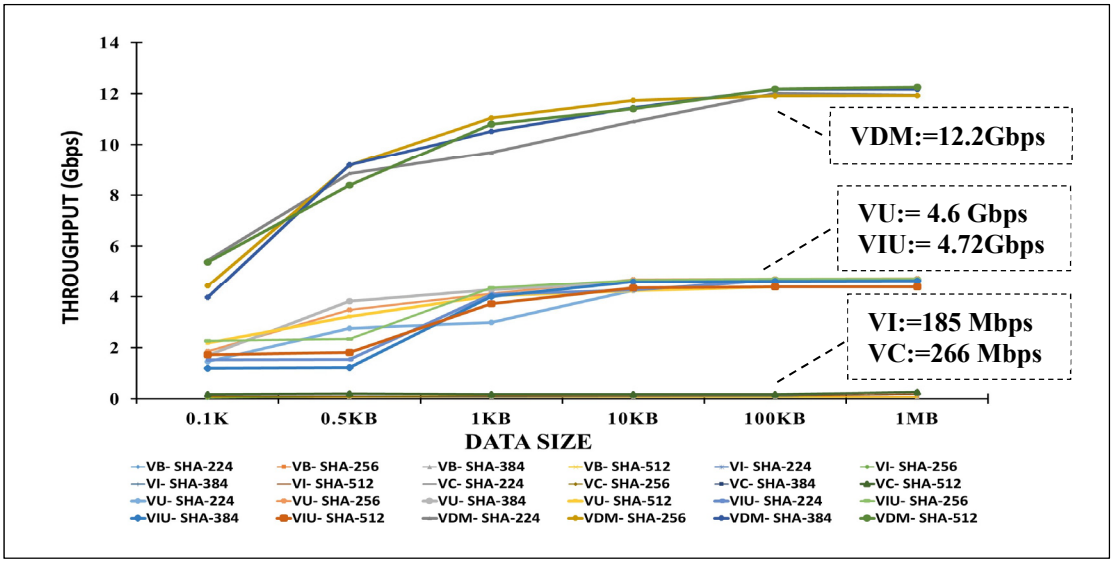


Figure 3.11 Débit des différentes versions d'implémentation de SHA-3

Tableau 3.6 Les performances des différentes implémentations du SHA-3

	Débit (Gbps)	Fréquence (MHz)	Accélération
VB	0.072	132	1×
VU	4.6	185	63×
VC	0.266	186	3.7×
VI	0.185	177	2.5×
VIU	4.72	180	65×
VDM	12.25	256	169×
VLU	22.36	312	310×

Le taux d'utilisation de ressources logique sur FPGA est également évalué et les taux moyens d'utilisation par rapport à la version de base sont présentés à la Figure 3.12. La moyenne est calculée en considérant les résultats de synthèse pour les quatre fonctions du SHA-3 (SHA-224, SHA-256, SHA-384 et SHA-512). La version VDM est l'implémentation la plus coûteuse en termes d'occupation logique sur FPGA et elle consomme 9,88% de ALUT et 8,58% de FF, tandis que le nombre de BRAM consommé diminue de 6,57 % par rapport à la version VB. La version VLU est l'implémentation la plus efficace et consomme 22,37% moins de ALUT et 18,48% moins de FF que l'implémentation de base, tandis que l'utilisation de BRAM augmente de 3,42% pour 20 Ko de mémoire locale insérée. Cette réduction de la consommation de ressources par rapport à l'implémentation de base s'explique par la surcharge logique supplémentaire pour interfacer la mémoire globale externe au FPGA lorsqu'elle est exploitée dans un noyau OpenCL. La fréquence est obtenue après la synthèse et la plus élevée est de 312 MHz offerte par la version VLU.

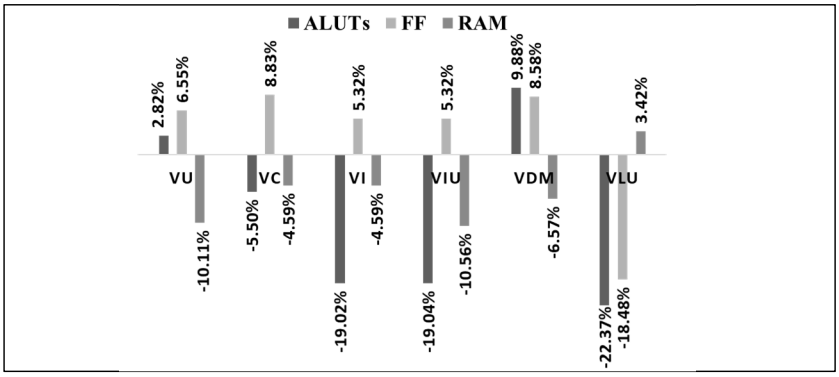


Figure 3.12 La surcharge logique moyenne sur FPGA comparée par rapport à la version de base du SHA-3 (VB)

Les meilleures performances en termes de débit et d'accélération du SHA-3, lorsqu'une mémoire globale externe au FPGA est utilisée, sont obtenues par l'implémentation VDM. Le noyau VDM est donc proposé pour les applications qui nécessitent une lourde charge de données d'entrées. Dans ce type d'applications, les mémoires externes au FPGA seront utilisées pour stocker des données. D'autre part, la version VLU est l'implémentation OpenCL proposée pour les applications qui nécessitent une quantité limitée de données qui peuvent être stockée dans la mémoire interne du FPGA et qui possèdent une forte intensité de calculs. Notons que les performances reportées du SHA-2 et SHA-3 sont comparées avec celles des travaux publiés dans la littérature à la section 4.3 du chapitre 4.

3.4 Conclusion

Dans ce chapitre, les techniques d'optimisation et les transformations du code OpenCL ont été présentées et évaluées sur les algorithmes SHA-2 et SHA-3. Les résultats présentés montrent que ces techniques sont nécessaires pour obtenir des implémentations performantes sur FPGA. De plus, ces résultats ont montré la grande sensibilité des compilateurs OpenCL pour FPGA au code de haut niveau. En effet, les optimisations destinées aux FPGA sont différentes de ceux destinés aux GPU ou CPU. Les transformations faites au niveau OpenCL manuellement ouvrent la porte vers un outil automatisé permettant d'identifier la technique utilisée ainsi d'adopter le code OpenCL adéquat.

CHAPITRE 4

DISCUSSION GÉNÉRALE

Ce chapitre discute les contributions présentées dans cette thèse. Premièrement, un résumé des contributions est présenté. Deuxièmement, les travaux présentés dans le cadre de cette thèse sont comparés aux travaux antérieurs. La section 4.2 évalue la plateforme d'instrumentation proposée et la section 4.3 compare les performances des algorithmes SHA-2 et SHA-3. Finalement, les travaux futurs recommandés sont présentés.

4.1 Résumé des contributions

Quatre contributions ont été présentées dans le deuxième et troisième chapitre. Les deux premières contributions répondent à l'objectif de proposer une solution permettant l'instrumentation des applications élaborées en OpenCL. En effet, la première contribution présente une preuve de concept d'un moniteur qui a été proposé pour valider la méthode d'instrumentation. Dans la deuxième contribution, une nouvelle plateforme d'instrumentation permettant l'extraction des performances temporelles des noyaux OpenCL a été proposée. Cette plateforme est illustrée par la Figure 4.1 où les étapes d'instrumentation et d'analyse de performance ont été ajoutées au flot de conception d'OpenCL. Les deux autres contributions présentées au chapitre 3 s'inscrivent dans le cadre de l'étude et l'évaluation des techniques d'optimisation et de transformations du code OpenCL pour FPGA. Les résultats empiriques présentés lors de l'optimisation des algorithmes SHA-2 et SHA-3 (chapitre 3) montrent que l'optimisation est une étape primordiale au développement des applications élaborées en OpenCL mises en œuvre sur des plateformes hétérogènes reconfigurables.

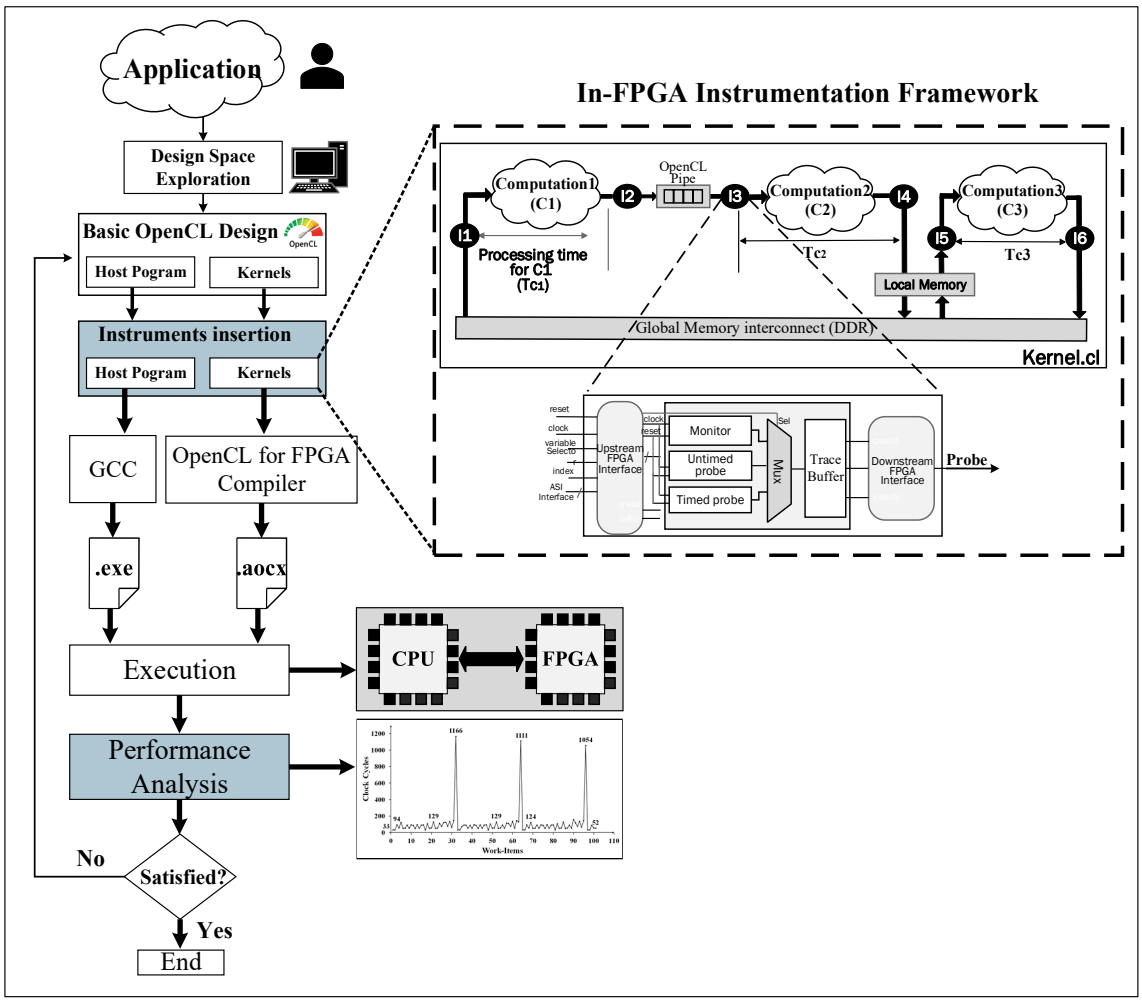


Figure 4.1 Illustration de la plateforme d'instrumentation proposée dans le flot OpenCL pour les plateformes hétérogènes reconfigurables

4.2 Comparaison de la plateforme d'instrumentation proposée

La plateforme d'instrumentation proposée dans le cadre de cette thèse permet de dévoiler les performances d'applications élaborées en OpenCL par l'extraction des performances temporelles clés comme le temps d'exécution d'éléments par une boucle ou une fonction donnée. Elle permet aussi d'identifier les décrochages lors du traitement de données. Ces performances permettent aux développeurs de détecter plus précisément les parties critiques à améliorer dans un noyau OpenCL.

La méthode d'instrumentation consiste à insérer dans le noyau des instruments développés en Verilog. L'utilisation de ces instruments exige seulement l'appel d'une fonction d'instrumentation appelée *instrument* ( ) (section 2.3.4, page 68) en spécifiant les arguments de cette fonction tels qu'ils sont définis dans fichier de la bibliothèque d'instrumentation (*header file*). Cette méthode facilite l'instrumentation puisqu'aucun développement matériel n'est nécessaire et le développeur logiciel doit seulement la fonction, le mode d'instrumentation et les arguments de cette fonction d'instrumentation. Notre plateforme offre trois modes d'instrumentation : (1) moniteur (*monitor*), (2) débogage (*untimed probe*) ou (3) les deux en mem temps (*timed probe*), comme expliqué à la section 2.3.4.1.

La surcharge logique reportée lorsqu'un seul instrument est inséré, comparée à une application sans instrument, est de 0.2 %, 0.1 % et 0.14 % en termes de ALUT, FF et LE, respectivement (Tableau 4.1). Les résultats montrent que notre plateforme consomme en moyenne près de 1,5 fois moins d'ALUT et 6 fois de moins de FF que les travaux publiés par J. S. Monson & Hutchings (2015), lorsqu'un instrument est inséré. En effet, le taux de surcharge en termes d'utilisation de FF est de 0,10 %, ce qui est comparé à un taux de 0,6 % dans le travail de J. S. Monson & Hutchings (2015), tandis que l'utilisation moyenne d'élément logique (LE) est de 2,5% dans le travail de Goeders & Wilton (2015). Les différents FPGA utilisés dans les travaux rapportés sont résumés par le Tableau 4.2.

Contrairement aux travaux présentés par Goeders et Wilton (Goeders & Wilton, 2014) (Goeders & Wilton, 2015), Jose Pablo (Jose Pablo Pinilla, 2016) et Monson et al (J. S. Monson & Hutchings, 2015), notre plateforme ne nécessite aucun accès au code du compilateur et ne modifie pas le flot de compilation de l'outil OpenCL (Figure 4.1). Au meilleur de nos connaissances, notre plateforme proposée est également la première à pouvoir simultanément prendre en charge, le débogage et l'analyse des performances temporelles avec une précision au cycle d'horloge du FPGA dédié aux applications élaborées en OpenCL et mise en œuvre sur les plateformes hétérogènes reconfigurables de type FPGA, comme l'illustre le Tableau 4.3. Bien que la preuve de concept n'ait été faite qu'avec l'environnement OpenCL d'Intel, la plateforme proposée peut être utilisée avec n'importe quel outil HLS qui supporte la

conception conjointe matérielle/logicielle, ce qui confirme sa portabilité sur d'autres fournisseurs d'FPGA, contrairement aux travaux rapportés (Jose Pablo Pinilla, 2016), (Goeders & Wilton, 2015) et (Xilinx, 2018a).

Tableau 4.1 Comparaison de l'utilisation des ressources logique par instrument (%)

	ALUT		**FF**		**LE**			
Nombre d'instruments	Notre travail	(J. S. Monson & Hutchings, 2015)	Notre travail	(J. S. Monson & Hutchings, 2015)	Notre travail	(Jose Pablo Pinilla, 2016)	(Goeders & Wilton, 2015)	(Verma et al., 2017)
2	0.2	0.3	0.1	0.6	0.14	13.9	2.5	1.09
10	0.13	N/A	0.17	N/A	0.26	N/A	N/A	N/A
20	0.23	N/A	0.16	N/A	0.18	N/A	N/A	N/A
50	0.19	N/A	0.17	N/A	0.15	N/A	N/A	N/A

Tableau 4.2 Les FPGA utilisés par les différents travaux

	Notre travail	**(J. S. Monson & Hutchings, 2015)**	**(Jose Pablo Pinilla, 2016)**	**(Verma et al., 2017)**
FPGA	Arria10	Zynq XC7Z020	Stratix V 5SGXEA7	StratixV/Arria10
Technologie	20 nm	28 nm	28 nm	28 nm/20 nm
Outil HLS	Intel SDK for OpenCL	Vivado HLS	LegUp	Intel SDK for OpenCL

Tableau 4.3 Comparaison des fonctionnalités offertes par diverses plateformes d'instrumentation

Fonctionnalités de la plateforme	**Notre plateforme**	**(Jose Pablo Pinilla, 2016)**	**(Goeders & Wilton, 2015)**	**(Xilinx, 2018a)**
Débogage	Oui	Oui	Oui	Oui
Analyse de performance temporelle	Oui	Non	Non	Non
Vérification fonctionnelle	Oui	Non	Non	Non

4.3 Comparaison de l'optimisation des implémentations OpenCL du SHA-2 et SHA-3

Les implémentations OpenCL du SHA-2 et SHA-3 peuvent être comparées aux implémentations faites avec un outil HLS ou avec un langage matériel. Par conséquent, nous comparons dans cette section les implémentations proposées aux travaux publiés dans la littérature.

4.3.1 Optimisation du SHA-2

Notre meilleure implémentation OpenCL obtenue du SHA-256 est la version VSUP (section 3.2.4) et elle est comparée à celles publiées dans la littérature (Tableau 4.4). Elle offre un débit 4.3 fois plus grand que la meilleure des implémentations précédemment publiées et conçues avec un outil HLS (3.97 Gbps comparés à 0.917 Gbps dans Kammoun, Elleuchi, Abid, & BenSaleh (2020).

Tableau 4.4 Comparaison avec les meilleurs résultats comparables publiés dans la littérature

	Débit (Gbps)	**Fréquence (MHz)**	**Nombre d'éléments logiques (LE)**	**Langage**	**FPGA**
Notre implémentation	3.97	243	67150	OpenCL	Arria10
(Kammoun et al., 2020)	0.917	181	6367	C-HLS	Zynq7000
(Rote, Vijendran, & Selvakumar, 2015)	2.047	271	905	HDL	Virtex6
(Y. Chen & Li, 2020)	1.98	255.7	979		Virtex4
(F. Kahri, Bouallegue, Machhout, & Tourki, 2013)	1.47	82	350		Virtex7
(Wong, Pudi, & Chattopadhyay, 2018)	1.405	354	197		Virtex6
(Mohamed & Nadjia, 2015)	1.35	170	1203		Virtex5
(Florin & Ionut, 2019)	1.21	151.5	N/A		Kintex7
(Gad, Abdalazeem, Abdelmegid, & Mostafa, 2020)	0.637	83.33	275		Virtex7

De plus, l'implémentation OpenCL proposée offre un débit supérieur à la meilleure implémentation publiée précédemment et qui est faite avec HDL (2.047 Gbps dans Rote et al., (2015)), comme indiqué dans le Tableau 4.4. Les implémentations OpenCL de SHA-256 consomment plus de ressources que les implémentations faites avec HDL et cela est dû à la complexité des circuits générés par les compilateurs OpenCL qui incluent des interfaces pour les mémoires externes et des interfaces PCI pour la communication CPU-FPGA.

4.3.2 Optimisation du SHA-3

La meilleure performance du SHA-3 est obtenue avec la version VLU (section 3.3.2). Le Tableau 4.5 compare les performances de notre implémentation VLU par rapport aux travaux antérieurs. Le débit obtenu avec l'implémentation VLU, qui est 22,36 Gbit/s, est 2 fois plus élevé que le meilleur débit reporté dans la littérature (Homsirikamol & Gaj, 2015) et qui est fait à l'aide d'un outil HLS. Ce débit est obtenu lorsque les mémoires internes au FPGA sont utilisées au lieu de mémoires externes. Le débit maximal atteint lors de l'utilisation de mémoires externes (implémentation VDM) est égal 12,2 Gbit/s. Il est aussi plus élevé que la meilleure implémentation, sans l'utilisation de mémoire externe, reportée dans la littérature (Jacinto et al., 2017) (Homsirikamol & Gaj, 2015). De plus, les débits de nos implémentations OpenCL du SHA-3 surpassent la plupart des implémentations publiées précédemment, incluant des implémentations développées en utilisant un langage HDL.

Tableau 4.5 Comparaison par rapport aux travaux antérieurs

	Débit (Gbps)	**Fréquence (MHz)**	**Langage**	**FPGA**
Notre implémentation (VLU)	22.36	312	OpenCL	Arria10
(Homsirikamol & Gaj, 2015)	11.35	271	C-HLS	Stratix IV
(Jacinto et al., 2017)	1.9348	124		Zynq-7000
(Arribas, 2019)	24.47/33.5	551	HDL	Virtex-5/6
(Sideris et al., 2020)	22.6	498.62		Arria10
(G. S. Athanasiou et al., 2014)	20.8	434		Virtex-7
(Fatma Kahri et al., 2016)	16.58	414.5		Virtex-7
(Wong, Haj-Yahya, et al., 2018)	16.51	344		Virtex-6
(Assad et al., 2019)	7.51	312		Virtex-5

Cependant, le meilleur débit obtenu est légèrement inférieur à celui rapporté dans les travaux de Sideris et al (2020) et (Arribas, 2019). Ces dernières implémentations rapportées dans la littérature sont faites au niveau RTL où des techniques d'optimisation sont aussi appliquées à l'architecture du SHA-3 à ce même niveau. Ces techniques incluent, sans s'y limiter, l'insertion d'étages de pipeline pour augmenter le débit.

La consommation logique de notre meilleure implémentation (version VLU) est comparée à celles publiées dans la littérature (Tableau 4.6). Le résultat reporté montre que notre implémentation consomme beaucoup plus de ressources logiques que les travaux publiés comme indiqué au Tableau 4.6. Cette consommation est dûe à la logique supplémentaire nécessaire aux interfaces de communication pour les mémoires externes et la communication CPU-FPGA. Il est important de mentionner que l'objectif principal est d'accélérer l'algorithme SHA-3 en appliquant les techniques présentées dans la section 3.3.1 et il est prévu que la consommation logique sera élevée par rapport aux implémentations faites avec un langage matériel.

Tableau 4.6 Comparaison des ressources logiques

	Slices	**ALUTs**	**FFs**	**LE**	**RAMs**	**FPGA**
Notre implémentation (VLU)	N/A	73664	100014	55536	787	Arria10
(Homsirikamol & Gaj, 2015)	N/A	N/A	N/A	N/A	N/A	Stratix IV
(Jacinto et al., 2017)	1174	N/A	N/A	N/A	N/A	Zynq-7000
(Arribas, 2019)	1507/ 1171	3344/ 4635	5887/ 5887	N/A	N/A	Virtex-5 Virtex-6
(Sideris et al., 2020)	1422	N/A	N/A	N/A	N/A	Arria10
(G. S. Athanasiou et al., 2014)	1618	N/A	N/A	N/A	N/A	Virtex-7
(Fatma Kahri et al., 2016)	N/A	N/A	N/A	N/A	N/A	Virtex-7
(Wong, Haj-Yahya, et al., 2018)	1406	N/A	N/A	N/A	N/A	Virtex-6
(Assad et al., 2019)	1304	N/A	N/A	N/A	N/A	Virtex-5

4.4 Discussion et travaux futurs recommandés

Les travaux présentés dans le cadre de cette thèse contribuent à deux domaines de recherche : 1) l'instrumentation des applications élaborées en OpenCL mises en œuvre sur une plateforme hétérogène reconfigurable de type FPGA+CPU et 2) l'optimisation des applications élaborées en OpenCL. Ces travaux contribuent à l'amélioration de la productivité de développement des applications en accélérant le processus d'instrumentation et en dévoilant l'impact des techniques d'optimisation sur les performances.

Les résultats expérimentaux obtenus dans le cadre de ces travaux montrent que :

1) notre plateforme d'instrumentation permet de dévoiler les performances des noyaux OpenCL en utilisant une méthode simple d'insertion d'instruments qui ne nécessite pas la modification du flot de compilation ni un outil supplémentaire. En effet, la plateforme proposée permet entre autres de tracer la courbe d'exécution des éléments d'entrées, de sorties ou de n'importe quelles variables dans le noyau ainsi que d'identifier les causes de décrochage. Ces informations temporelles permettent aux développeurs d'évaluer les performances des applications à la précision de l'horloge du FPGA et de détecter les parties critiques du noyau. Au meilleur de nos connaissances, ces informations temporelles ne peuvent pas être obtenues avec les solutions proposées dans la littérature, ainsi qu'avec les outils existants tels Chipscope de Xilinx et SignalTap d'Intel.
2) les techniques d'optimisation au niveau OpenCL sont cruciales pour obtenir de bonnes performances. En particulier, les techniques appliquées aux SHA-2 et SHA-3 améliorent les performances de 22.9 (version VSUP pour SHA-256) et 310 fois (version VLU pour SHA-3) par rapport à une implémentation non optimisée. Bien que les implémentations à base d'OpenCL du SHA-2 et SHA-3 offrent de meilleures performances en termes de débit par rapport à la majorité des travaux publiés dans la littérature, la motivation derrière l'optimisation de ces algorithmes est l'évaluation des techniques d'optimisation en étudiant leur impact sur les performances. Les techniques d'optimisation, qui peut-être aussi vu comme un style de codage, sont alors essentielles pour obtenir de meilleures performances. Ce style de codage doit prendre en considération la nature de l'architecture de

l'accélérateur utilisé lors de déploiement des applications puisque les techniques ciblant les FPGA ne sont évidemment pas efficaces pour les GPU.

L'exploration de l'espace de conception pour l'implémentation des algorithmes SHA-2 et SHA-3 a permis d'appliquer exhaustivement une combinaison de techniques d'optimisation. Cette exploration doit présentement être guidée manuellement par le développeur et plusieurs itérations d'exploration et d'évaluation sont nécessaires avant d'arriver à la meilleure implémentation. Ces itérations d'exploration et d'implémentation faites dans la présente thèse étaient en premier lieu une source d'apprentissage permettant d'identifier les techniques d'optimisation pour SHA-2 et SHA-3. Ces techniques explorées étaient :

- l'insertion d'une mémoire locale pour éviter la dépendance à la mémoire globale ;
- le déroulement des boucles ;
- le pipelinage des boucles ;
- le fractionnement des boucles ;
- l'insertion de canaux (*pipes*) (modèle multi-noyaux) ;
- l'insertion de directives propres aux compilateurs (exemple : pragma ivdep).

Néanmoins, les itérations d'exploration et d'évaluation deviennent à la fois très complexes et de longue durée et augmente significativement avec la complexité des algorithmes. De plus, le nombre d'implémentations ciblées pour évaluation ne cesse d'augmenter vu la diversité de techniques et des directives d'optimisation propres à chaque domaine d'applications et même à chaque outil HLS. Pour cette raison, les chercheurs continuent toujours d'étudier les méthodes DSE pour accélérer ou automatiser le processus d'exploration ainsi que pour identifier la meilleure combinaison des techniques d'optimisation qui offrent les meilleures performances. L'exploration de l'espace de conception est la première étape dans un flot de conception à base d'outil HLS tandis que l'instrumentation vient après l'exécution de l'application. Ces deux étapes sont nécessaires et complémentaires à l'optimisation des applications et s'ajoutent au flot de conception OpenCL offert par les fournisseurs des FPGA tel qu'illustré à la Figure 4.2.

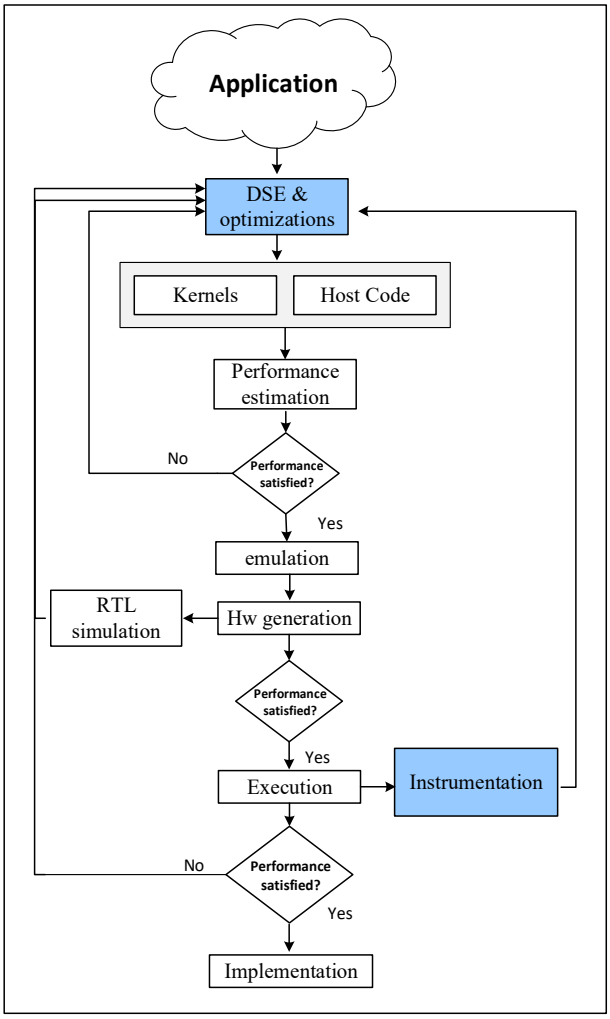


Figure 4.2 Flot de conception OpenCL pour FPGA

En élargissant les champs d'application, une approche automatisée est nécessaire. Cette approche doit utiliser les implémentations antérieures comme une nouvelle entrée à l'outil HLS. Une méthode DSE qui possède cette capacité d'apprentissage est l'exploration à base d'apprentissage automatique (*machine learning-based method*). Une méthode à base de réseaux de neurones permettant l'apprentissage en utilisant itérativement les résultats d'implémentations antérieures est proposée par Jihye Kwon & Luca P. Carloni (2020). Cependant, les applications évaluées dans ce dernier travail sont de faible complexité ainsi que la base utilisée pour l'apprentissage est limitée. Puisque le problème d'exploration de l'espace est très complexe et que les options d'implémentation sont nombreuses. Nous estimons qu'une

classification par domaine d'applications peut être faite en visant une technique d'exploration à base d'apprentissage pour chaque domaine d'applications.

Les travaux présentés dans le cadre de cette thèse ouvrent la voie vers les idées de recherche suivantes :

- développer et optimiser d'autres algorithmes de cryptographie (AES, RSA, DSA, etc.) pour fournir une bibliothèque de cryptographie en OpenCL. Cette bibliothèque pourrait être utilisée dans des applications de haute performance élaborées avec OpenCL ;
- élaborer une base des techniques d'optimisation classifiées selon le domaine d'application ;
- évaluer les techniques d'optimisation à base du partitionnement et de la structure des données (exemple : Array of Structures, Structure of Arrays et Double-buffering) ;
- évaluer les techniques de vectorisation (nombre de CU et SIMD) ciblant les noyaux de type NDRange pour FPGA ;
- étudier les compromis entre la vectorisation et l'énergie consommée ;
- identifier les techniques de conception des circuits à faible consommation de puissance dans les descriptions à haut niveau d'abstraction ;
- élaborer une méthode d'exploration à base d'une machine d'apprentissage supervisé qui prend comme entrées les performances extraites par la plateforme d'instrumentation.

CONCLUSION

Les plateformes hétérogènes reconfigurables se sont révélées très prometteuses pour augmenter la capacité de calcul des applications nécessitant de hautes performances. La productivité de développement avec ces plateformes est améliorée grâce aux outils HLS qui permettent la programmation des CPU et FPGA à un même niveau d'abstraction. En effet, les outils HLS permettent la configuration des FPGA en utilisant des langages de haut niveau tel l'OpenCL. Bien que l'outil OpenCL pour FPGA a amélioré la programmabilité des FPGA, il conduit souvent à de mauvaises performances en comparaison de celles conçues avec des langages de description matérielle. Par ailleurs, l'implémentation matérielle générée est complètement abstraite aux développeurs, ce qui complique son instrumentation et son optimisation visant l'amélioration de performances. Les travaux présentés dans le cadre de cette thèse avaient comme objectifs de : 1) proposer une solution permettant l'amélioration d'observabilité à l'intérieur des implémentations matérielles élaborées à la base en OpenCL, 2) contribuer à l'optimisation des applications élaborées en OpenCL mise en œuvre sur des plateformes hétérogènes reconfigurables de type FPGA. Le premier objectif fut atteint par la réalisation d'une plateforme d'instrumentation pour les applications élaborées OpenCL, mise en œuvre sur des plateformes hétérogènes reconfigurables, décrite au chapitre 2. Ces premiers travaux ont mené à deux articles dont :

1) un article a été publié à la conférence ISCAS (Bensalem et al., 2019) dans laquelle une preuve du concept d'intégration d'un moniteur est présentée ;
2) un second a été publié dans la revue IEEE Access (Bensalem et al., 2020) où une nouvelle plateforme d'instrumentation pour extraire les performances temporelles des noyaux OpenCL a été proposée.

En réponse au deuxième objectif, des techniques d'optimisation dans un environnement HLS ont été exhaustivement explorées et appliquées pour améliorer la performance des algorithmes SHA-2 et SHA-3 sur une plateforme HPC intégrant des FPGA, décrite au chapitre 3. Les résultats d'implémentation du SHA-2 (SHA-256) ont entre autres montré que l'application de techniques d'optimisation améliore le débit de traitement de données par un facteur de 22.9

par rapport à une implémentation non optimisée, avec un débit de 3.97 Gbps qui est également 4.3 fois plus élevé que le meilleur débit reporté dans les travaux publiés dans la littérature. Les résultats de plusieurs implémentations de SHA-3 avec différentes techniques d'optimisation montrent un débit de traitement de données amélioré par un facteur de 310 par rapport à une implémentation non optimisée, soit de 22.36 Gbps qui est 2 fois plus élevé que les meilleures implémentations publiées dans la littérature. Ces résultats ont conduit à deux autres articles dont :

3) un publié à la conférence ISCAS 2021, intitulé *Acceleration of the Secure Hash Algorithm-256 (SHA-256) on an FPGA-CPU Cluster Using OpenCL* (Bensalem et al., 2021);
4) et un second récemment soumis à la revue IEEE TCAS-II intitulé *An Efficient OpenCL-Based Implementation of a SHA-3 Co-Processor on an FPGA-Centric Platform*.

Ces contributions ouvrent la voie vers une méthodologie de conception plus automatisée d'accélérateurs intégrant des FPGA élaborés en OpenCL, en offrant un environnement de développement avec des instruments de diagnostic et une meilleure compréhension des techniques d'optimisation.

À notre connaissance, nos travaux de recherche sont uniques et prometteurs, car ils contribuent à offrir une infrastructure de programmation multiparadigme qui intègre non seulement un outil permettant l'optimisation automatisée des applications élaborées en OpenCL, mais aussi un outil de support à l'instrumentation et au débogage. Bien que ces travaux aient montré que l'utilisation de techniques d'optimisation dans un environnement HLS peut améliorer considérablement la performance, une étude plus large et une automatisation de processus d'identification des techniques à appliquer sont nécessaires. Une telle automatisation de l'exploration de l'espace de conception ouvre les pistes de recherches suivantes comme :

- l'étude et l'application de l'apprentissage supervisé pour améliorer l'exploration de l'espace de conception. Ceci pourrait se faire grâce à un moteur d'apprentissage alimenté par les performances extraites avec la plateforme d'instrumentation ainsi que par les résultats obtenus par les implémentations évaluées ;

- l’unification des outils d’exploration, d’instrumentation et d’optimisation pour fournir un environnement complet visant à améliorer la productivité du processus de développement à base de l’OpenCL.

LISTE DE RÉFÉRENCES BIBLIOGRAPHIQUES

Aaftab, M. (2011). The OpenCL specification version: Khronos Group.

Aaftab, M., Benedict, G., G, M. T., & Dan, G. (2011). OpenCL programming guide. Pearson Education.

Abel, F., Weerasinghe, J., Hagleitner, C., Weiss, B., & Paredes, S. (2017). An FPGA Platform for Hyperscalers. Dans 2017 IEEE 25th Annual Symposium on High-Performance Interconnects (HOTI) (pp. 29-32). doi: 10.1109/HOTI.2017.13

Aisopos, K., Kakarountas, A. P., Michail, H., & Goutis, C. E. (2005). High throughput implementation of the new Secure Hash Algorithm through partial unrolling. Dans IEEE Workshop on Signal Processing Systems Design and Implementation, 2005. (pp. 99-103). IEEE.

Al-Odat, Z. A., Ali, M., Abbas, A., & Khan, S. U. (2020). Secure hash algorithms and the corresponding fpga optimization techniques. ACM Computing Surveys (CSUR), 53(5), 1-36.

Al-Odat, Z. A., Ali, M., Abbas, A., & Khan, S. U. (2020). Secure Hash Algorithms and the Corresponding FPGA Optimization Techniques. ACM Comput. Surv., 53(5), Article 97. Repéré à https://doi.org/10.1145/3311724.

AlphaData. Alpha Data ADM-PCIE-7V3 datasheet. [Online]. Répéré à http://www.alpha-data.com/pdfs/adm-pcie-7v3.pdf.

Altera. (2013). Quartus Handbook version 13.1 : Design Debugging Using the SignalTap II Logic Analyzer. Volume 3: verification.

Amazon. (2017). Amazon EC2 F1 Instance. Répéré à https://aws.amazon.com/ec2/instance-types/f1/.

Andrade, H., Lwakatare, L. E., Crnkovic, I., & Bosch, J. (2019). Software Challenges in Heterogeneous Computing: A Multiple Case Study in Industry. Dans 2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA) (pp. 148-155). doi: 10.1109/SEAA.2019.00031.

Andrade, H., Schroeder, J., & Crnkovic, I. (2019). Software deployment on heterogeneous platforms: A systematic mapping study. IEEE Transactions on Software Engineering.

Arribas, V. (2019). Beyond the Limits: SHA-3 in Just 49 Slices. Dans 2019 29th International Conference on Field Programmable Logic and Applications (FPL) (pp. 239-245). doi: 10.1109/FPL.2019.00044.

Arvind, R. S. N. (2009). What is Bluespec? SIGDA Newsl., 39(1), 1. doi:10.1145/1862876.1862877. Repéré à https://doi.org/10.1145/1862876.1862877.

Assad, F., Elotmani, F., Fettach, M., & Tragha, A. (2019). An optimal hardware implementation of the KECCAK hash function on virtex-5 FPGA. Dans 2019 International Conference on Systems of Collaboration Big Data, Internet of Things & Security (SysCoBIoTS) (pp. 1-5). doi: 10.1109/SysCoBIoTS48768.2019.9028020.

Athanasiou, G. S., Makkas, G., & Theodoridis, G. (2014). High throughput pipelined FPGA implementation of the new SHA-3 cryptographic hash algorithm. Dans 2014 6th International Symposium on Communications, Control and Signal Processing (ISCCSP) (pp. 538-541). doi: 10.1109/ISCCSP.2014.6877931.

Athanasiou, G. S., Michail, H. E., Theodoridis, G., & Goutis, C. E. (2013). Optimising the SHA-512 cryptographic hash function on FPGAs. IET Computers & Digital Techniques, 8(2), 70-82.

Bell, S., Pu, J., Hegarty, J., & Horowitz, M. (2018). Compiling algorithms for heterogeneous systems. Synthesis Lectures on Computer Architecture, 13(1), 1-105.

Bensalem, H., Blaquière, Y., & Savaria, Y. (2019). Toward In-System Monitoring of OpenCL-Based Designs on FPGA. Dans IEEE International Symposium on Circuits and Systems (ISCAS) (pp. 1-5). doi: 10.1109/ISCAS.2019.8702551.

Bensalem, H., Blaquière, Y., & Savaria, Y. (2020). In-FPGA Instrumentation Framework for OpenCL-Based Designs. IEEE Access, 8, 212979-212994. doi: 10.1109/ACCESS.2020.3040081.

Bensalem, H., Blaquière, Y., & Savaria, Y. (2021). Acceleration of the Secure Hash Algorithm-256 (SHA-256) on an FPGA-CPU Cluster Using OpenCL. Dans 2021 IEEE International Symposium on Circuits and Systems (ISCAS) (pp. 1-5). doi: 10.1109/ISCAS51556.2021.9401197.

Bertoni, G., Daemen, J., Peeters, M., & Van Assche, G. (2009). Keccak specifications. Submission to NIST (round 2), 320-337.

Biberman, A., & Bergman, K. (2012). Optical interconnection networks for high-performance computing systems. Reports on Progress in Physics, 75(4), 046402.

Bussa, P. K., Goeders, J., & Wilton, S. J. E. (2017). Accelerating in-system FPGA debug of high-level synthesis circuits using incremental compilation techniques. Dans 27th

International Conference on Field Programmable Logic and Applications (FPL) (pp. 1-4). doi: 10.23919/FPL.2017.8056800.

Calagar, N., Brown, S. D., & Anderson, J. H. (2014). Source-level debugging for FPGA high-level synthesis. Dans 24th International Conference on Field Programmable Logic and Applications (FPL) (pp. 1-8). doi: 10.1109/FPL.2014.6927496.

Canis, A., Choi, J., Aldham, M., Zhang, V., Kammoona, A., Anderson, J. H., Czajkowski, T. (2011). LegUp: high-level synthesis for FPGA-based processor/accelerator systems. Dans Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays (pp. 33-36). doi: 10.1145/1950413.1950423.

Canis, A., Choi, J., Aldham, M., Zhang, V., Kammoona, A., Czajkowski, T., Anderson, J. (2013). LegUp: An open-source high-level synthesis tool for FPGA-based processor/accelerator systems. ACM Transactions on Embedded Computing Systems (TECS), 13(2), 1-27. doi: 10.1145/2514740.

Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., Lee, S.-H., & Skadron, K. (2009). Rodinia: A benchmark suite for heterogeneous computing. Dans 2009 IEEE international symposium on workload characterization (IISWC) (pp. 44-54). doi: 10.1109/IISWC.2009.5306797.

Chellappa, S., Franchetti, F., & Püschel, M. (2008). How to Write Fast Numerical Code: A Small Introduction. Dans R. Lämmel, J. Visser & J. Saraiva (Éds.), Generative and Transformational Techniques in Software Engineering II: International Summer School, GTTSE 2007, Braga, Portugal, July 2-7, 2007. Revised Papers (pp. 196-259). Berlin, Heidelberg: Springer Berlin Heidelberg. Repéré à https://doi.org/10.1007/978-3-540-88643-3_5.

Chen, T., Moreau, T., Jiang, Z., Shen, H., Yan, E. Q., Wang, L., Krishnamurthy, A. (2018). TVM: end-to-end optimization stack for deep learning. arXiv preprint arXiv:1802.04799, 11, 20.

Chen, Y., & Li, S. (2020). A High-Throughput Hardware Implementation of SHA-256 Algorithm. Dans 2020 IEEE International Symposium on Circuits and Systems (ISCAS) (pp. 1-4). doi: 10.1109/ISCAS45731.2020.9181065.

Choi, Y.-k. (2019). Performance Debugging Frameworks for FPGA High-Level Synthesis (UCLA).

Choi, Y.-k., Cong, J., Fang, Z., Hao, Y., Reinman, G., & Wei, P. (2016). A quantitative analysis on microarchitectures of modern CPU-FPGA platforms présentée à 2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC), Austin, TX, USA. doi: 10.1145/2897937.2897972. Repéré à https://doi.org/10.1145/2897937.2897972.

Choi, Y.-K., Cong, J., Fang, Z., Hao, Y., Reinman, G., & Wei, P. (2019). In-Depth Analysis on Microarchitectures of Modern Heterogeneous CPU-FPGA Platforms. ACM Trans. Reconfigurable Technol. Syst., 12(1), Article 4. doi: 10.1145/3294054. Repéré à https://doi.org/10.1145/3294054.

Choi, Y. K., & Cong, J. (2017). HLScope: High-Level Performance Debugging for FPGA Designs. Dans IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM) (pp. 125-128). doi: 10.1109/FCCM.2017.44.

Cong, J., Liu, B., Neuendorffer, S., Noguera, J., Vissers, K., & Zhang, Z. (2011). High-level synthesis for FPGAs: From prototyping to deployment. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 30(4), 473-491. doi: 10.1109/TCAD.2011.2110592.

Cong, J., Sarkar, V., Reinman, G., & Bui, A. (2010). Customizable domain-specific computing. IEEE Design & Test of Computers, 28(2), 6-15. doi: 10.1109/MDT.2010.141.

Coussy, P., Gajski, D. D., Meredith, M., & Takach, A. (2009). An Introduction to High-Level Synthesis. IEEE Design & Test of Computers, 26(4), 8-17. doi: 10.1109/MDT.2009.69

Bertoni, G., DAEMEN, J., Peeters, M., & Assche, G. (2011). The keccak SHA-3 submission Version. Repéré à https://keccak.team/files/Keccak-submission-3.pdf

Derek, C. (2017). The Microsoft catapult project. Dans 2017 IEEE International Symposium on Workload Characterization (IISWC) (pp. 124-124). IEEE Computer Society. doi: 10.1109/IISWC.2017.8167769.

Dhar, V. (2013). Data science and prediction. Communications of the ACM, 56(12), 64-73.

Esmaeilzadeh, H., Blem, E., Amant, R. S., Sankaralingam, K., & Burger, D. (2012). Dark silicon and the end of multicore scaling. IEEE Micro, 32(3), 122-134.

Farahmand, F., Homsirikamol, E., & Gaj, K. (2016). A Zynq-based testbed for the experimental benchmarking of algorithms competing in cryptographic contests. Dans 2016 International Conference on ReConFigurable Computing and FPGAs (ReConFig) (pp. 1-7). doi: 10.1109/ReConFig.2016.7857148.

FIPS. (1995). FIPS Pub 180-1 Secure Hash Standard. National Institute of Standards and Technology, 17, 15.

Flake, P., Moorby, P., Golson, S., Salz, A., & Davidmann, S. (2020). Verilog HDL and its ancestors and descendants. Proc. ACM Program. Lang., 4(HOPL), Article 87. doi: 10.1145/3386337. Repéré à https://doi.org/10.1145/3386337.

Florin, R., & Ionut, R. (2019). FPGA based architecture for securing IoT with blockchain. Dans 2019 International Conference on Speech Technology and Human-Computer Dialogue (SpeD) (pp. 1-8). doi: 10.1109/SPED.2019.8906595.

Gad, A. H., Abdalazeem, S. E. E., Abdelmegid, O. A., & Mostafa, H. (2020). Low power and area SHA-256 hardware accelerator on Virtex-7 FPGA. Dans 2020 2nd Novel Intelligent and Leading Emerging Sciences Conference (NILES) (pp. 181-185). doi: 10.1109/NILES50944.2020.9257922.

Gawiejnowicz, S. (2020). Heuristic algorithms. Dans S. Gawiejnowicz (Éd.), Models and Algorithms of Time-Dependent Scheduling (pp. 301-327). Berlin, Heidelberg: Springer Berlin Heidelberg. doi: 10.1007/978-3-662-59362-2_16. Repéré à https://doi.org/10.1007/978-3-662-59362-2_16.

Gepner, P., & Kowalik, M. F. (2006). Multi-core processors: New way to achieve high system performance. Dans International Symposium on Parallel Computing in Electrical Engineering (PARELEC'06) (pp. 9-13). IEEE.

Goeders, J., & Wilton, S. J. E. (2014). Effective FPGA debug for high-level synthesis generated circuits. Dans 24th International Conference on Field Programmable Logic and Applications (FPL) (pp. 1-8). doi: 10.1109/FPL.2014.6927498.

Goeders, J., & Wilton, S. J. E. (2015). Using Dynamic Signal-Tracing to Debug Compiler-Optimized HLS Circuits on FPGAs. Dans 2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines (pp. 127-134). doi: 10.1109/FCCM.2015.25.

Goodfellow, I., Bengio, Y., Courville, A., (2016). Deep learning (Vol. 1). MIT press Cambridge.

Grauer-Gray, S., Xu, L., Searles, R., Ayalasomayajula, S., & Cavazos, J. (2012). Auto-tuning a high-level language targeted to GPU codes. Dans 2012 Innovative Parallel Computing (InPar) (pp. 1-10). doi: 10.1109/InPar.2012.6339595.

Homsirikamol, E., & Gaj, K. (2015). Hardware Benchmarking of Cryptographic Algorithms Using High-Level Synthesis Tools: The SHA-3 Contest Case Study. Dans (pp. 217-228). Springer International Publishing.

Hsieh, J. Y. J., Mahoney, J. E., Ngo, L. T., & Shelly, L. (1986). A User Programmable Reconfigurable Logic Array. .

Huawei. (2019). FPGA-accelerated cloud server. Repéré à https://www.huaweicloud.com/en-us/product/fcs.html.

Hung, E., & Wilton, S. J. E. (2011). Speculative Debug Insertion for FPGAs. Dans 21st International Conference on Field Programmable Logic and Applications (pp. 524-531). doi: 10.1109/FPL.2011.103.

Hussain, K., Salleh, M. N. M., Cheng, S., & Shi, Y. (2019). Metaheuristic research: a comprehensive survey. Artificial Intelligence Review, 52(4), 2191-2233.

Intel. (2013). Implementing FPGA Design with the OpenCL Standard. White paper.

Intel. (2019). Intel® FPGA SDK for OpenCL™ Support (Design Examples). Répéré à https://www.intel.ca/content/www/ca/en/programmable/products/design-software/embedded-software-developers/opencl/support.html. Consulté en Juin 2019.

Intel. (2020a). Intel FPGA SDK for OpenCL™ Pro Edition Programming Guide, v20.2.

Intel. (2020b). Intel VTune Amplifier. Repéré à https://www.intel.com

Intel. (2021a). Avalon Interface Specifications. Version 20.1 Repéré à https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/manual/mnl_avalon_spec.pdf

Intel. (2021b). Intel High Level Synthesis Compiler Pro Edition : Reference Manual (MNL-1083).

Iyer, S. S. (2016). Heterogeneous Integration for Performance and Scaling. IEEE Transactions on Components, Packaging and Manufacturing Technology, 6(7), 973-982. doi: 10.1109/TCPMT.2015.2511626.

Jacinto, H. S., Daoud, L., & Rafla, N. (2017). High level synthesis using vivado HLS for optimizations of SHA-3. Dans 2017 IEEE 60th International Midwest Symposium on Circuits and Systems (MWSCAS) (pp. 563-566). doi: 10.1109/MWSCAS.2017.8052985.

Jack, D., & L, L. A. (2009). High performance heterogeneous computing (Vol. 78). John Wiley & Sons.

Jordan, M. I., & Mitchell, T. M. (2015). Machine learning: Trends, perspectives, and prospects. Science, 349(6245), 255-260.

Kahri, F., Bouallegue, B., Machhout, M., & Tourki, R. (2013). An FPGA implementation and comparison of the SHA-256 and Blake-256. Dans 14th International Conference on Sciences and Techniques of Automatic Control & Computer Engineering - STA'2013 (pp. 152-157). doi: 10.1109/STA.2013.6783122.

Kahri, F., Mestiri, H., Bouallegue, B., & Machhout, M. (2016). High speed FPGA implementation of cryptographic KECCAK hash function crypto-processor. Journal of Circuits, Systems and Computers, 25(04), 1650026.

Kaliski, B. S. (1990). The MD4 message digest algorithm. Dans Workshop on the Theory and Application of of Cryptographic Techniques (pp. 492-492). Springer.

Kammoun, M., Elleuchi, M., Abid, M., & BenSaleh, M. S. (2020). FPGA-based implementation of the SHA-256 hash algorithm. Dans 2020 IEEE International Conference on Design & Test of Integrated Micro & Nano-Systems (DTS) (pp. 1-6). doi: 10.1109/DTS48731.2020.9196134.

Kathail, V. (2020). Xilinx Vitis Unified Software Platform présentée à Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Seaside, CA, USA. doi: 10.1145/3373087.3375887. Repéré à https://doi.org/10.1145/3373087.3375887.

KhronosGroup. (2011, October 2011). The OpenCL Specification: Version 1.0.

Koch, D., Ziener, D., & Hannig, F. (2016). FPGA Versus Software Programming: Why, When, and How? Dans D. Koch, F. Hannig & D. Ziener (Éds.), FPGAs for Software Programmers (pp. 1-21). Cham: Springer International Publishing. doi: 10.1007/978-3-319-26408-0_1. Repéré à https://doi.org/10.1007/978-3-319-26408-0_1.

Kowalski, T., Geiger, D., Wolf, W., & Fichtner, W. (1985). The VLSI design automation assistant: From algorithms to silicon. IEEE Design & Test of Computers, 2(4), 33-43.

Kowalski, T. J., & Thomas, D. E. (1983). The VLSI design automation assistant: Prototype system. Dans 20th Design Automation Conference Proceedings (pp. 479-483). doi: 10.1109/MDT.1985.294721.

Kowalski, T. J., & Thomas, D. E. (1985). The VLSI design automation assistant: What's in a knowledge base. Dans 22nd ACM/IEEE Design Automation Conference (pp. 252-258). doi: 10.1109/DAC.1985.1585949.

Krommydas, K. (2017). Towards enhancing performance, programmability, and portability in heterogeneous computing (Virginia Tech).

Krommydas, K., Helal, A. E., Verma, A., & Feng, W.-C. (2016). Bridging the performance-programmability gap for fpgas via opencl: A case study with opendwarfs. Department of Computer Science, Virginia Polytechnic Institute & State.

Kwon, J., & Carloni, L. P. (2020). Transfer Learning for Design-Space Exploration with High-Level Synthesis. Dans Proceedings of the 2020 ACM/IEEE Workshop on Machine Learning for CAD (pp. 163-168). doi: 10.1145/3380446.3430636.

Kwon, J., & Carloni, L. P. (2020). Transfer Learning for Design-Space Exploration with High-Level Synthesis présentée à Proceedings of the 2020 ACM/IEEE Workshop on Machine Learning for CAD, Virtual Event, Iceland. doi: 10.1145/3380446.3430636. Repéré à https://doi.org/10.1145/3380446.3430636.

Lahti, S., Sjövall, P., Vanne, J., & Hämäläinen, T. D. (2018). Are we there yet? A study on the state of high-level synthesis. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 38(5), 898-911. doi: 10.1109/TCAD.2018.2834439.

Lahti, S., Sjövall, P., Vanne, J., & Hämäläinen, T. D. (2019). Are We There Yet? A Study on the State of High-Level Synthesis. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 38(5), 898-911. doi: 10.1109/TCAD.2018.2834439.

Lai, Y.-H., Chi, Y., Hu, Y., Wang, J., Yu, C. H., Zhou, Y., Zhang, Z. (2019). HeteroCL: A Multi-Paradigm Programming Infrastructure for Software-Defined Reconfigurable Computing présentée à Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Seaside, CA, USA. doi: 10.1145/3289602.3293910. Repéré à https://doi.org/10.1145/3289602.3293910.

Latif, K., Rao, M. M., Mahboob, A., & Aziz, A. (2012). Novel arithmetic architecture for high performance implementation of SHA-3 finalist KECCAK on FPGA platforms. Dans International Symposium on Applied Reconfigurable Computing (pp. 372-378). Springer.

Lau, J., Sivaraman, A., Zhang, Q., Gulzar, M. A., Cong, J., & Kim, M. (2020). HeteroRefactor: Refactoring for Heterogeneous Computing with FPGA. Dans 2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE) (pp. 493-505).

Lau, J. H. (2011). Evolution, challenge, and outlook of TSV, 3D IC integration and 3d silicon integration. Dans Advanced Packaging Materials (APM), 2011 International Symposium on (pp. 462-488). doi: 10.1109/ISAPM.2011.6105753.

Liang, Y., Wang, S., & Zhang, W. (2018). FlexCL: A model of performance and power for OpenCL workloads on FPGAs. IEEE Transactions on Computers, 67(12), 1750-1764. doi: 10.1109/TC.2018.2840686.

Lim, R., Norris, B., & Malony, A. (2017). Autotuning GPU Kernels via Static and Predictive Analysis. Dans 2017 46th International Conference on Parallel Processing (ICPP) (pp. 523-532). doi: 10.1109/ICPP.2017.61. doi: 10.1109/ICPP.2017.61.

Lipsett, R., Marschner, E., & Shahdad, M. (1986). VHDL-the language. IEEE Design & Test of Computers, 3(2), 28-41.

Lis, J. S., & Gajski, D. D. (1988). Synthesis from VHDL. Dans Proceedings 1988 IEEE International Conference on Computer Design: VLSI (pp. 378,379,380,381-378,379,380,381). IEEE Computer Society.

Liu, B., Zydek, D., Selvaraj, H., & Gewali, L. (2012). Accelerating high performance computing applications: Using cpus, gpus, hybrid cpu/gpu, and fpgas. Dans 2012 13th International Conference on Parallel and Distributed Computing, Applications and Technologies (pp. 337-342). IEEE.

Luebke, D. (2008). CUDA: Scalable parallel programming for high-performance scientific computing. Dans 2008 5th IEEE international symposium on biomedical imaging: from nano to macro (pp. 836-838). doi: 10.1109/PDCAT.2012.34.

Mansureh, M. S., Cho, J.-M., & Choi, K. (2017). Reconfigurable Architectures. Dans S. Ha & J. Teich (Éds.), Handbook of Hardware/Software Codesign (pp. 1-42). Dordrecht: Springer Netherlands. doi: 10.1007/978-94-017-7358-4_12-1. Repéré à https://doi.org/10.1007/978-94-017-7358-4_12-1.

Martino, R., & Cilardo, A. (2020). SHA-2 Acceleration Meeting the Needs of Emerging Applications: A Comparative Survey. IEEE Access, 8, 28415-28436. doi: 10.1109/ACCESS.2020.2972265. doi: 10.1109/ACCESS.2020.2972265.

Mohamed, A., & Nadjia, A. (2015). SHA-2 hardware core for virtex-5 FPGA. Dans 2015 IEEE 12th International Multi-Conference on Systems, Signals & Devices (SSD15) (pp. 1-5). doi: 10.1109/SSD.2015.7348110.

Monroe, D. (2018). Chips for artificial intelligence. Communications of the ACM, 61(4), 15-17.

Monson, J. S., & Hutchings, B. (2014). New approaches for in-system debug of behaviorally-synthesized FPGA circuits. Dans 2014 24th International Conference on Field Programmable Logic and Applications (FPL) (pp. 1-6). doi:10.1109/FPL.2014.6927495.

Monson, J. S., & Hutchings, B. (2015). Using source-to-source compilation to instrument circuits for debug with High Level Synthesis. Dans International Conference on Field Programmable Technology (FPT) (pp. 48-55). doi: 10.1109/FPT.2015.7393129.

Monson, J. S., & Hutchings, B. L. (2018). Enhancing debug observability for HLS-based FPGA circuits through source-to-source compilation. Journal of Parallel and Distributed Computing, 117, 148-160. doi: https://doi.org/10.1016/j.jpdc.2018.02.012. Repéré à http://www.sciencedirect.com/science/article/pii/S0743731518300893.

Moore, G. (1965). Moore's law. Electronics Magazine, 38(8), 114.

Muralidharan, S., Roy, A., Hall, M., Garland, M., & Rai, P. (2016). Architecture-Adaptive Code Variant Tuning présentée à Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, Atlanta, Georgia, USA. doi: 10.1145/2872362.2872411. Repéré à https://doi.org/10.1145/2872362.2872411.

Najjar, W. A., Villarreal, J., & Halstead, R. J. (2016). ROCCC 2.0. Dans FPGAs for Software Programmers (pp. 191-204). Springer.

Nallatech. (2021). Nallatech 510T compute acceleration card. Repéré à https://www.bittware.com/fpga/510t/.

Nane, R., Sima, V., Olivier, B., Meeuws, R., Yankova, Y., & Bertels, K. (2012). DWARV 2.0: A CoSy-based C-to-VHDL hardware compiler. Dans 22nd International Conference on Field Programmable Logic and Applications (FPL) (pp. 619-622). doi:10.1109/FPL.2012.6339221.

Nane, R., Sima, V., Pilato, C., Choi, J., Fort, B., Canis, A., Bertels, K. (2016). A Survey and Evaluation of FPGA High-Level Synthesis Tools. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 35(10), 1591-1604. doi: 10.1109/TCAD.2015.2513673.

Ndu, G., Navaridas, J., & Luján, M. (2015). CHO: towards a benchmark suite for OpenCL FPGA accelerators. Dans Proceedings of the 3rd International Workshop on OpenCL (pp. 10). doi: 10.1145/2791321.2791331.

NIST-FIPS. (1993). PUB 180, Secure hash standard. National Institute of Standards and Technology, US Department of Commerce, DRAFT.

NIST. (2004). SHA-2 Secure Hash Standard, FIPS PUB 180-2. Federal Information Processing Standard (FIPS).

NIST. (2015). “SHA-3 standard: Permutation-based hash and extendable-output functions, FIPS, PUB 202.

Nunez-Yanez, J., Hosseinabady, M., Rodríguez, A., Asenjo, R., Navarro, A., Gran-Tejero, R., & Suárez-Gracia, D. (2018). Simultaneous Multiprocessing on a FPGA+ CPU Heterogeneous System-On-Chip. Dans Parallel Computing is Everywhere (pp. 677-686). IOS Press.

NVIDIA. (2019). NVIDIA Nsight. Repéré à https://developer.nvidia.com/nsight-visual-studio-edition-2019_3.

O'Loughlin, D., Coffey, A., Callaly, F., Lyons, D., & Morgan, F. (2014). Xilinx vivado high level synthesis: Case studies.

Oliver, N., Sharma, R. R., Chang, S., Chitlur, B., Garcia, E., Grecco, J., Gupta, P. (2011). A Reconfigurable Computing System Based on a Cache-Coherent Fabric. Dans 2011 International Conference on Reconfigurable Computing and FPGAs (pp. 80-85). doi: 10.1109/ReConFig.2011.4.

Paar, C., & Pelzl, J. (2009). Understanding cryptography: a textbook for students and practitioners. Springer Science & Business Media.

Paar, C., & Pelzl, J. (2010). SHA-3 and The Hash Function Keccak. Understanding Cryptography-A Textbook for Students and Practitioners.

Padhi, M., & Chaudhari, R. (2017). An optimized pipelined architecture of SHA-256 hash function. Dans 2017 7th International Symposium on Embedded Computing and System Design (ISED) (pp. 1-4). doi: 10.1109/ISED.2017.8303943.

Pilato, C., & Ferrandi, F. (2013). Bambu: A modular framework for the high level synthesis of memory-intensive applications. Dans 2013 23rd International Conference on Field programmable Logic and Applications (pp. 1-4). doi: 10.1109/FPL.2013.6645550.

Pinilla, J. P. (2016). Source-level instrumentation for in-system debug of high-level synthesis designs for FPGA (University of British Columbia).

Pinilla, J. P., & Wilton, S. J. E. (2016). Enhanced source-level instrumentation for FPGA in-system debug of High-Level Synthesis designs. Dans International Conference on Field-Programmable Technology (FPT) (pp. 109-116). doi:10.1109/FPT.2016.7929514.

Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., & Amarasinghe, S. (2013). Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. Acm Sigplan Notices, 48(6), 519-530. doi: 10.1145/2499370.2462176.

Rajagopalan, V., Boppana, V., Dutta, S., Taylor, B., & Wittig, R. (2011). Xilinx Zynq-7000 EPP: An extensible processing platform family. Dans 2011 IEEE Hot Chips 23 Symposium (HCS) (pp. 1-24). doi: 10.1109/HOTCHIPS.2011.7477495.

Reichenbach, M., Holzinger, P., Häublein, K., Lieske, T., Blinzer, P., & Fey, D. (2019). Heterogeneous Computing Utilizing FPGAs. Journal of Signal Processing Systems, 91(7), 745-757. Repéré à https://doi.org/10.1007/s11265-018-1382-7.

Rivest, R. L. (1990). The MD4 message digest algorithm. Dans Conference on the Theory and Application of Cryptography (pp. 303-311). Springer.

Rogers, P., & Fellow, A. (2013). Heterogeneous system architecture overview. Dans Hot Chips Symposium (pp. 1-41).

Rote, M. D., Vijendran, N., & Selvakumar, D. (2015). High performance SHA-2 core using the Round Pipelined Technique. Dans 2015 IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT) (pp. 1-6). doi: 10.1109/CONECCT.2015.7383912.

Sanaullah, A., & Herbordt, M. C. (2018). Unlocking Performance-Programmability by Penetrating the Intel FPGA OpenCL Toolflow. Dans IEEE High Performance extreme Computing Conference (HPEC) (pp. 1-8). doi: 10.1109/HPEC.2018.8547646.

Sanaullah, A., Patel, R., & Herbordt, M. (2018). An empirically guided optimization framework for FPGA OpenCL. Dans 2018 International Conference on Field-Programmable Technology (FPT) (pp. 46-53). doi: 10.1109/FPT.2018.00018.

Schafer, B. C., & Wang, Z. (2019). High-Level Synthesis Design Space Exploration: Past, Present, and Future. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 39(10), 2628-2639. doi: 10.1109/TCAD.2019.2943570.

Schafer, B. C., & Wang, Z. (2020). High-Level Synthesis Design Space Exploration: Past, Present, and Future. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 39(10), 2628-2639. doi: 10.1109/TCAD.2019.2943570.

Sean, S. (2013). High-performance dynamic programming on fpgas with opencl. Dans Proc. IEEE High Perform. Extreme Comput. Conf.(HPEC) (pp. 1-6).

Sideris, A., Sanida, T., & Dasygenis, M. (2020). High Throughput Pipelined Implementation of the SHA-3 Cryptoprocessor. Dans 2020 32nd International Conference on Microelectronics (ICM) (pp. 1-4). doi: 10.1109/ICM50269.2020.9331803.

Terzo, O., Djemame, K., Scionti, A., & Pezuela, C. (2019). Heterogeneous Computing Architectures: Challenges and Vision. CRC Press.

Theis, T. N., & Wong, H. P. (2017). The End of Moore's Law: A New Beginning for Information Technology. Computing in Science & Engineering, 19(2), 41-50. doi: 10.1109/MCSE.2017.29.

Trickey, H. (1987). Flamel: A high-level hardware compiler. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 6(2), 259-269. doi: 10.1109/TCAD.1987.1270270.

Tuan, H. A., Yamazaki, K., & Oyanagi, S. (2008). Three-stage pipeline implementation for SHA2 using data forwarding. Dans 2008 International Conference on Field Programmable Logic and Applications (pp. 29-34). doi: 10.1109/FPL.2008.4629903.

Vanderbauwhede, W., & Benkrid, K. (2013). High-performance computing using FPGAs. Springer.

Verma, A., Huiyang, Z., Booth, S., King, R., Coole, J., Keep, A., Wu-chun, F. (2017). Developing dynamic profiling and debugging support in OpenCL for FPGAs. Dans 2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC) (pp. 1-6). doi: 10.1145/3061639.3062230.

Waidyasooriya, H. M., Hariyama, M., & Uchiyama, K. (2018). Design of FPGA-based computing systems with OpenCL. Springer. doi: 10.1007/978-3-319-68161-0.

Wang, Z., He, B., Zhang, W., & Jiang, S. (2016). A performance analysis framework for optimizing OpenCL applications on FPGAs. Dans 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA) (pp. 114-125). doi: 10.1109/HPCA.2016.7446058.

Windh, S., Ma, X., Halstead, R. J., Budhkar, P., Luna, Z., Hussaini, O., & Najjar, W. A. (2015). High-Level Language Tools for Reconfigurable Computing. Proceedings of the IEEE, 103(3), 390-408. doi: 10.1109/JPROC.2015.2399275.

Wong, M. M., Haj-Yahya, J., Sau, S., & Chattopadhyay, A. (2018). A New High Throughput and Area Efficient SHA-3 Implementation. Dans 2018 IEEE International Symposium on Circuits and Systems (ISCAS) (pp. 1-5). doi: 10.1109/ISCAS.2018.8351649.

Wong, M. M., Pudi, V., & Chattopadhyay, A. (2018). Lightweight and High Performance SHA-256 using Architectural Folding and 4-2 Adder Compressor. Dans 2018 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC) (pp. 95-100). doi: 10.1109/VLSI-SoC.2018.8644825.

Wu, Q., Ha, Y., Kumar, A., Luo, S., Li, A., & Mohamed, S. (2014). A heterogeneous platform with GPU and FPGA for power efficient high performance computing. Dans 2014 International Symposium on Integrated Circuits (ISIC) (pp. 220-223). IEEE.

Wu, R., Zhang, X., Wang, M., & Wang, L. (2020). A High-Performance Parallel Hardware Architecture of SHA-256 Hash in ASIC. Dans 2020 22nd International Conference on Advanced Communication Technology (ICACT) (pp. 1242-1247). IEEE.

Wyant, C. M., Cullinan, C. R., & Frattesi, T. R. (2012). Computing performance benchmarks among cpu, gpu, and fpga. Computing.

Xilinx. (2012a). AXI Reference Guide UG761. Repéré à https://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug761_axi_reference_guide.pdf.

Xilinx. (2012b). ChipScope Pro Software and Cores: User Guide UG029. Repéré à https://www.xilinx.com/support/documentation/sw_manuals/xilinx14_7/chipscope_pro_sw_cores_ug029.pdf.

Xilinx. (2016). Xilinx SDAccel Development Environment.

Xilinx. (2018a). SDAccel Environment Debugging Guide UG1281. Repéré à https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_3/ug1281-sdaccel-debugging-guide.pdf.

Xilinx. (2018b). Vivado Design Suite User Guide: High-Level Synthesis, UG902 V2017.4.

Xilinx. (2021). Vivado Design Suite User Guide, High-Level Synthesis , UG902 (v2020.1). Repéré à https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_3/ug902-vivado-high-level-synthesis.pdf.

Yang, X.-S. (2011). Metaheuristic optimization. Scholarpedia, 6(8), 11472.

Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., & Stoica, I. (2010). Spark: Cluster computing with working sets. HotCloud, 10(10-10), 95.

Zahran, M. (2017). Heterogeneous computing: Here to stay. Communications of the ACM, 60(3), 42-45. Répéré à https://doi.org/10.1145/3024918.

Zhao, J., Feng, L., Sinha, S., Zhang, W., Liang, Y., & He, B. (2019). Performance Modeling and Directives Optimization for High-Level Synthesis on FPGA. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 39(7), 1428-1441. doi: 10.1109/TCAD.2019.2912916.

Zohouri, H. R. (2018). High performance computing with FPGAs and OpenCL. arXiv preprint arXiv:1810.09773.