Opérateurs trigonométriques en virgule flottante sur FPGA

Jérémie Detrey, Florent de Dinechin

LIP, ENS-Lyon
46, allée d'Italie – 69364 Lyon Cedex 07
{Jeremie.Detrey, Florent.de.Dinechin}@ens-lyon.fr

Résumé
Les circuits reconfigurables FPGA ont désormais une capacité telle qu'ils peuvent être utilisés à des tâches d'accélération de calcul en virgule flottante. La littérature (et depuis peu les constructeurs) proposent des opérateurs pour les quatre opérations. L'étape suivante est de proposer des opérateurs pour les fonctions élémentaires les plus utilisées. Parmi celles-ci, les fonctions sinus et cosinus présentent une gamme de compromis d'implémentation extrêmement large, qui est étudiée dans cet article.

Mots-clés : Virgule flottante, FPGA, sinus, cosinus

1. Introduction

1.1. Virgule flottante sur FPGA

Longtemps cantonnés au domaine de la virgule fixe, les circuits reconfigurables sont depuis les années 2000 de plus en plus utilisés pour implémenter des applications de calcul en virgule flottante. La virgule flottante permet de manipuler des nombres réels de grande dynamique, et est à ce titre d'un emploi plus simple que la virgule fixe. En particulier, de nombreux algorithmes de calcul sont souvent développés et testés en logiciel en utilisant la virgule flottante. Il est plus simple alors de les implanter sur FPGA avec la même arithmétique.
Toutefois, les opérateurs flottants sont plus lents et plus coûteux en terme de ressources du FPGA que les opérateurs en virgule fixe. Ils sont ainsi beaucoup plus lents que les opérateurs flottants très optimisés des processeurs du commerce. Typiquement, un opérateur flottant implémenté en FPGA a un débit de crête (en termes d'opérations par seconde) au moins dix fois inférieur à son équivalent dans un processeur de la même génération [11, 16].
Pour surpasser en performance le processeur, il faut donc une application présentant un parallélisme massif [10, 8, 2, 7]. On peut alors maximiser le nombre d'opérateurs flottants par circuit FPGA en utilisant une précision *ad-hoc* pour l'application, là où le processeur ne laisse le choix qu'entre double et simple précision.

1.2. Les fonctions élémentaires

Pour les quatre opérations, une implémentation reconfigurable (et qui paye le prix de la reconfigurabilité) est forcément moins performante que l'implémentation hautement optimisée en ASIC du processeur. Par contre, dès lors qu'on s'intéresse à des opérations plus complexes telles que les fonctions élémentaires, comme exponentielle [5] et logarithme [4], le processeur utilise un algorithme microcodé ou programmé en logiciel, alors que le FPGA peut implémenter des algorithmes matériels beaucoup plus efficaces. Nos travaux précédents ont montré que cette flexibilité du FPGA lui permet alors de surpasser en débit l'implémentation dans le processeur. Pour la simple précision, on peut ainsi obtenir un débit dix fois plus élevé [5, 4].

1.3. Les fonctions trigonométriques

Cet article étudie de ce point de vue les fonctions sinus et cosinus. Comme pour les fonctions exponentielle [5] et logarithme [4] déjà réalisées, nous décrivons une implémentation compatible avec les fonctions simple précision de la bibliothèque mathématique standard disponible dans tout ordinateur

récent. Ceci permet de transposer rapidement un code C ou Matlab sur FPGA sans crainte d'obtenir un comportement numérique différent. Toutefois, ces fonctions sont paramétrées en précision et en dynamique, ce qui permet aussi d'optimiser les ressources FPGA et donc d'exploiter le parallélisme au mieux. Cette implémentation, décrite en section 3, consomme des ressources raisonnables jusqu'à la simple précision.

La première contribution de cet article est donc la description d'algorithmes d'évaluation en matériel de fonctions trigonométriques. Comme pour nos travaux précédents, ces algorithmes s'écartent significativement des équivalents logiciels pour exploiter au mieux la flexibilité offerte par la cible.

Toutefois, la particularité des fonctions trigonométriques est que leur spécification dans les langages de programmation courants n'est pas des plus heureuses du point de vue de l'implémentation. La fonction sinus standard de la bibliothèque mathématique (libm) telle que définie par la norme C99 [9] prend un argument en radian. Cela conduit à une réduction d'argument soit complexe et coûteuse, soit imprécise, comme exposé dans la section 2. Une implémentation de $\sin(\pi x)$, ou une implémentation en degrés, sera bien moins coûteuse pour la même précision. Or on observe que de nombreux codes scientifiques ou de traitement du signal passent au sinus un argument qui a été multiplié préalablement par un terme en π. Cette multiplication préalable non seulement coûte en elle-même, mais rend aussi bien plus coûteuse l'implémentation des fonctions trigonométriques utilisées derrière [1]. Un autre cas particulier courant est celui où l'argument d'entrée est toujours compris entre $-\pi$ et $+\pi$: ici aussi, inutile de réaliser la difficile réduction d'argument. En logiciel, d'ailleurs, ce cas particulier est traité bien plus rapidement que le cas général. En matériel, on devra s'assurer que la partie de circuit réalisant cette réduction d'argument n'est pas synthétisée.

Nous proposons donc également, en section 4, différentes implémentations des fonctions trigonométriques spécifiques, et comparons leurs performances et leur coût matériel. C'est là sans doute l'aspect le plus intéressant de cet article. Il faut noter à ce sujet que le seul article comparable de la littérature[15] est extrêmement flou quant à la spécification de la fonction effectivement implémentée.

Avant de présenter en section 4 les implémentations de ces différentes fonctions, il est sans doute nécessaire de présenter plus en détail, dans la section 2 ci-dessous, la problématique de l'évaluation des fonctions trigonométriques.

Notations

Nous utilisons dans tout l'article les notations de la bibliothèque FPLibrary [3], librement accessible depuis `http://www.ens-lyon.fr/LIP/Arenaire/`, et dans laquelle s'insère ce travail.

La dynamique des nombres manipulés est donnée par w_E, le nombre de bits de l'exposant (E_x), et leur précision est donnée par w_F, le nombre de bits de la partie fractionnaire de la mantisse (F_x). Pour mémoire, en simple précision, on a $w_E = 8$ et $w_F = 23$. Les nombres flottants comptent aussi un bit de signe (S_x), ainsi que deux bits supplémentaires (exn_x) permettant de gérer simplement les cas exceptionnels tels que zéros, infinis et NaN (*Not a Number*).

La taille d'un nombre flottant dans ce format est donc de $w_E + w_F + 3$ bits, comme représenté en figure 1. Ce nombre x a pour valeur

$$x = (-1)^{S_x} \times 1, F_x \times 2^{E_x - E_0},$$

avec $E_0 = 2^{wE-1} - 1$.

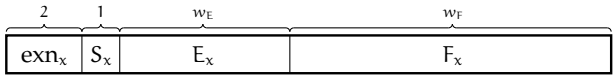


FIG. 1 – Format d'un nombre flottant x.

[1] La révision de la norme définissant le calcul flottant (voir `http://754r.ucbtest.org/`) devrait préconiser de fournir également $\sin(\pi x)$, mais il faudra de nombreuses années avant que ces fonctions se généralisent.

2. Les fonctions sinus et cosinus

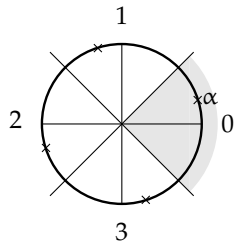
2.1. Réduction d'argument pour les angles en radian

Le lecteur intéressé par les questions discutées dans cette section pourra se rapporter à [13].
Il s'agit de ramener un nombre flottant x qui peut être très grand dans l'intervalle $\left[-\frac{\pi}{4}, \frac{\pi}{4}\right]$, en calculant un entier k et un flottant α tel que

$$\alpha = x - k\frac{\pi}{2} \in \left[-\frac{\pi}{4}, \frac{\pi}{4}\right].$$

Une fois cette réduction d'argument effectuée, le sinus ou le cosinus du nombre de départ est déduit du sinus ou du cosinus de l'argument réduit modulo une identité trigonométrique triviale, donnée par la figure 2. Il faut y ajouter que l'on se ramène toujours à $\alpha \in [0, \pi/4]$ par simple changement de signe :

$$\begin{cases} \sin(-\alpha) = \sin(\alpha), \\ \cos(-\alpha) = \cos(\alpha). \end{cases}$$



Quadrant	Reconstruction	Quadrant	Reconstruction
0	$\begin{cases} \sin(x) = \sin(\alpha) \\ \cos(x) = \cos(\alpha) \end{cases}$	1	$\begin{cases} \sin(x) = \cos(\alpha) \\ \cos(x) = -\sin(\alpha) \end{cases}$
2	$\begin{cases} \sin(x) = -\sin(\alpha) \\ \cos(x) = -\cos(\alpha) \end{cases}$	3	$\begin{cases} \sin(x) = -\cos(\alpha) \\ \cos(x) = \sin(\alpha) \end{cases}$

FIG. 2 – Réduction d'argument par quadrants.

Ici, k est défini comme l'arrondi de $x \times \frac{2}{\pi}$ à l'entier le plus proche. Une variante (utilisée par les implémentations logicielles de Markstein sur IA64 [12]) utilise comme argument réduit y la partie fractionnaire de $x \times \frac{4}{\pi}$, et évalue ensuite $\sin\left(\frac{\pi}{4}y\right)$ et $\cos\left(\frac{\pi}{4}y\right)$. De notre point de vue, cette variante présente deux avantages. D'une part, il y a une soustraction et une multiplication en moins par rapport au calcul de $\alpha = x - k\frac{\pi}{2}$. D'autre part, nous évaluons le sinus et le cosinus de l'argument réduit par une méthode à base de tables [6], qui est plus facile à mettre en œuvre pour un argument réduit dans l'intervalle $\left[-\frac{1}{2}, \frac{1}{2}\right]$ que dans un intervalle comme $\left[-\frac{\pi}{4}, \frac{\pi}{4}\right]$. L'inconvénient principal de cette variante est que le développement limité de $\sin\left(\frac{\pi}{4}y\right)$, dont on a besoin lors de l'évaluation proprement dite, est légèrement moins simple que celui de $\sin(\alpha)$.
La question difficile est de calculer $x \times \frac{4}{\pi}$ avec une précision suffisante. Naturellement, on va utiliser un multiplieur par la constante $\frac{4}{\pi}$, mais pour de très grandes valeurs de x, on a besoin d'une grande précision de cette constante, puisqu'on ne veut garder que la partie fractionnaire du résultat. On peut toutefois remarquer qu'on n'a pas besoin de calculer entièrement la partie entière k de $x \times \frac{4}{\pi}$: on a besoin juste des trois bits de poids faible de k pour déterminer l'octant de l'argument réduit, et donc toute l'information nécessaire à notre réduction d'argument ($|y|$ et le signe de y). Les bits de poids plus forts n'ont pas besoin d'être calculés, puisqu'ils correspondent à des multiples entiers de la période. Comme x est un nombre en virgule flottante, c'est sa mantisse qui sera multipliée par $\frac{4}{\pi}$, et son exposant peut être utilisé pour tronquer à gauche la constante et ne garder que les bits dont on a besoin pour obtenir trois bits entiers du produit. Cette idée est dûe à Payne et Hanek et a été popularisée par K.C. Ng [14].
Combien de bits de $\frac{4}{\pi}$ doivent-ils être multipliés par la mantisse de x ? Sans parler des bits de garde, qui seront déterminés par l'analyse d'erreur et par le paragraphe suivant, il faut multiplier par la mantisse de x au moins $2w_F$ bits de $\frac{4}{\pi}$. En effet, les w_F bits de poids forts du produit ne seront pas valides puisque calculés au moyen d'une constante $\frac{4}{\pi}$ tronquée.

Il reste encore une subtilité. La partie fractionnaire y de $x \times \frac{4}{\pi}$ peut s'approcher extrêmement près de zéro pour certaines valeurs de x. Ainsi, il peut arriver que y commence par une longue suite de zéros. En virgule fixe ce ne serait pas un souci, mais si l'on veut calculer un sinus en virgule flottante dont tous les bits sont significatifs, il faut commencer par assurer que tous les bits de y le soient. Pour cela, il faut ajouter encore g_K bits de garde à la constante $\frac{4}{\pi}$. Un algorithme dû à Kahan et Douglas [13] permet de calculer, pour un domaine flottant donné, quelle est la plus petite valeur possible de y, et donc la valeur de g_K. On constate que g_K est de l'ordre de w_F.
Finalement, la réduction d'argument trigonométrique nécessite
– de stocker la constante $\frac{4}{\pi}$ sur un nombre de bits de l'ordre de $2^{w_E-1} + 3w_F$,
– du matériel pour extraire de l'ordre de $3w_F$ bits de cette constante,
– un multiplieur de l'ordre de $w_F \times 3w_F$ bits,
– un décaleur pour renormaliser éventuellement le résultat.
Le coût en matériel est donc élevé. Le coût en temps peut toutefois être réduit par une architecture à double chemin de calcul (ou *dual path*, comme on en trouve dans les additionneurs flottants), présentée ci-dessous.
D'autres réductions d'argument sont présentées dans [13]. La technique de Cody et Waite n'est utile que pour les petits arguments dans une approche logicielle utilisant des opérateurs flottants. Les variantes de la réduction d'argument modulaire [1, 17] ont été considérées, mais s'avèrent plus coûteuses que l'approche présentée ci-dessus. Plus précisément, il s'agit d'approches itératives qui se prêtent très mal à une implémentation pipelinée.

2.2. Fonctions trigonométriques à réduction d'argument simple

On l'a vu, la difficulté de la réduction d'argument pour les fonctions en radian tient à ce que la constante $\frac{4}{\pi}$ est irrationnelle et qu'on a besoin d'un grand nombre de bits de cette constante. Dès lors que la constante est rationnelle (angles en degrés ou $\sin(\pi x)$), le problème devient beaucoup plus simple. Par exemple, pour $\sin(\pi x)$, la réduction d'argument consiste simplement à décomposer x en sa partie entière et sa partie fractionnaire. Comme x est un flottant, cela coûte tout de même un décalage, mais l'avantage est que cette opération est toujours exacte : la précision intermédiaire requise reste de l'ordre de w_F bits, pas plus.

2.3. Seconde réduction d'argument

Il est possible d'utiliser une seconde réduction d'argument tabulant des valeurs de sinus et cosinus en des points intermédiaires de l'intervalle $\left[0, \frac{\pi}{4}\right]$. En découpant $y = y_H + y_L$, avec y_H les p bits de poids fort de y, on peut alors utiliser les identités suivantes, dans lesquelles les valeurs en y_H sont tabulées :

$$\begin{cases} \sin\left(\frac{\pi}{4}y\right) = \sin\left(\frac{\pi}{4}y_H + \frac{\pi}{4}y_L\right) = \cos\left(\frac{\pi}{4}y_H\right)\sin\left(\frac{\pi}{4}y_L\right) + \sin\left(\frac{\pi}{4}y_H\right)\cos\left(\frac{\pi}{4}y_L\right), \\ \cos\left(\frac{\pi}{4}y\right) = \cos\left(\frac{\pi}{4}y_H + \frac{\pi}{4}y_L\right) = \cos\left(\frac{\pi}{4}y_H\right)\cos\left(\frac{\pi}{4}y_L\right) - \sin\left(\frac{\pi}{4}y_H\right)\sin\left(\frac{\pi}{4}y_L\right). \end{cases}$$

Cette idée a été explorée mais non finalisée. La conclusion est toutefois que la quantité de calculs nécessaires ne se justifie pas pour des précisions allant jusqu'à la simple précision. Les opérateurs sont en principe plus petits, mais l'analyse d'erreur montre qu'il faut rajouter 7 bits de garde à la plupart des chemins de calculs, ce qui annule l'intérêt de cette méthode.
Par contre, cette seconde réduction d'argument devrait rendre l'approche par table viable jusqu'à la double précision, ce que nous explorerons dans un avenir proche.

2.4. Architecture duale sinus et cosinus

Si la réduction d'argument est coûteuse, il est possible de la partager dans le cas fréquent où l'on doit calculer le sinus et le cosinus d'un même angle (par exemple pour calculer une rotation). En fait, telle que présentée ci-dessus, la réduction d'argument oblige l'opérateur à calculer toujours en parallèle le sinus et le cosinus de l'angle, puisque le résultat final sera l'un ou l'autre suivant le quadrant.
Si l'on veut un opérateur plus économique qui ne produise qu'une des deux fonctions, il faut remplacer la constante $\frac{\pi}{4}$ par $\frac{\pi}{2}$ dans la réduction d'argument. Nous verrons que le bénéfice est maigre.

3. Implémentation de référence

Cette implémentation est compatible avec l'esprit de la norme IEEE-754, et avec les pratiques en usage en logiciel : elle implémente une réduction d'argument précise pour obtenir l'arrondi fidèle (c'est-à-dire juste jusqu'au dernier bit) du résultat pour tout nombre en entrée.
L'architecture générale de cette implémentation est donnée par la figure 3. L'architecture spécifique à la réduction d'argument, présentée plus en détail dans la section 3.1, est représentée figure 4. De même, les architectures détaillées pour le calcul du sinus et du cosinus sont données figures 5(a) et 5(b) respectivement.
L'étage de reconstruction quant à lui se contente d'appliquer les identités trigonométriques présentées figure 2 pour retrouver $\sin x$ et $\cos x$ à partir de $\sin\left(\frac{\pi}{4}y\right)$ et $\cos\left(\frac{\pi}{4}y\right)$, en fonction de l'octant indiqué par k. Le signe de x est aussi pris en compte dans le cas du sinus.
Enfin, une petite unité dédiée permet de traiter les cas exceptionnels, comme par exemple le calcul de $\sin(+\infty)$ qui renverra NaN.

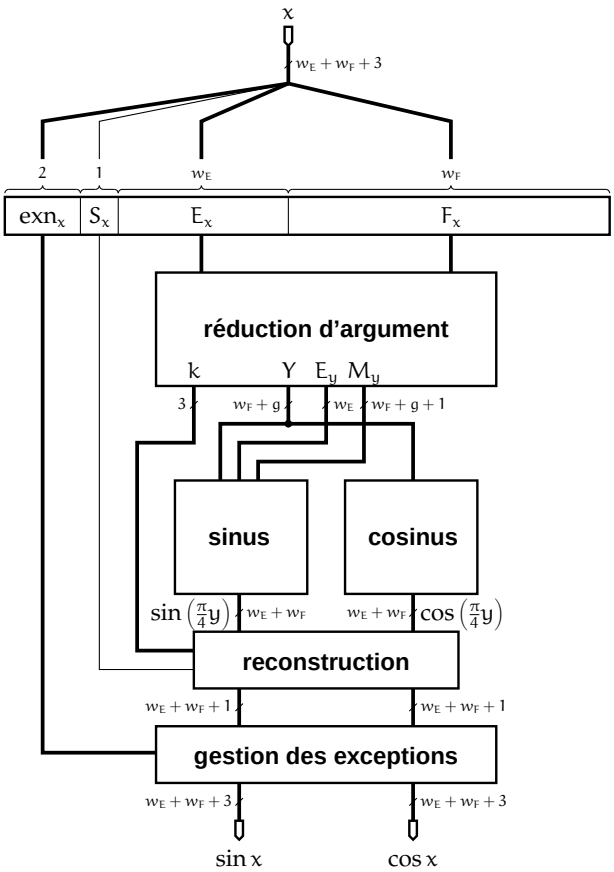


FIG. 3 – Vue d'ensemble de l'architecture de l'opérateur dual sinus/cosinus.

3.1. Réduction d'argument à double chemin

Il faut décrire plus en détail la sortie de la réduction d'argument. On veut pouvoir calculer $\sin\left(\frac{\pi}{4}y\right)$ et $\cos\left(\frac{\pi}{4}y\right)$, tous deux en virgule flottante. Le cosinus sera compris entre $\frac{\sqrt{2}}{2}$ et 1, donc son exposant est connu. Par conséquent, pour le calculer, on peut se contenter d'une représentation de y en virgule fixe, notée Y. Par contre, le sinus peut s'approcher très près de zéro, donc il faut avoir également une représentation flottante de y, dénotée (E_y, M_y) (représentant respectivement exposant et mantisse nor-

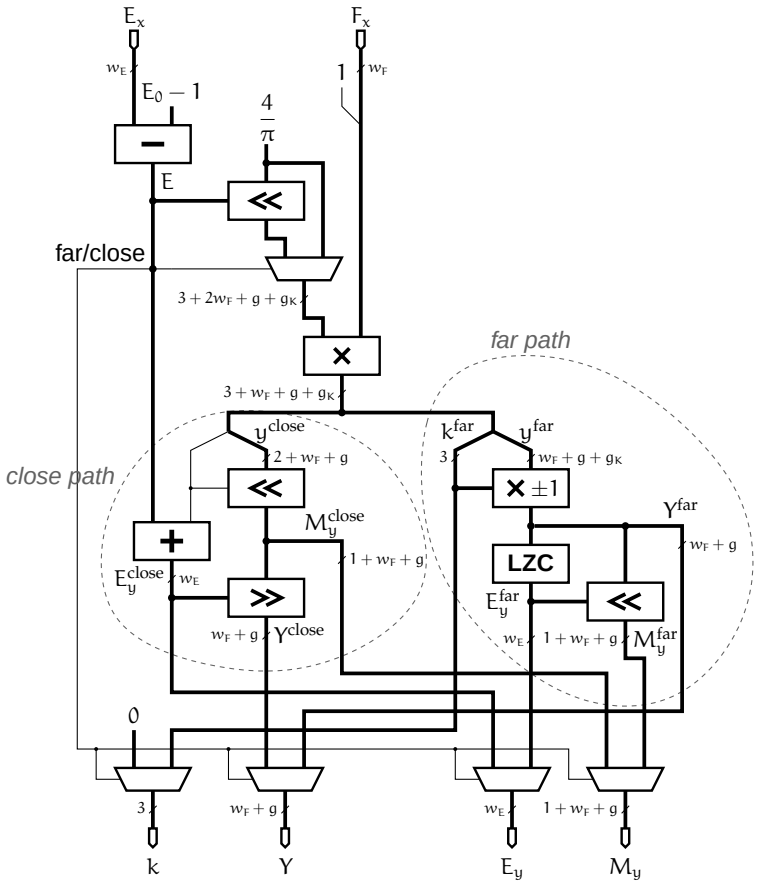


FIG. 4 – Architecture détaillée de la réduction d'argument *dual path*.

malisée de y), obtenue dans le cas général par un compteur de zéros de poids forts (*leading zero counter* ou LZC) couplé à un décaleur.

Il y a un cas particulier lorsque x, le nombre de départ, est proche de zéro (en pratique $x < \frac{1}{2}$). Dans ce cas on peut déduire directement l'exposant de y de celui de x, puisque les deux nombres sont dans un rapport constant de $\frac{4}{\pi}$ (il faut au plus une petite normalisation par un décalage de un bit). Donc on obtient M_y et E_y rapidement, par contre, on obtient Y par un décalage variable de M_y suivant l'exposant E_y.

On obtient donc deux chemins de calcul exclusifs visibles sur la figure 4 : le chemin *close* pour les valeurs de x proches de zéro, qui calcule Y à partir de (E_y, M_y), et le chemin *far*, pour les valeurs de x distantes de zéro, qui calcule (E_y, M_y) à partir de Y.

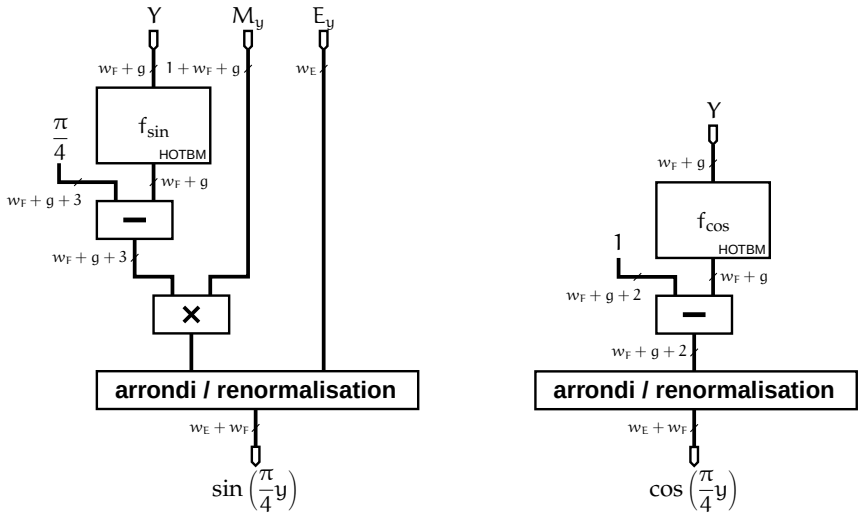
3.2. Évaluation par table de $\cos\left(\frac{\pi}{4}y\right)$ et $\sin\left(\frac{\pi}{4}y\right)$

On veut ici utiliser la méthode HOTBM [6], qui permet d'obtenir des implémentations très compactes de fonctions continues en virgule fixe.

Pour le cosinus, on peut utiliser cette méthode directement puisque, comme déjà dit, l'exposant du résultat est connu, autrement dit le résultat est en virgule fixe. On économise toutefois le stockage du premier bit, qui vaut toujours 1, en choissant d'évaluer la fonction

$$f_{\cos}(y) = 1 - \cos\left(\frac{\pi}{4}y\right).$$

Le sinus, par contre, peut s'approcher près de zéro, et donc aura un exposant variable. Pour se ramener



(a) Sinus (b) Cosinus

FIG. 5 – Architecture détaillée de l'évaluation du sinus (a) et du cosinus (b).

à une fonction en virgule fixe, on utilise l'équation

$$\sin\left(\frac{\pi}{4}y\right) = y \times \frac{\sin\left(\frac{\pi}{4}y\right)}{y}$$

L'intérêt est que le terme de droite du produit peut être tabulé en virgule fixe : son développement de Taylor est

$$\frac{\sin\left(\frac{\pi}{4}y\right)}{y} \approx \frac{\pi}{4} + O(y^2),$$

alors que l'exposant du résultat est déterminé par l'exposant de y, E_y. On choisit donc de tabuler la fonction

$$f_{sin}(y) = \frac{\pi}{4} - \frac{\sin\left(\frac{\pi}{4}y\right)}{y}.$$

La multiplication par y est une multiplication d'un nombre en virgule flottante (E_y, M_y) par un nombre en virgule fixe, et donc relativement simple.

3.3. Analyse d'erreur

Le dilemme du fabriquant de table [13] nous empêchant d'atteindre l'arrondi correct, nous devons nous contenter d'arrondi fidèle pour les résultats. L'erreur relative commise sur la mantisse est bornée par 2^{-w_F}.

Cette erreur maximale nous donne donc une contrainte qui, couplée à un suivi précis de la propagation de toutes les erreurs d'arrondi et de troncation dans le circuit, permet de déterminer le nombre minimal de bits de garde nécessaires. On trouve ainsi que $g = 2$ bits sont requis pour garantir l'arrondi fidèle.

4. Implémentations dégradées

Devant la surface relativement importante occupée par l'opérateur dual sinus/cosinus (voir section 5), nous avons exploré diverses alternatives à cet opérateur, pour proposer plusieurs compromis entre précision, surface et latence.

4.1. Opérateur seul

Certaines applications n'ont besoin de calculer que des sinus mais pas de cosinus (ou, de manière équivalente, l'inverse). Pour gagner un peu de matériel, nous proposons donc une version de notre opérateur ne calculant que le sinus de x.

Pour éviter d'avoir à calculer de cosinus, la réduction d'argument doit se faire sur les quadrants et non les octants du cercle unité. Ainsi, l'angle réduit α sera dans l'intervalle $\left[0, \frac{\pi}{2}\right]$, deux fois plus large que pour l'opérateur dual.
La réduction est quasiment identique à la réduction d'argument de l'opérateur dual, si ce n'est la constante qui est $\frac{2}{\pi}$ au lieu de $\frac{4}{\pi}$. Cela ne va donc pas permettre d'économiser de matériel : le gros multiplieur $w_F \times 3w_F$ bits est toujours nécessaire.
Quant à l'évaluation de la fonction $f_{sin}(y)$, puisqu'elle s'effectue sur un intervalle deux fois plus large, l'opérateur HOTBM va occuper plus de place.
Ainsi, le gain de la suppression de l'opérateur pour $f_{cos}(y)$ ne peut être que très léger, comme le confirment les résultats présentés section 5.

4.2. Une implémentation avec sortie en virgule fixe

Une grande part de la surface et du délai de l'opérateur sont dus au multiplieur utilisé pour la réduction d'argument. En relâchant les contraintes de précision, on peut diminuer le nombre de bits de la constante $\frac{4}{\pi}$ à prendre en compte, et ainsi directement diminuer la taille et la latence de ce multiplieur.
Considérer un opérateur qui fournirait sa sortie en virgule fixe nous a semblé une option intéressante, puisqu'elle permettait justement d'éliminer les g_K bits de garde supplémentaires nécessaires pour la constante, mais aussi simplifiait le calcul de $\sin\left(\frac{\pi}{4}y\right)$. En effet, n'ayant plus besoin de renormaliser le résultat, on peut évaluer la fonction sinus directement à l'aide de HOTBM, sans passer par la fonction f_{sin}. On économise donc alors aussi la multiplication par M_y.
Le gain est effectivement substantiel, comme le montrent les résultats de la section 5.

4.3. Fonctions trigonométriques en degrés et en πx

Comme déjà mentionné précédemment, les fonctions trigonométriques en radians ne sont pas nécessaires pour de nombreux programmes, qui pourraient très bien s'accommoder de fonctions trigonométriques travaillant en degrés ou en multiples de π.
Nous avons donc étudié de près la réalisation d'un opérateur dual calculant $\sin(\pi x)$ et $\cos(\pi x)$. L'intérêt d'un tel opérateur est de simplifier grandement le processus de réduction d'argument, en épargnant le calcul coûteux du produit $x \times \frac{4}{\pi}$. De plus, y étant calculé exactement, cela permet de gagner un bit de garde pour l'évaluation des fonctions f_{sin} et f_{cos}.
Encore une fois, les résultats obtenus sont très intéressants et encouragent à poursuivre le développement aussi sur cette voie.

5. Résultats

Tous les opérateurs présentés dans cet article ont été synthétisés, placés et routés pour diverses précisions. Ces résultats ont été obtenus sous Xilinx ISE et XST 7.1, en ciblant un FPGA Virtex-II XC2V1000-4. Ces résultats sont présentés dans la table 1, en termes de *slices* et de pourcentage de la surface du FPGA, et en termes de nanosecondes pour la latence des opérateurs.
Dans un souci de portabilité, les opérateurs ne nécessitent pas forcément l'utilisation des petits multiplieurs 18×18 bits présents dans les modèles récents de FPGA. Cependant, les opérateurs peuvent bien entendu tirer parti de ces multiplieurs si ceux-ci sont disponibles. Nous présentons donc les résultats pour les deux alternatives : multiplieurs synthétisés sur la logique du FPGA, ou bien sur les petits multiplieurs dédiés.
Tous les FPGA actuels disposent aussi de petits blocs de mémoire embarqués. Cependant, du fait de la taille fixe (généralement 18k bits) de ces blocs et malgré la souplesse de leurs modes d'adressage, l'utilisation de ceux-ci est généralement sous-optimale et gâche trop de matériel. Nous ne présentons donc pas dans cet article de résultats de placement/routage avec ces blocs mémoire. Cependant, leur utilisation reste bien entendu possible, grâce à une simple option du synthétiseur.

6. Conclusion et perspectives

Nous avons décrit un opérateur de grande qualité pour le calcul en matériel des fonctions sinus et cosinus en virgule flottante paramétrée. En raison du coût de la réduction d'argument, plusieurs versions

Précision (w_E, w_F)	Multiplieurs	Dual sinus/cosinus Surface (slices)	%	mults)	Latence (ns)	Sinus seul Surface (slices)	%	mults)	Latence (ns)
(5, 10)	logique	803	(15%)	–	69	709	(13%)	–	70
	18 × 18	424	(8%)	7	61	310	(6%)	6	62
(6, 13)	logique	1159	(22%)	–	86	1027	(20%)	–	86
	18 × 18	537	(10%)	7	76	474	(9%)	6	79
(7, 16)	logique	1652	(32%)	–	91	1428	(27%)	–	92
	18 × 18	816	(15%)	10	87	609	(11%)	10	83
(7, 20)	logique	2549	(49%)	–	99	2050	(40%)	–	101
	18 × 18	1372	(26%)	17	96	979	(19%)	15	92
(8, 23)	logique	3320	(64%)	–	109	2659	(51%)	–	105
	18 × 18	1700	(33%)	19	99	1279	(24%)	16	100

Précision (w_E, w_F)	Multiplieurs	Sinus/cosinus en virgule fixe Surface (slices)	%	mults)	Latence (ns)	Sinus/cosinus en πx Surface (slices)	%	mults)	Latence (ns)
(5, 10)	logique	376	(7%)	–	57	363	(7%)	–	58
	18 × 18	241	(4%)	4	51	244	(4%)	3	46
(6, 13)	logique	642	(12%)	–	63	641	(12%)	–	61
	18 × 18	380	(7%)	4	58	462	(9%)	3	61
(7, 16)	logique	923	(18%)	–	72	865	(16%)	–	73
	18 × 18	528	(10%)	5	70	559	(10%)	4	69
(7, 20)	logique	1620	(31%)	–	82	1531	(29%)	–	84
	18 × 18	973	(19%)	9	75	1005	(19%)	8	76
(8, 23)	logique	2203	(43%)	–	88	2081	(40%)	–	89
	18 × 18	1294	(25%)	12	80	1365	(25%)	10	85

TAB. 1 – Surface et latence des différentes variantes d'opérateurs trigonométriques.

dégradées ont été également décrites. L'histoire dira lesquels de ces opérateurs sont les plus utiles, et dans quel contexte.

Les opérateurs présentés n'étant pour l'instant disponibles que sous forme purement combinatoire, nous comptons dans un premier temps achever de les pipeliner.

Mais surtout, comme la littérature récente s'intéresse de plus en plus au calcul en double précision sur FPGA, nous comptons adapter FPLibrary pour qu'elle passe à l'échelle pour ce format. C'est déjà le cas pour les quatre opérations, même si des optimisations à la marge sont possibles. Pour les fonctions élémentaires, par contre, les méthodes à base de tables montrent leurs limites aux alentours de 32 bits [6]. Les deux grandes directions à explorer sont :

– une meilleure réduction d'argument permettant de continuer à utiliser les méthodes à base de tables (ce sera le cas pour les fonctions trigonométriques) ;
– des techniques plus classiques d'approximation polynomiales utilisant plus de multiplications, dont la taille sera maîtrisée par une étude soigneuse des précisions intermédiaires nécessaires.

Bibliographie

1. M. Daumas, C. Mazenc, X. Merrheim, and J. M. Muller. Modular range reduction : A new algorithm for fast and accurate computation of the elementary functions. *Journal of Universal Computer Science*, 1(3) :162–175, March 1995.
2. M. deLorimier and A. DeHon. Floating-point sparse matrix-vector multiply for FPGAs. In *ACM/SIGDA Field-Programmable Gate Arrays*, pages 75–85. ACM Press, 2005.
3. J. Detrey and F. de Dinechin. Outils pour une comparaison sans a priori entre arithmétique logarithmique et arithmétique flottante. *Technique et science informatiques*, 24(6) :625–643, 2005.

4. J. Detrey and F. de Dinechin. A parameterizable floating-point logarithm operator for FPGAs. In *39th Asilomar Conference on Signals, Systems & Computers*. IEEE Signal Processing Society, November 2005.
5. J. Detrey and F. de Dinechin. A parameterized floating-point exponential function for FPGAs. In *IEEE International Conference on Field-Programmable Technology (FPT'05)*. IEEE Computer Society Press, December 2005.
6. J. Detrey and F. de Dinechin. Table-based polynomials for fast hardware function evaluation. In *16th Intl Conference on Application-specific Systems, Architectures and Processors*. IEEE Computer Society Press, July 2005.
7. Y. Dou, S. Vassiliadis, G. K. Kuzmanov, and G. N. Gaydadjiev. 64-bit floating-point FPGA matrix multiplication. In *ACM/SIGDA Field-Programmable Gate Arrays*. ACM Press, 2005.
8. G. Govindu, L. Zhuo, S. Choi, and V. Prasanna. Analysis of high-performance floating-point arithmetic on FPGAs. In *Reconfigurable Architecture Workshop, Intl. Parallel and Distributed Processing Symposium*. IEEE Computer Society, 2004.
9. ISO/IEC. *International Standard ISO/IEC 9899 :1999(E). Programming languages – C*. 1999.
10. G. Lienhart, A. Kugel, and R. Männer. Using floating-point arithmetic on FPGAs to accelerate scientific N-body simulations. In *FPGAs for Custom Computing Machines*. IEEE, 2002.
11. W.B. Ligon, S. McMillan, G. Monn, K. Schoonover, F. Stivers, and K.D. Underwood. A re-evaluation of the practicality of floating-point operations on FPGAs. In *IEEE Symposium on FPGAs for Custom Computing Machines*, Napa Valley, USA, 1998.
12. P. Markstein. *IA-64 and Elementary Functions : Speed and Precision*. Hewlett-Packard Professional Books. Prentice Hall, 2000. ISBN : 0130183482.
13. J.-M. Muller. *Elementary Functions, Algorithms and Implementation*. Birkhauser, Boston, 1997/2005.
14. K. C. Ng. Argument reduction for huge arguments : Good to the last bit. *SunPro*, July 1992.
15. F. Ortiz, J. Humphrey, J. Durbano, and D. Prather. A study on the design of floating-point functions in FPGAs. In *Field Programmable Logic and Applications*, volume 2778 of *LNCS*, pages 1131–1135. Springer, September 2003.
16. K. Underwood. FPGAs vs. CPUs : Trends in peak floating-point performance. In *ACM/SIGDA Field-Programmable Gate Arrays*. ACM Press, 2004.
17. J. Villalba, T. Lang, and M. A. Gonzalez. Double-residue modular range reduction for floating-point hardware implementations. *IEEE Transactions on Computers*, 55(3) :254–267, March 2006.