

Exploration architecturale de l'accumulateur de Kulisch

Yohann Uguen, Florent de Dinechin

▶ **To cite this version:**

Yohann Uguen, Florent de Dinechin. Exploration architecturale de l'accumulateur de Kulisch. Compas'2017 - Conférence d'informatique en Parallélisme, Architecture et Système, Jun 2017, Sophia Antipolis, France. pp.1-8. hal-02131977

HAL Id: hal-02131977
https://inria.hal.science/hal-02131977

Submitted on 16 May 2019

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'2017 : Parallélisme/ Architecture / Système
Sophia Antipolis, France, du 27 au 30 juin 2017

Exploration architecturale de l'accumulateur de Kulisch

Yohann Uguen et Florent de Dinechin

Univ Lyon, INSA Lyon, Inria, CITI
F-69621 Villeurbanne, France
prénom.nom-composé@insa-lyon.fr

Résumé
Les sommes de produit utilisant le format flottant accumulent des erreurs d'arrondi pouvant altérer la précision du résultat. Face à ce constat, Kulisch a proposé d'utiliser un accumulateur interne suffisamment grand pour couvrir l'éventail d'exposants flottants, ce qui permet de ne jamais arrondir les additions. Cette architecture n'a jamais aboutie dans les processeurs grand public car elle était considérée comme trop lente et/ou utilisant trop de ressources. Cependant, elle peut être intéressante dans le cadre des FPGAs pour deux raisons. D'une part, on peut y utiliser un format flottant non standard, plus petit que 32 bits. La faible précision du format peut être compensée par l'exactitude de l'accumulation dans une architecture dont la taille devient raisonnable. D'autre part, l'addition de nombres flottants dans un tel accumulateur est associative. Dans un flot de synthèse de haut niveau, ceci permet des optimisations qui sont interdites en flottant standard. Ce travail compare donc plusieurs implémentations de l'accumulateur de Kulisch, dont deux sont originales. Ces architectures sont implémentées dans un générateur de C++ entièrement configurable produisant du code compatible avec VivadoHLS. Les comparaisons effectuées sur FPGAs Xilinx Kintex 7 montrent une amélioration par rapport à la solution de Kulisch en termes de surface et de rapidité. De plus, la comparaison avec des implémentations flottantes classiques montre des compromis intéressants.

Mots-clés : Arithmétique, Virgule flottante, Synthèse de haut niveau

---

1. Introduction

Les circuits logiques programmables (FPGAs) ont démontré leur potentiel en tant qu'accélérateurs pour de nombreuses applications. Les FPGAs peuvent s'abstraire de certaines limitations des processeurs classiques, par exemples dues au coût du décodage d'instructions, à la contention sur la file de registres, au parallélisme d'instruction limité, aux formats de données fixés, etc.
Cette liberté de conception entraine cependant une plus grande complexité de programmation. Pour y remédier, les outils de synthèse de haut niveau (HLS) transforment en circuits des programmes écrits dans un langage de programmation dérivé du C. Mais le C vient lui-même avec de nombreuses restrictions, en particulier pour le calcul sur les nombres réels : les types de données sont alors les quelques formats flottants définis par la norme IEEE-754 [1], et la sémantique d'exécution est définie par la norme C11.
Selon cette sémantique, l'addition flottante n'est pas associative, ce qui laisse au programmeur deux choix : soit conserver la sémantique séquentielle du programme, ce qui va empêcher de nombreuses optimisations de compilation des outils HLS ; soit permettre ces optimisations mais risquer d'obtenir un résultat différent dû à des erreurs d'arrondis différentes.

Cet article propose de sortir de ce dilemne par le haut dans un cas particulier important : les sommes de produits, qui sont au cœur de l'algèbre linéaire, et qui peuvent devenir très imprécises lorsque la taille des vecteurs augmente [9].
De précédents travaux tentent d'améliorer le problème de parallélisme des additionneurs flottants standards [14, 7, 16] ou modifiés [12, 8, 11]. Une variante de l'additionneur flottant qui calcule l'erreur d'arrondi permet de régler les problèmes de précision, mais le nombre de cycles nécessaires à l'accumulation est non déterministe [10, 3]. Un troisième axe de recherche, que nous suivons ici, est l'utilisation de formats internes différents des formats flottants standards [2, 15, 13, 6].
Ce travail revisite l'accumulateur exact proposé par Kulisch [4]. Son idée générale, détaillée dans la partie 2, est d'enlever des sommes de produits toutes sources d'erreurs d'arrondi. Plusieurs implémentations de cette idée sont possibles, chacune avec de nombreux paramètres. La partie 3 présente une nouvelle technique pour effectuer des accumulations, améliorant la vitesse et la surface de l'architecture de Kulisch. La partie 4 introduit une architecture pipelinée. La partie 5 compare ces implémentations dans un générateur paramétrable.

2. Présentation de l'accumulateur de Kulisch

L'idée principale de Kulisch est de retirer toute source d'erreur d'arrondi dans une somme de produits. Construire un multiplieur exact est en réalité plus simple que de construire un multiplieur classique. En effet, comme présenté dans la figure 1 (gauche), il s'agit d'un multiplieur classique duquel l'arrondi et la normalisation ont été retirés. Ensuite, l'additionneur flottant est remplacé par un accumulateur en virgule fixe très large dont les bits couvrent l'éventail des exposants d'un produit flottant.
Kulisch à même proposé [4] d'ajouter suffisamment de bits pour absorber des dépassements de formats temporaires. Son architecture possède ainsi 4288 bits en précision flottante double.
Dans le reste de cet article, la taille de l'accumulateur est notée w_a. Comme montré ci-dessus, la taille minimum de w_a peut être déduite de la taille de l'exposant w_e et de la mantisse w_f du format d'entrée. Par exemple, pour un format de 16 bits d'entrée, le w_a minimum sera de 80, pour 32 bits il faudra au moins $w_a = 554$ et pour 64 bits en entrée, 4196 bits sont nécessaires comme w_a minimum.
Dans son livre, Kulisch suggère trois façons d'implémenter son accumulateur [4]. La première, détaillée en 2.1, utilise un additionneur long ainsi qu'un décaleur long. La seconde, revue en 2.2, utilise un petit additionneur et de la RAM afin de contenir l'accumulateur de manière segmentée. Enfin, la troisième utilise des petits additionneurs parallèles et est décrite en 2.3.
Dans la suite de cet article, les termes "exposant" de taille w_e et "mantisse" de taille w_f sont utilisés pour décrire les sorties du multiplieur.

2.1. Additionneur long et décaleur long

Dans cette approche, l'architecture contient un registre de w_a bits et un additionneur de w_a bits. Chaque entrée est décalée au bon endroit en fonction de son exposant, puis ajoutée à l'accumulateur long.
Cette architecture n'est praticable que pour de petits formats pour deux raisons. Premièrement, elle nécessite un décaleur très long. Ensuite, elle nécessite une addition de w_a bits à chaque cycle qui sera soit lente à cause de la propagation de retenue, soit chère si un additionneur rapide est utilisé. Cependant, pour de très petits formats, cette architecture peut être le bon choix.

Toutes les architectures présentées dans la suite se basent sur le fait d'éclater l'accumulateur

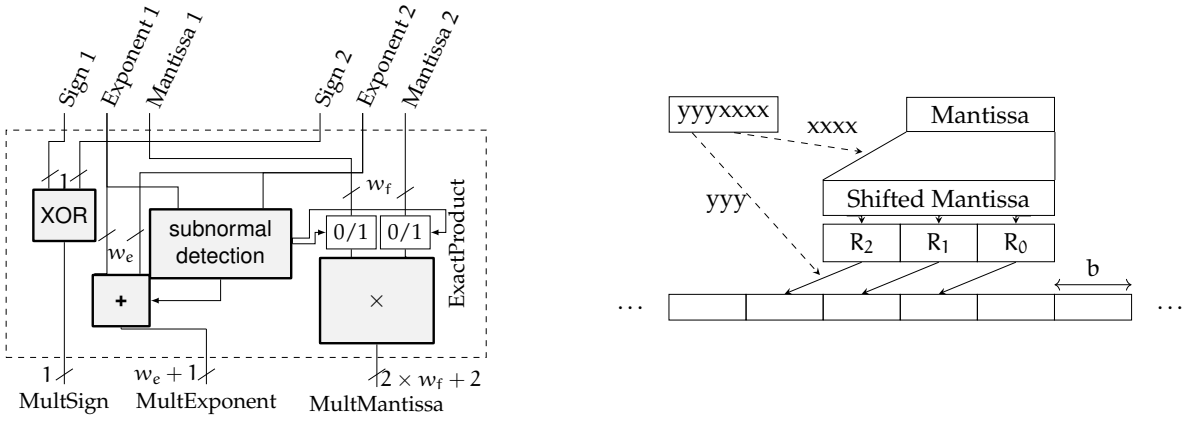


FIGURE 1 – Architecture du multiplicateur flottant (gauche) et du décalage d'une mantisse dans trois mots de l'accumulateur long (ici $S = 3$)

en de petits mots de b bits. Il y a donc $N = \lceil w_a/b \rceil$ mots. Cela décompose le décaleur en deux opérations : sélectionner le mot auquel envoyer la mantisse, puis décaler au sein d'un mot. Le décalage au sein d'un mot est obtenu en prenant les k bits de poids faible de l'exposant alors que le mot adressé A est obtenu grâce aux $w_e - k$ bits de poids fort.

La mantisse décalée sera étalée sur plusieurs mots : au moins deux, mais potentiellement plus. En effet, après un décalage de taille maximum $b - 1$, la mantisse décalée est de taille $w_f - 1 + b$. On peut alors calculer $S = \left\lceil \frac{w_f - 1 + b}{b} \right\rceil$, où S représente le nombre de mots sur lequel la mantisse décalée s'étale. Ces mécanismes sont illustrés sur la figure 1 (droite).

Un tel sectionnement offre deux avantages ; les deux étapes du décalage peuvent être effectuées en parallèle ; ajouter une valeur ne demande que d'additionner S mots (Figure 1 à droite).

Cependant il y a deux problèmes à résoudre. Le premier est la *propagation de retenue* d'un mot de l'accumulateur à l'autre. Cela nécessite de potentiellement mettre à jour tous les mots "au dessus" du mot ayant généré une retenue. Si cela est fait naïvement, cela nécessitera un long délai ou un grand nombre de cycles.

Le second est la *gestion du signe* : les mantisses d'entrée peuvent être signées, elles ont donc besoin d'être ajoutées ou soustraites de l'accumulateur. Dans ce cas, la propagation de *retenue de soustraction (borrow)* est aussi un problème.

2.2. Accumulateur éclaté en RAM

Dans cette variante, les mots de l'accumulateur long sont stockés sur puce dans une mémoire adressable (RAM).

Afin de s'occuper du problème de propagation de retenue, Kulisch utilise un tableau contenant l'état de chaque mot de l'accumulateur. L'état peut être parmi les trois suivants : le mot contient uniquement des 1 (état 1), des 0 (état 0) ou un mélange des deux (état NA). Quand une mantisse est ajoutée à un mot M, l'architecture charge aussi depuis la mémoire le mot M' tel que $M' > M$ tel que $!\exists M'', (M < M'' < M') \wedge \text{état}(M'' \neq 1)$.

Si une retenue est produite, c'est le mot M' qui la recevra. Dans ce cas, l'état de tous les mots entre M et M' passe de 1 à 0. Les retenues de soustraction sont gérées de la même manière.

Si les mots en RAM ont une taille égale à $S \times b$, au moins deux mots mémoire doivent être chargés pour chaque accumulation, en plus du mot mémoire chargé pour la propagation de retenue. Cela requiert une mémoire RAM à trois ports, ce qui représente un certain coût en ressources.

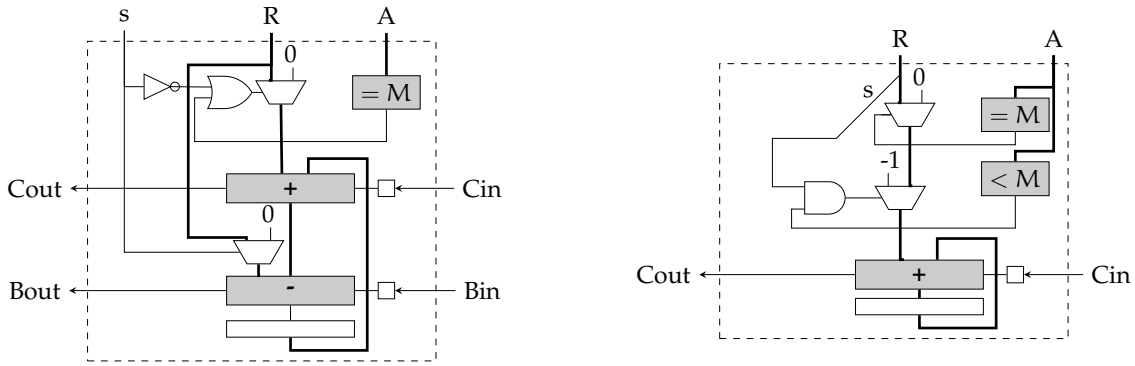


FIGURE 2 – Architecture du SA de l'accumulateur éclaté en registres, avec *carry* et *borrow* (gauche) et en complément à deux (droite)

La limitation principale de cette architecture est qu'elle est difficile à pipeliner. En effet, une nouvelle accumulation ne peut débuter avant que l'écriture mémoire précédente ne soit terminée.

2.3. Accumulateur éclaté en registres avec *carry* et *borrow*

Cette architecture instancie un sous-additionneur par mot de l'accumulateur. Ces N sous-additionneurs (SA) peuvent travailler en parallèle. Chaque SA est associé à deux registres, contenants la valeur des retenues d'addition et de soustraction, qui seront transmises au prochain SA au cycle suivant. Ainsi, la propagation de retenue est reportée. Les retenues sont en partie propagées à chaque cycle, mais il faut N cycles à la fin du calcul d'accumulation (en ajoutant des zéros) afin de terminer la propagation de retenue.
Ces N cycles à ajouter au calcul ne sont pas un problème si le nombre de données à ajouter est bien plus grand que N. Une alternative plus coûteuse serait d'utiliser un tableau d'états comme discuté en 2.2. Ceci n'a pas été exploré dans le cadre de cette étude.
En pratique chaque partie de mantisse, annotée avec une adresse de destination A, est envoyée sur le bus de l'accumulateur. Chaque SA écoute ce bus, puis les SAs effectuent leurs sommes en parallèle.
Cette architecture permet une accumulation pipelinée effectuant une itération par cycle. Dans la version de Kulisch, décrite en figure 2 (gauche), chaque SA accumule une mantisse, une retenue et une retenue de soustraction. Ainsi, chaque SA doit effectuer une addition et une soustraction en un cycle.

3. Accumulateur éclaté en registres en complément à deux

Nous proposons dans cet article une amélioration de l'accumulateur éclaté en registres décrit dans la partie 2.3. À la place d'envoyer une mantisse accompagnée de son signe, l'idée est d'utiliser une représentation en complément à deux. De cette manière les bits de retenue de soustraction disparaissent. Cela simplifie l'architecture des SAs n'ayant maintenant plus de soustracteurs (voir figure de droite de 2). Cette approche déplace le problème de la gestion du signe depuis les SAs (répliqués N fois) vers un simple composant, placé juste après la multiplication, inversant bit à bit la mantisse si le signe est négatif.
En complément à deux, un nombre négatif voit son signe étendu jusqu'au bit de poids fort de l'accumulateur. Ainsi, si une mantisse négative est envoyée sur le bus, les S SAs correspondants vont faire l'addition, et tout les SAs de poids plus significatifs vont ajouter -1 (en complément à deux, le mot dont tous les bits sont à 1) à leur mot. Ceci effectue alors l'extension de signe de la mantisse en complément à deux sur tout l'accumulateur.

On effectue alors un comptage des bits de poids fort au lieu d'un comptage des zéros pour récupérer l'exposant du résultat. Puis, la mantisse récupérée doit être convertie afin d'obtenir le résultat final si son signe est négatif.

4. Accumulateur éclaté pipeliné

Cette architecture propose de pipeliner le bus de l'accumulateur éclaté en registres. Chaque SA est associé à un étage de pipeline. Le format utilisé est le complément à deux.
La figure 3 donne un aperçu de cette architecture. Dans cette figure, les étages de pipeline sont séparés par des lignes verticales. Les lignes horizontales séparent les termes à ajouter et le résultat de la somme. Les flèches en pointillé représentent la propagation de donnée entre les étages du pipeline à chaque cycle d'horloge.
Les S mots de la mantisse décalée sont envoyés aux étages spéciaux à droite (numérotés de $-S+1$ à 0). On transmet aussi au pipeline le signe et l'étage auquel les mantisses doivent être ajoutées.

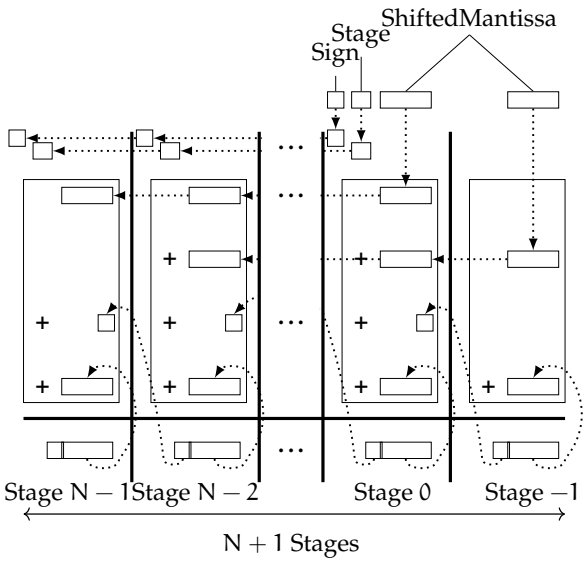


FIGURE 3 – Accumulateur pipeliné pour $S = 2$

Les étages dont le numéro est entre 0 et $N - S$ sont identiques. Ils additionnent S mantisses dans leur SA plus la retenue de l'étage précédent. Cette retenue n'est donc pas réduite à un seul bit.
De chaque coté du pipeline, il y a $S - 1$ étages qui ne sont pas identiques. Ils calculent la somme d'un nombre de termes moins important.
Dans les étages centraux, les mots de mantisse sont ajoutés si leur étage de destination est l'étage actuel, sinon des 0 sont ajoutés. Une fois ajoutées, les mantisses sont remplacées par des 0. Si l'étage courant est celui dans lequel la mantisse de tête doit être ajoutée et qu'elle est négative, après l'addition, la mantisse sera remplacée par -1 et transmise à l'étage suivant. De cette manière, elle sera ajoutée à tous les étages supérieurs et effectuera l'extension de signe.
Afin de récupérer le résultat final, on utilise la même méthode que dans la partie précédente.

5. Evaluation des accumulateurs présentés

Les architectures présentées dans cet article sont entièrement configurables en termes de tailles d'entrée/sortie ainsi qu'au niveau de la taille des mots des SAs. Nous avons écrit un générateur pour ces architectures qui produit du code C++ compatible avec VivadoHLS.
Tout les résultats de synthèse présentés ici ont été obtenus avec la suite Vivado 2016.3 après placement et routage ayant pour cible un FPGA Kintex 7. Les programmes VivadoHLS calculent des accumulations de 1000 termes et convertissent leurs résultats finaux au format flottant.
Dans un but de complétude de cette comparaison, nous avons aussi implémenté les deux accumulateurs de Kulisch originaux des parties 2.1 et 2.2. Les résultats de synthèse de ces architectures présents dans la littérature visaient des technologies VLSI anciennes et n'étaient donc pas comparables avec nos résultats. Les résultats de nos synthèses sont renseignés dans le tableau 1.
Comme attendu, les versions avec RAM utilisent très peu de ressources mais souffrent d'une

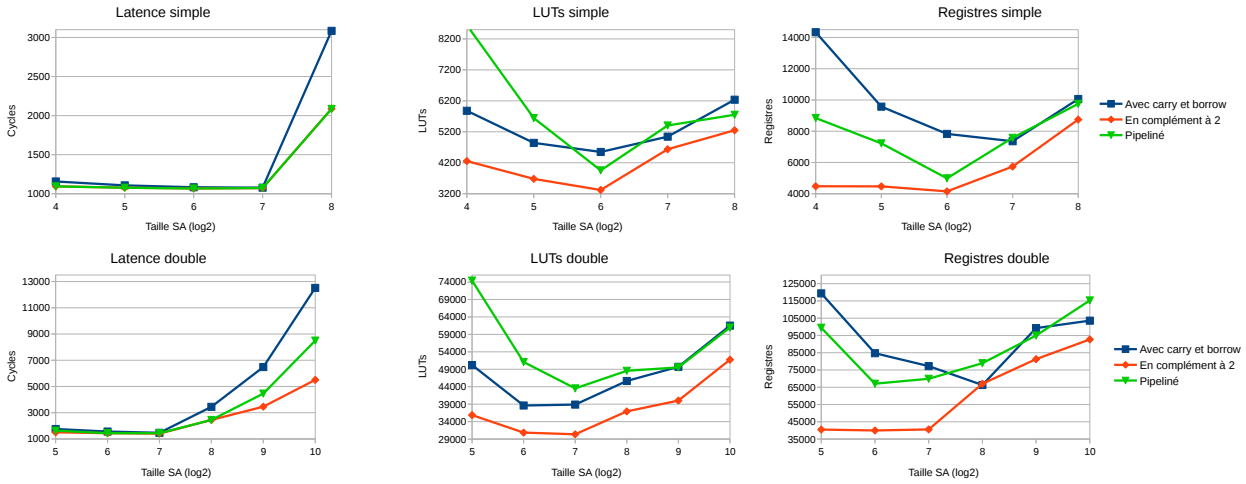


FIGURE 4 – Latence et surface (pour une fréquence de 100MHz) de trois accumulateurs éclatés (en registre avec *carry* et *borrow*, en complément à deux, et pipeliné) pour différentes tailles de SA (données en échelle logarithmique), pour des flottants en précision simple et double

longue latence due aux accès mémoire. De plus, l'architecture utilisant un long additionneur possède une latence plus courte mais utilise bien plus de ressources.
Pour l'ensemble des accumulateurs éclatés, la première question que nous nous sommes posé était de déterminer leur taille de mot b optimale. Cette étude est résumée dans la figure 4. La taille optimale observée est de 64 bits dans le plupart des cas. Des résultats plus détaillés sont présentés dans le tableau 1 indiquant les fréquences maximum atteintes.
On peut voir que la version pipelinée est la moins bonne même si elle a été créée pour aller le plus vite. Les rapports de synthèse nous indiquent que les sommes à plusieurs entrées ralentissent l'architecture. De l'autre coté, on observe que l'accumulateur éclaté en complément à deux s'en sort le mieux, tant en fréquence qu'en termes de ressources.
Les faibles différences de latences observées entre 1000 et 2000 cycles sont dues à la différence de fréquence entre les implémentations. En effet, une fréquence plus élevée implique un nombre de cycles plus grand pour effectuer la normalisation finale. Le débit de ces architectures est donc d'un nombre en entrée par cycle, ce qui n'est pas le cas pour les architectures avec un nombre de cycles supérieur à 2000.
Enfin, une manière plus classique d'augmenter la précision de l'accumulation est de la réaliser dans un format flottant plus grand. Il est donc intéressant de comparer le meilleur accumulateur de Kulisch en précision simple avec l'implémentation IEEE-754 double précision. L'accumulateur de Kulisch nécessite 5 fois plus de ressources mais améliore la latence par un facteur 30. Les techniques visant à réduire la latence des opérateurs flottants nécessitent 5 fois plus de ressources [16]. De plus, l'accumulateur de Kulisch possède une précision garantie quel que soit le nombre de termes ajoutés.
De manière similaire, calculer avec des flottants demi-précision devient bien plus sûr avec ce type d'accumulateur.

6. Conclusion

Cet article a présenté plusieurs implémentations de l'accumulateur de Kulisch, une manière coûteuse mais très rapide de calculer de manière exacte les sommes de produits.
Cet article a montré qu'une version en complément à deux de l'accumulateur de Kulisch pouvait

Entrée	Implémentation	Taille SA	LUTs	Registres	DPSs	Latence	Max Freq
demi-flottants (16 bits)	complément à deux éclaté	16	841	822	1	1021	363 MHz
précision simple (32 bits)	complément à deux éclaté	64	4751	6516	2	1111	344 MHz
	Kulisch éclaté	64	5162	9760	2	1103	209 MHz
	Pipeliné	32	7774	11960	2	1143	298 MHz
		64	5968	9716	2	1098	297 MHz
précision simple (32 bits)	IEEE-754		305	673	9	24218	356 MHz
précision double (64 bits)	complément à deux éclaté	64	42415	72621	9	1795	300 MHz
		128	39590	69826	9	1643	273 MHz
	Kulisch éclaté	64	36688	76171	9	1493	173 MHz
	Pipeliné	64	58175	82188	9	1707	201 MHz
	Additionneur long		58129	59932	9	9233	100 MHz
	en RAM	64	2814	1919	9	68173	100 MHz
précision double (64 bits)	IEEE-754		854	1873	14	35317	338 MHz

TABLE 1 – Résultats de synthèse des accumulateurs présentés pour différentes tailles d'entrées pour 1000 termes à accumuler comparés à des implémentations utilisant la norme IEEE-754 (en gris)

obtenir de meilleures performances pour une plus faible quantité de ressources.
Nos travaux futurs pourront être axés sur l'implémentation d'un arbre de compression [5] pour améliorer les sommes multiples. Ces travaux pourraient aussi être évalués dans le contexte de conception VLSI, de manière à être intégrés dans des processeurs grand public.

Bibliographie

1. *IEEE standard for binary floating-point arithmetic*. Institute of Electrical and Electronics Engineers. Note : Standard 754–1985.
2. F. de Dinechin, B. Pasca, O. Creţ, and R Tudoran. An FPGA-specific approach to floating-point accumulation and sum-of-products. In *Field-Programmable Technologies*, pages 33–40. IEEE, 2008.
3. E. Kadric, P. Gurniak, and A. DeHon. Accurate parallel floating-point accumulation. *IEEE Transactions on Computers*, pages 3224–3238, 2016.
4. U. Kulisch. *Computer arithmetic and validity : Theory, implementation, and applications*. 2013.
5. M. Kumm and P. Zipf. Pipelined compressor tree optimization using integer linear programming. In *Field Programmable Logic and Applications*, pages 1–8, 2014.
6. D. U. Lee, A. A. Gaffar, R. C. C. Cheung, O. Mencer, W. Luk, and G. A. Constantinides. Accuracy-guaranteed bit-width optimization. *Computer-Aided Design of Integrated Circuits and Systems*, pages 1990–2000, 2006.
7. M. Lin, S. Cheng, and J. Wawrzynek. Cascading deep pipelines to achieve high throughput in numerical reduction operations. In *Reconfigurable Computing and FPGAs*, pages 103–108, 2010.
8. Z. Luo and M. Martonosi. Accelerating pipelined integer and floating-point accumulations in configurable hardware with delayed addition techniques. *IEEE Transactions on Computers*, pages 208–218, 2000.
9. J-M. Muller, N. Brisebarre, F. de Dinechin, C-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, 2010.
10. T. Ogita, S. M. Rump, and S. Oishi. Accurate sum and dot product. *SIAM Journal on Scientific Computing*, pages 1955–1988, 2005.
11. T. Ould Bachir and J. P. David. Performing floating-point accumulation on a modern FPGA in single and double precision. In *Field-Programmable Custom Computing Machines*, pages

105–108, 2010.

12. A. Paidimarri, A. Cevrero, P. Brisk, and P. Ienne. FPGA implementation of a single-precision floating-point multiply-accumulator with single-cycle accumulation. In *Field Programmable Custom Computing Machines*, pages 267–270, 2009.
13. O. Sentieys, D. Menard, D. Novo, and K. Parashar. Automatic fixed-point conversion : a gateway to high-level power optimization. Design Automation and Test in Europe, 2014.
14. S. Sun and J. Zambreno. A floating-point accumulator for FPGA-based high performance computing applications. In *Field-Programmable Technology*, pages 493–499, 2009.
15. T. Teufel and M. Baesler. FPGA implementation of a decimal floating-point accurate scalar product unit with a parallel fixed-point multiplier. *Reconfigurable Computing and FPGAs*, pages 6–11, 2009.
16. D. Wilson and G. Stitt. The unified accumulator architecture : A configurable, portable, and extensible floating-point accumulator. *Reconfigurable Technology and Systems*, pages 21 :1–21 :23, 2016.