HAL
open science

Un processeur SIMT généraliste synthétisable

Caroline Collange

To cite this version:

Caroline Collange. Un processeur SIMT généraliste synthétisable. Compas 2016 - Conférence d'informatique en Parallélisme, Architecture et Système, Jul 2016, Lorient, France. hal-01345070

HAL Id: hal-01345070

https://inria.hal.science/hal-01345070v2

Submitted on 7 Oct 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Compas'2016 : Parallélisme/ Architecture / Système
Lorient, France, du 5 au 8 juillet 2016

Un processeur SIMT généraliste synthétisable

Caroline Collange

Inria Centre de recherche Rennes – Bretagne Atlantique
caroline.collange@inria.fr

Résumé
Nous présentons Simty, un processeur massivement multi-threadé qui assemble dynamiquement des instructions SIMD à partir de code scalaire multi-thread. Il exécute le jeu d'instructions RISC-V (RV32-I). Contrairement aux processeurs SIMD ou SIMT existants tels que les GPU, Simty accepte du code binaire compilé pour des processeurs généralistes sans nécessiter la moindre extension du jeu d'instructions ni modification du compilateur. Le processeur est décrit en RTL synthétisable. Un prototype sur FPGA valide le passage à l'échelle jusqu'à 64 warps ou 64 threads par warp.

Mots-clés : SIMT, SIMD, FPGA, RISC-V

1. Introduction

Le modèle d'exécution SIMT (*Single Instruction, Multiple Threads*), tel qu'employé notamment dans les processeurs graphiques (GPU) de Nvidia, associe un modèle de programmation multi-thread à un mode d'exécution SIMD [17]. Il combine ainsi la simplicité d'un code scalaire du point de vue du programmeur et du compilateur et l'efficacité d'unités d'exécution SIMD au niveau matériel. Cependant, les architectures SIMT actuelles nécessitent des jeux d'instructions spécifiques, et en particulier des instructions de branchement additionnelles nécessitant l'intervention du compilateur. De fait, les GPU SIMT restent incompatibles avec les jeux d'instructions généralistes traditionnels.

Dans cet article, nous présentons un processeur SIMT qui lève cette incompatibilité des jeux d'instructions entre CPU et GPU. En assurant la compatibilité binaire entre CPU et GPU, nous permettons le partage du compilateur, du système d'exploitation et des langages de programmation. Outre la simplification des couches logicielle qu'elle engendre, l'unification des jeux d'instruction facilite la mise au point et le débogage des programmes parallèles.

La démocratisation des solutions de prototypage de matériel tels que les FPGA permet désormais à la communauté académique de construire des prototypes matériels [14, 3, 6]. Nous avons ainsi décrit Simty en VHDL synthétisable et l'avons synthétisé sur FPGA.

Nous présentons notre approche pour généraliser le modèle SIMT section 2, puis décrivons la micro-architecture de Simty section 3, et étudions le cas d'une synthèse sur FPGA section 4.

2. Contexte

2.1. SIMT généraliste

Le modèle d'exécution SIMT consiste à assembler des instructions vectorielles entre différents threads scalaires de programmes SPMD. Les jeux d'instructions SIMT actuels sont essentiel-

lement scalaires, à l'exception des instructions de branchement qui permettent de contrôler la divergence et la reconvergence des threads [4]. De manière équivalente, un tel jeu d'instructions peut être considéré comme entièrement SIMD avec des instructions gather et scatter, des instructions gardées et une gestion des masques assistée en matériel [11].
Nous avons montré qu'il était possible de généraliser le modèle d'exécution SIMT à des jeux d'instructions généralistes, sans instructions de branchement spécifiques [9]. Nous avons proposé une solution dont l'état se limite à un compteur de programme par thread à SympA'2011 [8]. Elle repose sur une détection dynamique de la reconvergence de deux chemins par l'égalité des compteurs de programme et d'une politique de choix du chemin basée sur l'ordre des compteurs de programme et des pointeurs de pile. Nous l'avons généralisé par la suite à l'exécution en parallèle de deux chemins, en proposant une représentation plus efficace des compteurs de programme à base de liste triée [5]. Simty est basé sur cette dernière solution, appliquée à un chemin d'exécution unique.
Les politiques de synchronisation et d'ordonnancement des threads ne garantissent qu'une progression globale : au moins un thread progresse au cours de l'exécution. En revanche, les threads individuels ne bénéficient pas d'une garantie de progression. Les boucles d'attentes actives peuvent donc mener à des interbloquages. Toute synchronisation entre thread devra s'effectuer au travers d'instructions dédiées.

2.2. Terminologie

Un processeur multi-thread ou multi-core présente au système d'exploitation un nombre fixe de threads matériels, sur lequel le système place des threads logiciels. Sauf mention contraire, *thread* désignera par la suite un thread matériel. Dans une architecture SIMT, les threads sont groupés arbitrairement en *warps* de taille fixe. Ce découpage est essentiellement invisible à la partie logicielle.
Le banc de registres et les unités de calcul suivent une organisation SIMD. La largeur SIMD correspond au nombre de threads d'un warp. Chaque thread d'un warp est assigné à une voie SIMD distincte. Son contexte réside entièrement dans un banc de registres associé à sa voie SIMD : les registres de voies différentes ne communiquent pas.
Le modèle de programmation étant multi-thread, chaque thread dispose de son propre compteur de programme (PC) du point de vue du logiciel. Nous introduisons un niveau intermédiaire entre le thread et le warp que nous appellerons *cordée*. Ce concept de cordée se retrouve sous les termes warp-split [16], sub-warp, context [5], fragment ou DV-thread [12] dans la littérature. Les threads d'une cordée sont synchronisés et partagent un PC commun dans le même espace d'adressage, que l'on notera CPC. Tous les threads d'une cordée appartiennent au même warp. Ainsi, chaque warp contient entre 1 et m cordées, avec m le nombre de threads par warp. L'appartenance de threads à une cordée est représentée en matériel par un masque de m bits par cordée. Le bit i du masque de la cordée j est levé lorsque le thread i du warp est inclut dans la cordée j. Il est ainsi possible de représenter l'état d'un warp de manière équivalente soit sous la forme d'un vecteur de m PC, soit sous la forme d'un ensemble de cordées représentées par des couples (CPC, masque).

3. Micro-architecture de Simty

Nous présentons les principes de conception, puis une vue générale du pipeline de Simty, et détaillons les mécanismes de suivi des cordées.

3.1. Principes de conception

L'idée centrale de Simty est de factoriser la logique de contrôle (fetch, décodage, ordonnancement d'instructions) tout en répliquant les chemins de données (registres et unités de calcul). Le pipeline est ainsi construit autour d'un front-end scalaire multi-thread et d'un back-end SIMD.

ISA scalaire généraliste. Nous avons choisi le jeu d'instructions libre RISC-V, sans aucun ajout [19]. Simty supporte actuellement l'ensemble RV32I (instructions généralistes et calcul entier 32 bits) ainsi que les instructions privilégiées gérant les threads matériels en attente d'inclusion dans le standard.

Architecture orientée débit. Simty exploite du parallélisme de threads entre les warps pour masquer les latences d'exécution et le parallélisme de données à l'intérieur des cordées pour augmenter le débit. À contrario, nous ne cherchons pas à tirer parti du parallélisme d'instructions au-delà d'une simple exécution en pipeline pour mieux nous focaliser sur les aspects spécifiques au SIMT. Par exemple, pour simplifier la logique de bypass, un même warp ne peut pas avoir des instructions dans deux étages de pipeline successifs. Cette limitation existe notamment dans le GPU Fermi [17] et le Xeon Phi Knights Corner [7].

Généricité. Le nombre de warps et de threads est configurable lors de la synthèse. L'espace de configurations que nous considérons se situe entre 4 warps $\times$ 4 threads et 16 warps $\times$ 32 threads pour des analogues au cœur Xeon Phi [7] ou au SM de GPU Nvidia [17]. Le code RTL est écrit en VHDL synthétisable. Les étages de pipeline sont séparés en composants distincts pour permettre un ré-équilibrage ou un allongement aisé du pipeline. Toutes les mémoires internes y compris le banc de registres sont construites à base de blocs SRAM à un port de lecture et un port d'écriture, afin de pouvoir être réalisés par des block-RAM sur FPGA ou par des macros génériques en synthèse ASIC.

Arbitrage de ressource par premier-suiveurs. Afin de maximiser le débit multi-thread et simplifier le contrôle, le pipeline est non-bloquant. L'ordonnancement des instructions respecte les dépendances de données ; néanmoins, les autres warps continuent de progresser pendant qu'un warp attend une dépendance. Les accidents tels que les conflits de ressources sont traités par la réexécution partielle de l'instruction fautive.

Lorsqu'une cordée accède à une ressource partagée, l'accès à la ressource est octroyé à un thread arbitraire, le premier de cordée, sans considération pour les autres threads. Les autres threads de la cordée vérifient ensuite si l'accès à la ressource peut satisfaire leur propre besoin (même adresse). Le cas échéant, ils sont considérés comme suiveurs et récupèrent le résultat de l'accès à la ressource. Dans le cas de l'unité de fetch, tous les autres threads de la cordée sont suiveurs. Lorsqu'il reste des threads non suiveurs, la cordée est scindée en deux : le premier et les suiveurs avancent à l'instruction suivante, tandis que les autres threads conservent le même PC pour que leur instruction soit ré-exécutée. Ce mécanisme de ré-exécution partielle garantit la progression (*forward progress*).

Validation basée sur le PC. Le suivi de l'avancement des threads des cordées est basé par leur PC. Le mécanisme de ré-exécution partielle des instructions est interruptible et ne pose pas de contraintes d'atomicité. En effet, l'état architectural individuel de chaque thread reste cohérent, le modèle de programmation ne spécifiant aucun ordre entre les instructions de threads différents.

Ordonnancement min(SP :PC). Afin de favoriser la reconvergence, dans chaque warp, la priorité est donnée à la cordée dont le niveau d'imbrication d'appel de fonction est maximal, et en cas d'égalité à celle dont le PC est minimal [4].

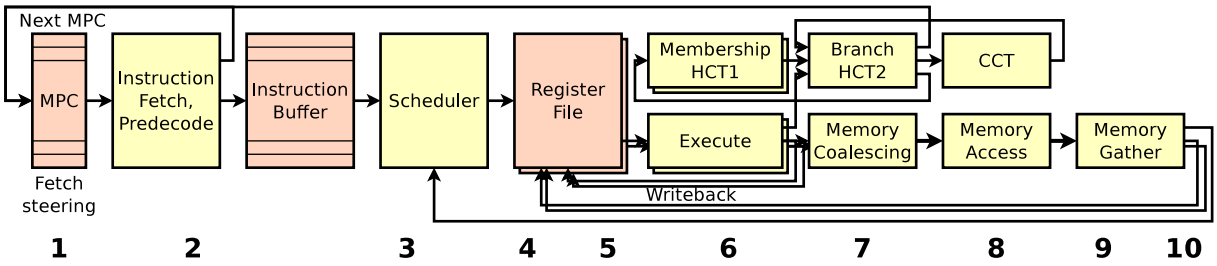


FIGURE 1 – Pipeline Simty.

3.2. Pipeline

Simty est construit autour d'un pipeline de 10 étages à exécution dans l'ordre traitant une instruction par cycle représenté sur la figure 1. Nous décrivons brièvement chaque étage.

Fetch steering sélectionne un warp et lit le CPC spéculatif de sa cordée courante dans une table indexée par le numéro de warp. Le choix du warp est actuellement basé sur un ordonnancement en portillon (*round-robin*). Le CPC spéculatif de la table est mis à jour vers l'adresse prédite de l'instruction suivante. Actuellement, la prédiction consiste simplement à incrémenter le CPC. Il est néanmoins possible d'employer un prédicteur de branchement.

Fetch/predecode récupère un mot d'instruction depuis le cache ou la mémoire d'instructions. Une première étape de pré-décodage vérifie pour chaque opérande s'il provient d'un registre. Le codage des instructions RISC-V qui représente chaque opérande à un emplacement fixe dans le mot d'instruction permet de se contenter de ce pré-décodage pour reconnaître les dépendances entre instructions. L'instruction prédécodée est placée dans un tampon d'instructions contenant une entrée par warp.

Schedule lance les instructions dont les opérandes sont prêts. Une étape de qualification sélectionne les warps dont la prochaine instruction est exécutable. Elle prend en compte les dépendances entre instructions et la disponibilité des ressources d'exécution : les conflits de bancs dans les registres sont évités a priori. Une étape de sélection choisit un des warps exécutables. La politique d'ordonnancement est actuellement en portillon : le premier warp prêt en comptant à partir du warp prioritaire est sélectionné. Pour une meilleure distribution dans le temps des accès mémoire, il est possible d'employer un ordonnancement aléatoire.

Collection des opérandes consiste en deux étages. Chaque voie SIMD possède son propre banc de registres. Chaque banc de registres est divisé en deux bancs, l'un contenant les données des warps d'identifiant pair, l'autre celles des warps d'identifiant impaire. Chaque banc dispose d'un unique port de lecture et d'un port d'écriture. Pour les instructions ayant deux opérandes source, deux ports de lecture sont émulés par des accès alternés sur deux cycles. L'instruction suivante doit alors appartenir à un warp ayant ses données dans l'autre banc. Lorsqu'une instruction n'a qu'un seul opérande lu depuis un registre, les deux bancs sont disponibles pour l'instruction suivante, qui peut alors appartenir indifféremment à un warp pair ou impair. Le port d'écriture de chaque banc est disponible pour l'écriture des résultats depuis les unités arithmétiques. Les valeurs retournées depuis la mémoire sont écrites par vol de cycle en sortie d'une FIFO. Les instructions sont aussi décodées totalement à cette étape.

Exécution. Les unités d'exécution consistent en une ALU 32-bit supportant le jeu d'instructions RV32I dans chaque voie. Les instructions arithmétiques sont actuellement toutes effectuées en un cycle. Par la suite, nous prévoyons d'intégrer des unités virgule flottante pipelinées.

Appartenance. En parallèle de l'exécution, le masque d'appartenance des threads à la cordée

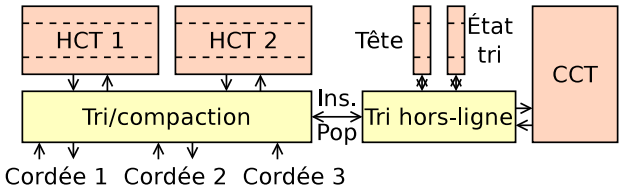


FIGURE 2 – Unité de suivi des cordées

est reconstitué à partir du PC. Pour les instructions arithmétiques, il s'agit du masque de validité définitif, non spéculatif. Pour les instructions d'accès mémoire, il s'agit du masque spéculatif avant détection des exceptions. Le *premier* thread de la cordée est calculé par une recherche du premier bit levé dans le masque. L'unité d'appartenance détecte également la reconvergence entre cordées. Elle s'appuie sur trois tables nommées HCT1, HCT2 et CCT disposées sur trois étages successifs du pipeline. Nous décrivons cette unité plus en détail section 3.3.

Writeback. Les résultats de l'étage d'exécution sont écrits dans les bancs de registres de manière conditionnelle. Pour la voie i, l'écriture est effectuée lorsque le bit i du masque de validité est levé. Le masque peut être entièrement nul dans le cas d'une mauvaise prédiction de branchement. Le résultat est également injecté dans le réseau de *bypass* de chaque voie lorsque le bit correspondant du masque est levé. La garde du *bypass* est nécessaire pour qu'une instruction précédant juste un point de reconvergence ne contamine pas l'instruction suivante avec des données incorrectes.

Branchements L'unité de branchement est chargée de construire les cordées après une instruction de branchement. Elle reçoit un vecteur de conditions c depuis les ALU et le masque de validité m depuis l'unité d'appartenance. Pour un saut conditionnel relatif, l'adresse destination scalaire PC_d est calculée à partir du PC et de l'immédiat. Deux cordées sont formées, une pour les threads qui suivent le branchement $(PC_d, c \wedge m)$ et une pour ceux qui ne le suivent pas $(PC + 4, \bar{c} \wedge m)$. Dans le cas d'un saut indirect dont le vecteur des adresses calculées contient plusieurs destinations différentes, le nombre de cordées peut atteindre le nombre de threads. Nous sérialisons les sauts vers chaque destination unique afin de limiter à deux le nombre de cordées en sortie de l'unité de branchement. Les threads ayant la même destination PC_l que le premier de cordée progressent, tandis que les autres conservent le même PC afin que l'instruction de branchement indirect soit ré-exécutée.

Arbitre mémoire et *Coalescing* L'arbitre mémoire coordonne les accès concurrents à une mémoire commune entre les threads d'une cordée. Il est optimisé pour deux cas communs : lorsque les threads du warp accèdent à des mots successifs dans la même ligne de cache, et lorsque les threads accèdent au même mot mémoire. Ces deux cas sont détectés par des comparaisons de l'adresse de chaque thread avec l'adresse du *premier*. Les threads n'étant pas dans un de ces cas forment une nouvelle cordée pour que leur instruction soit ré-exécutée.

Assemblage (*gather*) distribue les données de la ligne de cache lue depuis la mémoire au threads de la cordée. En particulier, elle effectue la diffusion (*broadcast*) du mot lu par le premier.

3.3. Suivi des cordées

L'unité de suivi des cordées assure trois fonctions : 1. valider l'appartenance des threads à une cordée pour calculer le masque de validité d'une instruction, 2. fusionner les cordées dont le PC est identique pour assurer la reconvergence des threads, et 3. sélectionner la cordée prioritaire qui sera parcourue par le *front-end*.

Un moyen conceptuellement simple de gérer les cordées est d'allouer un registre PC individuel

pour chaque thread comme dans un processeur multi-thread conventionnel. On constitue alors les cordées (fonction 1 et 2) dans chaque warp en comparant les PCentre eux à chaque cycle, et on sélectionne la cordée prioritaire (fonction 3) en appliquant une politique d'arbitrage entre tous les threads d'un warp [8]. Cependant, une approche plus efficace mais fonctionnellement équivalente consiste à adopter la représentation alternative évoquée Section 2.2 basée sur un ensemble de couples (CPC, masque). La fonction 1 consiste simplement à vérifier que l'adresse de l'instruction à valider correspond au CPC de la cordée courante et lire le masque correspondant. Les fonctions 2 et 3 se basent sur une liste des cordée triée par priorité. La priorité se basant sur la valeur du CPC, il n'est nécessaire de vérifier la reconvergence qu'entre la cordée courante et la seconde cordée par ordre de priorité.

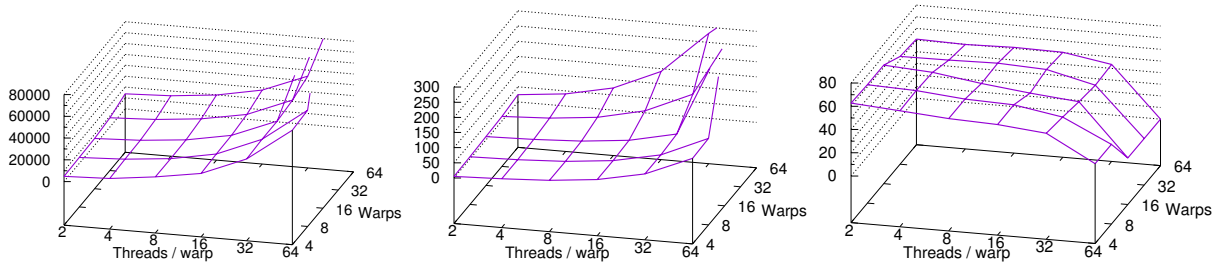
Les cordées sont maintenues triées par niveau d'appel de fonction et PC. Pour une gestion efficace d'un grand nombre de cordées en matériel, nous séparons les deux cordées *actives* de tête, triées en continu, des autres cordées *inactives*, triées hors-ligne par une machine à états. Les cordées actives sont conservées dans les HCT (*hot context table*) indexées par l'identifiant du warp, tandis que la CCT (*cold context table*) maintient les cordées inactives (figure 2) [5]. La CCT est dimensionnée pour le pire cas d'une cordée par thread. Elle contient une pile de cordées pour chaque warp, dont la tête est suivie par un pointeur. Le tri hors-ligne est une machine à état conservant 3 états et un pointeur par warp qui effectue progressivement un tri par insertion lorsque les ports de la CCT sont disponibles. Le pointeur parcourt les entrées de la CCT et l'entrée correspondante est comparée avec l'entrée de la HCT 2. Lorsque l'ordre n'est pas respecté, ces entrées sont échangées.

4. Réalisation sur FPGA

La synthèse du circuit sur FPGA est une première étape de prototypage vers la synthèse matérielle. Cela représente aussi une application en tant que telle : Simty peut servir de contrôleur parallèle pour un accélérateur reconfigurable, comme alternative à un processeur vectoriel *soft-core* [18]. Nous avons synthétisé et testé Simty sur un FPGA Altera Cyclone IV EP4CE115 d'une carte de développement Altera DE-2 115, avec une fréquence cible de 50 MHz.

Les paramètres micro-architecturaux principaux sont le nombre de warps n et la largeur d'un warp m. La largeur d'un warp détermine le nombre d'unités de calcul et par conséquent le débit d'exécution d'un cœur. Une architecture Simty à peu de cœurs comportant des warps larges profite aux applications dont les threads ont un comportement homogène et qui profitent de la vectorisation dynamique. À l'inverse, davantage de cœurs dotés de warps plus étroits offriront de meilleures performances sur du code parallèle irrégulier. Le nombre de warps détermine la tolérance aux latences d'exécution, et notamment celle de la mémoire externe. Le nombre total de threads par cœur nécessaires est proportionnel au produit latence $\times$ débit mémoire selon la loi de Little [15]. Dans le cas de la carte de développement DE2-115 dont la SDRAM PC133 offre un débit crête de 533 Mo/s et une latence (sur échec de page) de 75 ns, 10 threads suffisent à masquer la latence mémoire. Il s'agit cependant d'une technologie mémoire ancienne : les mémoires modernes ont des produits latence $\times$ débit très supérieurs. Dans un processeur *many-core*, il faudra également prendre en compte le temps de traversée des caches, du réseau sur puce et du contrôleur mémoire.

La figure 3 présente les résultats de synthèse après placement et routage en fonction du nombre de threads et de warps. L'architecture passe à l'échelle jusqu'à 64 warps $\times$ 32 threads et 8 warps $\times$ 64 threads, et la logique de contrôle est amortie par les unités d'exécution SIMD. Les configurations les plus avantageuses se situent entre 8 warps $\times$ 8 threads et 32 warps $\times$ 16 threads sur cette plate-forme : au-delà, la congestion du routage augmente le temps de cycle et



(a) Surface (LE) (b) Surface (M9K) (c) Frequence (MHz)

FIGURE 3 – Passage à l'échelle de Simty sur Altera Cyclone IV en fonction du nombre de threads par warp m et du nombre de warps n. La fréquence est l'estimation pire cas à 85°C, en MHz. La surface est donnée en éléments logiques (LE) et en blocs de RAM de 128 × 32-bit (M9K). Le placement-routage échoue sur les configurations {16, 32, 64} warps × 64 threads.

le gain en surface par rapport à plusieurs cœurs Simty est faible.

5. Travaux liés

Plusieurs processeurs parallèles synthétisables libres ont été récemment diffusés dans la communauté académique. Parmi les processeurs vectoriels, HWACHA est un processeur vectoriel synthétisable dont le jeu d'instructions est une extension vectorielle de RISC-V [14], tandis que VectorBlox MXP est un *soft-core* vectoriel pour FPGA [18]. Kingyens et Steffan proposent un processeur compatible avec l'unité de *fragment shader* de l'architecture GPU ATI R500 [13]. MIAOW est un GPU synthétisable libre basé sur l'architecture AMD GCN [3]. Flexgrip [2] est un GPU basé sur l'architecture Nvidia Tesla qui reprend le pipeline du simulateur Barra [10]. Guppy est un processeur basé sur Leon qui exécute un jeu d'instruction SIMD basé sur SPARC avec prédication simple (limité à la *if-conversion*) [1]. Nyuzi (ex-Nyami) est un processeur SIMD multi-thread destiné au rendu graphique [6]. Ces processeurs sont tous basés soit sur des jeux d'instructions vectoriels ou SIMD avec prédication, soit sur des jeux d'instructions SIMT avec des instructions de branchement spécifiques. Simty est le premier processeur SIMT basé sur un jeu d'instructions scalaire généraliste.

6. Conclusion

Simty démontre la faisabilité matérielle d'une architecture SIMT à jeu d'instructions généraliste. Il fournit une brique de base pour des processeurs *many-cores*. Après cette preuve de concept, nous comptons ajouter les instructions virgule flottante, atomiques et privilégiées, la mémoire virtuelle, ainsi que mettre en place l'infrastructure logicielle. L'emploi du jeu d'instruction RISC-V simplifie cette étape en permettant de s'appuyer sur les outils existants. Simty est distribué sous licence CeCILL à l'adresse `https://gforge.inria.fr/projects/simty`.

Bibliographie

1. Al-Dujaili (A.), Deragisch (F.), Hagiescu (A.) et Wong (W.-F.). – Guppy : A GPU-like soft-core processor. – In *Field-Programmable Technology (FPT), 2012 International Conference on*, pp. 57–60. IEEE, 2012.
2. Andryc (K.), Merchant (M.) et Tessier (R.). – FlexGrip : A soft GPGPU for FPGAs. – In *Field-Programmable Technology (FPT), 2013 International Conference on*, pp. 230–237. IEEE, 2013.

3. Balasubramanian (R.), Gangadhar (V.), Guo (Z.), Ho (C.-H.), Joseph (C.), Menon (J.), Drumond (M. P.), Paul (R.), Prasad (S.), Valathol (P.) et al. – Enabling GPGPU low-level hardware explorations with MIAOW : an open-source RTL implementation of a GPGPU. *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 12, n2, 2015, p. 21.
4. Brunie (N.) et Collange (S.). – Reconvergence de contrôle implicite pour les architectures SIMT. *Revue des Sciences et Technologies de l'Information - Série TSI : Technique et Science Informatiques*, vol. 32, n2, février 2013, pp. 153–178.
5. Brunie (N.), Collange (S.) et Diamos (G.). – Simultaneous Branch and Warp Interweaving for Sustained GPU Performance. – In *39th Annual International Symposium on Computer Architecture (ISCA)*, pp. 49 – 60, Portland, OR, United States, 2012.
6. Bush (J.), Dexter (P.), Miller (T. N.) et Carpenter (A.). – Nyami : a synthesizable GPU architectural model for general-purpose and graphics-specific workloads. – In *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on*, pp. 173–182. IEEE, 2015.
7. Chrysos (G.). – Intel® Xeon Phi™ coprocessor-the architecture. *Intel Whitepaper*, 2014.
8. Collange (S.). – Une architecture unifiée pour traiter la divergence de contrôle et la divergence mémoire en SIMT. – In *SYMPosium en Architectures*, p. 02, Saint-Malo, France, mai 2011.
9. Collange (S.), Daumas (M.), Defour (D.) et Parello (D.). – Étude comparée et simulation d'algorithmes de branchements pour le GPGPU. – In *SYMPosium en Architectures nouvelles de machines (SYMPA)*, 2009.
10. Collange (S.), Daumas (M.), Defour (D.) et Parello (D.). – Barra : a parallel functional simulator for GPGPU. – In *IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pp. 351–360, 2010.
11. Hennessy (J. L.) et Patterson (D. A.). – *Computer architecture : a quantitative approach*. – Elsevier, 2011.
12. Kalathingal (S.), Collange (S.), Narasimha Swamy (B.) et Seznec (A.). – *Transforming TLP into DLP with the Dynamic Inter-Thread Vectorization Architecture*. – Research Report nRR-8830, Inria Rennes Bretagne Atlantique, décembre 2015.
13. Kingyens (J.) et Steffan (J. G.). – A GPU-inspired soft processor for high-throughput acceleration. – In *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pp. 1–8. IEEE, 2010.
14. Lee (Y.), Waterman (A.), Avizienis (R.), Cook (H.), Sun (C.), Stojanovic (V.) et Asanović (K.). – A 45nm 1.3 GHz 16.7 double-precision GFLOPS/W RISC-V processor with vector accelerators. – In *European Solid State Circuits Conference (ESSCIRC), ESSCIRC 2014-40th*, pp. 199–202. IEEE, 2014.
15. Little (J. D. C.). – A proof for the queuing formula : $L = \lambda W$. *Operations Research*, vol. 9, n3, 1961, pp. 383–387.
16. Meng (J.), Tarjan (D.) et Skadron (K.). – Dynamic warp subdivision for integrated branch and memory divergence tolerance. *SIGARCH Comput. Archit. News*, vol. 38, n3, 2010, pp. 235–246.
17. Nickolls (J.) et Dally (W. J.). – The GPU computing era. *IEEE Micro*, vol. 30, March 2010, pp. 56–69.
18. Severance (A.), Edwards (J.), Omidian (H.) et Lemieux (G.). – Soft vector processors with streaming pipelines. – In *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays*, pp. 117–126. ACM, 2014.
19. Waterman (A.), Lee (Y.), Patterson (D. A.) et Asanović (K.). – *The RISC-V Instruction Set Manual. Volume 1 : User-Level ISA, Version 2.0*. – Rapport technique, DTIC Document, 2014.