

Université Paris-Saclay

Master 2 Calcul Haute Performance, Simulation
Rapport de stage de fin d'étude

POSIT GPGPU pour HPC

Auteur :

Rodolphe Thienard

Encadrants :

Anthony Besseau

Vijay Holimath

Soraya Zertal

Table des matières

1 Introduction	1
2 Représentation des nombres flottants	2
2.1 Représentation du standard IEEE	2
2.2 Représentation de Posit	2
2.3 Représentations supplémentaires	4
2.4 Exemples	4
3 Posit selon la littérature	6
3.1 Couverture	6
3.2 Précision	6
3.3 Nan, underflow, overflow	7
3.4 Représentation sur l'anneau réel	7
3.5 Posit sur quel matériel ?	8
4 Emulations de Posit	10
4.1 SoftPosit	10
4.2 Ceralaine	10
4.3 Posit-GCC	10
4.4 Fastsigmoid	11
4.5 Quel est l'impact de l'exposant ?	11
4.6 Validation des émulations	13
5 Orientation GPGPU / Autres alternatives au standard	18
5.1 CUDA	18
5.2 Représentation sur 16 bits	19
5.3 Représentation sur 8 bits	20
6 Posit implémentation matériel : RACER	23
6.1 Architecture	23
6.2 Compilation	24
6.3 Problèmes rencontrés	25
6.4 Resultats	26
7 Cas d'études	30
7.1 Simulation moléculaire	30
7.2 Algèbre linéaire	31
8 Conclusion	32
9. Bibliographie	33
10 Annexe	35
10.1 Glossaire	35
10.2 Mesures des émulations	35
10.3 Représentation architecture RacEr	38
10.4 Portage d'une application	39
10.5 Manuel RacEr	41
10.6 Implémentation sur carte FPGA	44
10.7 Modification IP Zynq Ultrascale pour la carte KV260	46

Remerciements

Je souhaiterais exprimer ma sincère reconnaissance envers mon encadrant, Anthony Besseau, qui a su me conseiller tout au long de ce stage. Ses connaissances techniques autour du FPGA m'ont permis de découvrir ce domaine qui m'attirait, avec plaisir, et d'être soutenu lors des difficultés que j'ai pu rencontrer.

Je suis également très reconnaissant envers mon co-encadrant, Vijay Holimath, pour sa réactivité, son aide et ses conseils lorsque j'avais des interrogations autour de la technologie RacEr, malgré le fait qu'il soit en Inde. Il a même su m'accompagner au-delà de RacEr lors de mes essais sur FPGA.

Je remercie également Morteza et Emmanuel, qui ont pris le temps de résoudre avec moi des problèmes autour de la création d'IP pour FPGA.

Enfin, je remercie également Élodie, Matis, Didier et Bernard, mes collègues, avec qui j'ai passé de bons moments lors des déjeuners et des pauses café.

Entreprise d'accueil

M. Eric Gaudibert était employé chez Analog Devices, une entreprise américaine présente dans les secteurs militaire, aéronautique et spatial. Suite à la suppression de son poste, Analog Devices a exprimé son intention de trouver une structure en France pour poursuivre son développement commercial. C'est ainsi que M. Gaudibert a contacté M. Michel Gallo pour créer une entreprise ensemble. Ils ont fondé cette entreprise en tant qu'agents commerciaux intermédiaires, entre les clients et Analog Devices. Leur objectif principal était de réaliser un chiffre d'affaires spécifique pour Analog Devices tout en travaillant également avec d'autres marques.

C'est 6 ans plus tard qu'EMG2 a été contactée par Bittware, une société américaine développant des cartes électroniques basées sur les composants d'Analog Devices. M. Anthony Besseau a rejoint EMG2 en tant qu'ingénieur d'applications, fournissant un soutien technique et des solutions aux clients. L'entreprise comptait également des assistantes commerciales et des commerciaux.

Au fil des années, des changements de direction ont eu lieu. M. Eric Gaudibert prit sa retraite et son poste fut proposé à M. Patrick Pruvot. La crise financière de 2008, a entraîné la mise en arrêt du contrat avec plusieurs fournisseurs, ce qui va engendrer une perte du chiffre d'affaires importante et le départ de certains employés.

Malgré ces contraintes, EMG2 a continué à évoluer. En 2010, M. Michel Gallo a pris sa retraite et M. Patrick Pruvot est devenu le gérant de l'entreprise, avec M. Anthony Besseau en tant qu'associé. EMG2 a également réussi à trouver de nouveaux partenaires après la résiliation du contrat avec Analog Devices en 2011.

Au fil du temps, EMG2 a adapté son modèle d'entreprise. En 2013, ils ont accepté une proposition de l'entreprise NAT pour devenir un distributeur plutôt qu'un simple agent commercial intermédiaire, ce qui a engendré l'intégration de toute la partie logistique à leurs activités. En 2020, M. Patrick Pruvot prend sa retraite, laissant M. Anthony Besseau comme unique propriétaire de l'entreprise.

Entre 2020 et 2022, l'entreprise est en constante évolution, la création de deux BU ou deux pôles (Business Unit), la distribution et l'activité d'agent commercial. En 2022, une structure holding appelée "AGAATE" est créée, et M. Anthony Besseau fonde ABTICS pour R&D dans le secteur de la visions et communications.

En 2023, M. Anthony Besseau fonde Vividsparks Europe pour proposer la technologie développée par VividSparks. VividSparks est une entreprise Indienne spécialisée dans les semi-conducteurs sans usine. Elle est orientée vers la résolution de problèmes fondamentaux dans la conception de circuits arithmétiques. Leur technologie est construite avec l'objectif de fournir de haute performances aux utilisateurs avec une consommation d'énergie minimale. VividSparks veut créer un monde informatique qui consomme moins d'énergie, moins de surface de silicium, tout en offrant des performances de haut niveau.

Grâce à ces nouveaux partenariats, EMG2 continue de renforcer son positionnement sur le marché de la distribution de technologies innovantes. En diversifiant son offre et en collaborant avec des entreprises internationales, EMG2 consolide sa capacité à fournir des solutions de pointe dans les secteurs médical, militaire, aéronautique et spatial, contribuant ainsi à l'avancement technologique en fournissant ces briques technologiques aux entreprises d'ingénierie.

À l'avenir et dans le but de consolider son positionnement premium, EMG2 souhaite renforcer son offre technologique en s'appuyant sur ABTICS, afin d'améliorer, son support technique, son équipe commerciale et sa communication numérique. Ils envisagent de diversifier leurs offres avec des solutions logicielles, tout en restant centrés sur le matériel.

1 Introduction

Dans le domaine des systèmes embarqués et des calculs intensifs, la recherche de performances optimales tout en minimisant la consommation d'énergie et l'espace de stockage est un enjeu majeur. Pour atteindre ces objectifs, les ingénieurs ont de plus en plus recours aux FPGA (Field Programmable Gate Arrays). Ces dispositifs logiques reprogrammables offrent une flexibilité sans égale, permettant d'adapter l'architecture matérielle aux besoins spécifiques des applications, qu'il s'agisse de traitement de signaux, de calcul scientifique ou d'autres tâches complexes nécessitant une haute performance ou de concevoir des architectures optimales.

Au cœur de ces calculs, la représentation numérique des nombres réels joue un rôle crucial. Le standard IEEE 754, largement adopté pour la représentation des nombres à virgule flottante, est depuis longtemps la référence. Il définit comment les nombres sont stockés et manipulés dans les systèmes informatiques. Toutefois, bien qu'efficace, ce standard présente certaines limites, notamment en matière de précision et d'efficacité.

C'est dans ce contexte qu'émerge Posit, une alternative au format IEEE 754. Développé par John Gustafson, Posit (également connu sous le nom d'Unum 3) se propose de résoudre certaines des limitations du standard IEEE 754 en offrant une représentation des nombres réels plus efficace. Grâce à une distribution plus homogène de la précision sur l'ensemble des valeurs numériques, Posit parvient à minimiser les erreurs d'arrondi et de troncature, tout en utilisant moins de bits. Cette efficacité accrue se traduit par des calculs plus précis et potentiellement plus rapides, une caractéristique particulièrement avantageuse.

L'adoption de Posit ne se limite pas à une simple amélioration de la précision. En optimisant la représentation des données numériques, Posit permet également de réduire la taille de stockage des informations et la consommation d'énergie lors des calculs. Cela ouvre la voie à des gains non négligeable dans des environnements où la puissance et l'espace sont limités. Les bénéfices s'étendent également à la réduction du temps de calcul, un facteur clé dans les applications nécessitant des réponses en temps réel. Tout ces points en font un concurrent très sérieux dans le domaine de l'IA.

Dans le cadre de ce stage, nous avons été amené à devoir vérifier une implémentation spécifique de Posit, nommée RacEr. RacEr est une implémentation de type CUDA utilisant Posit pour la représentation des nombres flottants. Actuellement disponible exclusivement sur FPGA, RacEr a pour objectif d'être produit pour des architectures GPGPU (General-Purpose computing on Graphics Processing Units). Cette implémentation tire parti des capacités massivement parallèles des GPU pour exécuter des calculs intensifs avec une précision et une efficacité énergétique accrues.

Ainsi, nous verrons dans les prochaines parties si Posit peut devenir une alternative pérenne au standard actuel. Nous commencerons donc par vérifier les valeurs théoriques annoncées par John Gustafson. Pour se faire, nous utiliserons les implémentations de Posit reconnues comme Softposit ou Ceralaine. Puis dans un second temps, nous comparerons ces résultats avec l'implémentation RacEr, afin de pouvoir valider ou non son implémentation de Posit.

2 Représentation des nombres flottants

Comme mentionné précédemment, la principale différence entre Posit et IEEE 754 réside dans la manière dont les bits sont représentés et interprétés. Cependant, Posit n'est pas la seule alternative possible au standard IEEE 754. Certaines implémentations, telles que TensorFloat-32 ou Bfloat16, adoptent une approche différente de la représentation des nombres tout en préservant certains aspects du standard IEEE, offrant ainsi une solution intermédiaire entre ces deux modèles.

2.1 Représentation du standard IEEE

Commençons par la représentation du standard IEEE 754, qui définit la manière dont les nombres à virgule flottante sont codés. Ce standard répartit les bits en trois composantes principales : le **signe**, l'**exposant** et la **mantisso**. Il permet de représenter des nombres à virgule flottante sur différentes longueurs de bits, notamment 16, 32, 64, 128, et 256 bits, avec des configurations spécifiques pour chaque taille.

La répartition des bits entre ces composantes varie en fonction du format utilisé. Par exemple, en FP16 (format 16 bits), les bits sont alloués de manière fixe, ce qui influence directement la précision et la plage de valeurs représentables.

Les rôles de chacune des parties sont les suivants :

- Le **signe** indique si le nombre est positif (0) ou négatif (1).
- L'**exposant** détermine l'ordre de grandeur du nombre, en décalant la valeur de la mantisse selon une puissance de 2. Un biais est appliqué à l'exposant pour permettre la représentation de nombres positifs et négatifs.
- La **mantisso** ajuste la précision du nombre en représentant les chiffres significatifs après la virgule.

La valeur d'un nombre à virgule flottante de 16 bits (FP16) selon le standard IEEE 754 se calcule à l'aide de la formule suivante :

$$(-1)^{\text{signe}} \times 2^{\text{exposant}-15} \times (1 + \text{mantisse})$$

Fig. 1. – Formule pour calculer une valeur du standard IEEE 754

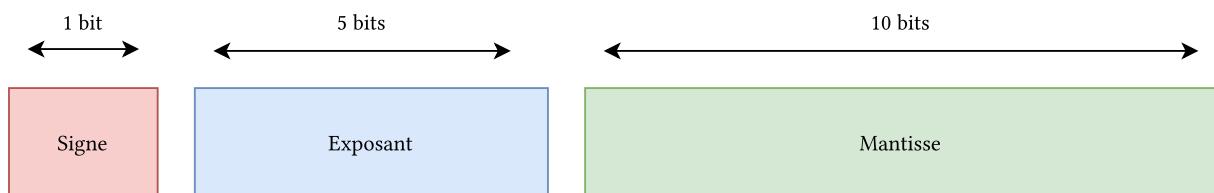


Fig. 2. – Représentation IEEE 754 16 bits

Ainsi, pour une représentation sur 16 bits, les bits sont répartis comme suit : 1 bit pour le signe, 5 bits pour l'exposant, et 10 bits pour la mantisse. Cette répartition permet de garantir une précision continue sur l'ensemble des valeurs représentées.

Le standard IEEE 754, tout en étant largement adopté, présente des compromis entre précision, plage de valeurs et utilisation des bits. Cette rigidité peut entraîner des erreurs d'arrondi et des pertes de précision dans certaines applications critiques, d'où la recherche d'alternatives comme Posit qui tentent de surmonter ces limitations.

2.2 Représentation de Posit

Contrairement au standard IEEE 754, la représentation de Posit n'est pas strictement déterminée par un nombre fixe de bits pour chaque composante. La structure de Posit est dynamique, ce qui signifie que la distribution des bits varie en fonction des valeurs représentées, offrant ainsi une flexibilité accrue.

Le rôle de l'exposant est un exemple de cette flexibilité : il peut être ajusté en fonction du nombre total de bits alloués, mais il oscille généralement entre 0 et 3 bits.

En plus de cette flexibilité, Posit introduit une structure différente en répartissant les bits entre quatre composantes principales : le **signe**, l'**exposant**, la **fraction** (similaire à la mantisse dans IEEE 754), et le **régime**.

- Le **signe** indique si le nombre est positif ou négatif, comme dans IEEE 754.
- Le **régime** est une composante unique à Posit, déterminant l'échelle de la valeur représentée par le nombre. Il permet de gérer plus efficacement une large gamme de valeurs, avec un impact significatif sur la précision.
- L'**exposant** ajuste la grandeur du nombre en fonction du régime, offrant une répartition plus équilibrée de la précision sur l'ensemble des valeurs.
- La **fraction** représente les chiffres significatifs, similaire à la mantisse en IEEE 754, mais avec une répartition des bits plus adaptative.

La valeur d'un nombre à virgule flottante de 16 bits grâce à Posit se calcule à l'aide de la formule suivante :

$$(-1)^{\text{signe}} \times 2^{2^{\text{es}} * k} \times 2^e \times (1 + \text{fraction})$$

Fig. 3. – Formule pour calculer une valeur de Posit

avec k représente la valeur du régime , es représente la taille de l'exposant , et e représente la valeur de l'exposant.

Les tableaux ci dessous permettent de calculer des valeurs calculables à partir du nombre binaire et de la valeur de l'exposant.

Binary	0000	0001	001x	01x	10x	110x	1110	1111
Numerical meaning, k	-4	-3	-2	-1	0	1	2	3

Tableau 1. – Signification de la longueur d'exécution k des bits du régime, issue de « Beating Floating Point at its Own Game » [1]

es	0	1	2	3	4
useed	2	$2^2 = 4$	$4^2 = 16$	$16^2 = 256$	$256^2 = 65536$

Tableau 2. – L'utilisation en fonction de l'es, issue de « Beating Floating Point at its Own Game » [1]

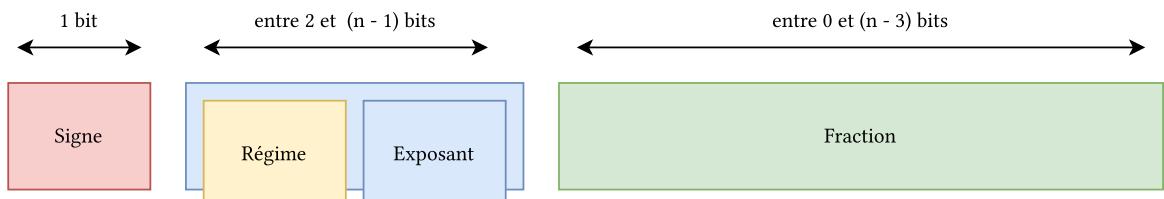


Fig. 4. – Représentation de Posit 16 bits

Cette architecture dynamique permet à Posit d'allouer les bits de manière plus efficace, ce qui se traduit par une précision améliorée sur une gamme plus large de valeurs par rapport à IEEE 754. De plus, cette flexibilité permet une réduction des erreurs de troncature et d'arrondi, tout en utilisant moins de bits pour représenter des nombres de manière précise. Cela en fait une solution particulièrement adaptée pour des applications exigeantes en calculs numériques, où la précision et l'efficacité sont cruciales.

2.3 Représentations supplémentaires

Nous avons également examiné d'autres représentations que le standard IEEE 754 et Posit. Parmi celles-ci, nous en avons retenu deux : TensorFloat-32 et BFloat16.

- **TensorFloat-32 (TF32)** : Ce format, développé par NVIDIA, est spécialement conçu pour accélérer les calculs dans les architectures de GPU modernes. Le TensorFloat-32 est une fusion des représentations 16 et 32 bits du standard IEEE 754. Utilisant la mantisse de la représentation 16 bits et l'exposant de celle à 32 bits, il vise à proposer une précision suffisante avec une rapidité de calcul améliorée pour les opérations de matrices et les réseaux de neurones profonds.



Fig. 5. – Représentation de Tensor Float 32

- **BFloat16 (Brain Floating Point 16)** : Introduit par Google, le BFloat16 est un format de nombre à virgule flottante qui utilise 16 bits. À l'égal du TF32, le BFloat16 offre une couverture aussi large que le format 32 bits du standard, mais au détriment de la précision. BFloat16 est conçu pour fournir une gamme d'exposants plus étendue tout en conservant une précision suffisante pour les calculs de réseaux de neurones.

Ces formats alternatifs offrent des avantages spécifiques pour certaines applications, notamment en termes de vitesse de traitement et de capacité à gérer des plages de valeurs différentes que celles proposées par Posit ou le standard IEEE 754.

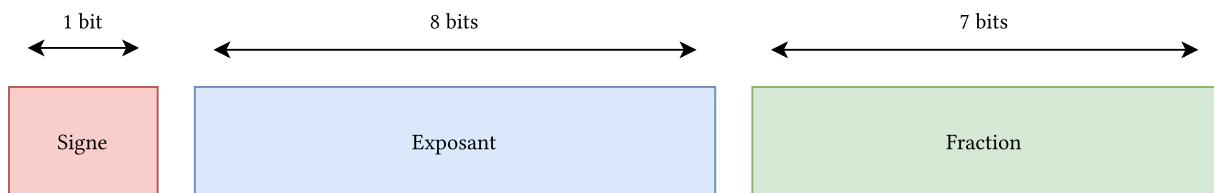


Fig. 6. – Représentation de Brain float 16

2.4 Exemples

De manière à rendre toutes ces explications de représentations plus concrètes, prenons comme exemple un nombre écrit en binaire et voyons comment celui-ci est représenté avec le standard IEEE 754 et avec Posit.

Le nombre binaire 16 bits est : 0011000101101001

Celui ci, une fois segmenté avec la représentation du standard, ressemble à : 0_01100_0101101001.

$$\begin{aligned} &= (-1)^0 \times 2^{-3} \times \left(1 + \frac{0101101001}{1024}\right) \\ &= 1 \times \frac{1}{8} \times (1 + 0.3544921875) \\ &= 0.1693115234375 \end{aligned}$$

Si on décide de prendre comme comparaison le Posit<16,1>, c'est à dire avec 1 bit d'exposant. On obtient alors la répartition des bits suivante : 0_01_1_000101101001.

$$\begin{aligned} &= (-1)^0 \times 4^{-1} \times 2^1 \times (1 + 0.171875) \\ &= 1 \times \frac{1}{4} \times 2 \times 1.171875 \\ &= \frac{1}{2} \times 1.171875 \\ &= 0.5859375 \end{aligned}$$

On remarque ainsi une différence notable dans les résultats des valeurs pour un même nombre binaire donné.

Poursuivons notre exemple en imaginant que nous voulions représenter le nombre 1,24680 en avec le standard IEEE. Ce nombre, encodé en FP16, serait arrondi à une valeur de 1.24707. Il serait ainsi compris entre 1.24609 et 1.24707. On aurait donc pour ce nombre une erreur de $\varepsilon = 0.00027$ (epsilon).

En repétant ce même calcul avec Posit<16,1>, c'est à dire représenter le nombre 1.24680 en Posit 16 bits avec 1 bit d'exposant. Ce nombre serait arrondi à une valeur de 1.24682. Il serait contenu entre 1.24658 et 1.24682. Ainsi, pour la représentation de ce nombre, nous aurions une erreur de $\varepsilon = 0.00002$. A titre d'information, le TensorFloat 32 renverrait une valeur de : 1.24609 et le BFloat une valeur de : 1.25000.

Comme nous l'avons vu dans cette section, différentes représentations émergent pour offrir des solutions précises aux besoins actuels. Hormis Posit, qui est proposé comme un remplaçant potentiel du standard IEEE 754, les autres propositions se concentrent sur des applications liées à l'intelligence artificielle. La représentation des nombres flottants dans le domaine de l'IA est primordiale pour garantir la précision des calculs numériques et de l'apprentissage automatique. En raison de sa grande précision, Posit apparaît comme une solution potentiellement supérieure pour ce type d'application.

3 Posit selon la littérature

La littérature présente Posit comme un concurrent sérieux du standard IEEE 754 sur plusieurs aspects cruciaux. Certains vont jusqu'à affirmer que Posit pourrait remplacer le standard IEEE 754 dans de nombreuses applications, grâce à ses avantages potentiels en termes de précision, de flexibilité et d'efficacité. Dans cette partie, nous mettrons en avant les éléments de la littérature qui favorisent la possibilité pour Posit de devenir le successeur du standard IEEE 754.

3.1 Couverture

Grâce à son implémentation, Posit permet de limiter les valeurs infinies en les remplaçant par des valeurs de saturation telles que Maxpos ou Minpos. À cet égard, nous avons examiné la couverture offerte par Posit, qui est souvent annoncée comme supérieure à celle du standard IEEE 754 pour certaines tailles, comme indiqué dans la référence Fig. 7, issue de [2].

Il est important de noter que, bien que Posit puisse offrir une meilleure couverture par rapport à IEEE 754, cela peut influencer sa précision. En effet, à mesure que la taille du format Posit augmente, sa précision autour de 1 peut se rapprocher de celle du format flottant du standard IEEE 754. Pour les besoins de notre analyse, nous avons choisi les configurations de Posit avec les exposants les plus équilibrés, c'est-à-dire ceux qui offrent un bon compromis entre la précision autour de 1 et une couverture étendue. Cette sélection vise à optimiser les performances tout en préservant une précision adéquate pour les applications ciblées.

Ce choix a été pris pour plusieurs raisons. Premièrement, il n'existe pas de version de Posit permettant d'utiliser un exposant dynamique. Secondelement, il n'y a qu'une seule version de Posit implémentée sur RacEr et pour la majorité des émulations, seul l'exposant équilibré est implémenté.

Size,Bits	IEEE Float Exp. Size	Approx. IEEE Float Dynamic Range	Posit es Value	Approx. Posit Dynamic Range
8	3	1/64 to 16	0	1/64 to 64
16	5	6×10^{-8} to 7×10^4	1	9×10^{-10} to 1×10^9
32	8	1×10^{-45} to 3×10^{38}	2	5×10^{-38} to 2×10^{37}
64	11	5×10^{-324} to 2×10^{308}	3	2×10^{-152} to 5×10^{151}
64			4	4×10^{-304} to 3×10^{303}

Tableau 3. – Tableau sur la zone de définition, extrait de « Posit Arithmetic » de John Gustafson [2].

3.2 Précision

En fonction de la manière dont les rôles sont agencés, il est possible d'obtenir soit une représentation des nombres sur une plage plus étendue, soit une précision significativement améliorée pour les valeurs proches de 1. Le tableau ci-dessous, extrait de « Posit Arithmetic [2] » par John Gustafson, met en avant comment Posit peut offrir un gain de précision potentiel sur une plage de valeurs définie. Cette visualisation permet de mieux comprendre les avantages en termes de précision et de couverture que Posit peut apporter, en fonction des configurations choisies.

Size,Bits	IEEE Float Max accuary bits	Posit Max accuary bits	Range where Posit accuracy is \geq Float accuracy
8	5	6	1/4 to 4
16	11	13	1/64 to 64
32	24	28	10^{-6} to 10^6
64	53	59	10^{-17} to 7×10^{16}

Tableau 4. – Tableau sur la précision

Ainsi, grâce aux deux parties précédentes, nous avons pu observer la zone de définition ainsi que la zone de précision accrue de celui-ci. Le graphique ,ci-dessous, représente la précision des nombres en comparant les implémentations de Posit<8,1> et de Float (8,4).

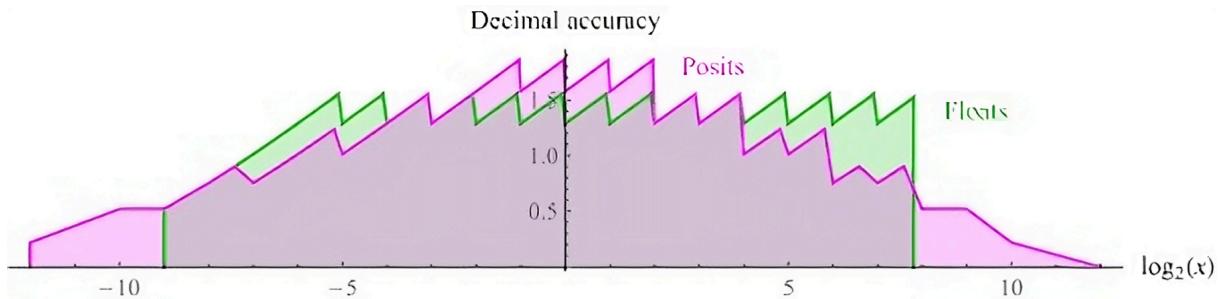


Fig. 7. – Représentation d'un Posit <8,1> et Float (8,4)

Comme le montre parfaitement le graphe, Fig. 7, la représentation de Posit est dites cônique. Cela signifie qu'il dispose d'une forte précision autour de 1 puis d'une décroissance de cette précision jusqu'à sa valeur MaxPos ou MinPos. On peut alors remarquer trois zones qui divisent les représentations de Posit et du standard IEEE 743.

La première concerne les valeurs en bordure de la zone de couverture. Posit, selon son exposant, peut permettre de définir des nombres plus éloignés. Dans le cas où il ne le peut pas, il arrondit ces valeurs à Maxpos ou Minpos. La seconde zone correspond à l'endroit où le standard est plus précis que Posit. Enfin, il reste la partie appelée « golden zone ». Dans cette zone, Posit est plus précis que le standard. Il est à noter que cette zone, centrée autour de 1, peut varier en importance en fonction de l'exposant de Posit.

3.3 Nan, underflow, overflow

La différence la plus marquante de Posit comparée au standard et aux alternatives, est sa possibilité de pouvoir s'abstraire des valeurs de dépassements de capacité (overflow), de sous-débordements (underflow) et les NaN (Not a Number). C'est valeurs ne sont pas représentées mais sont arrondies par Posit à la plus grande, ou la plus petite valeur représentable. C'est valeurs sont appellées « MaxPos » et « MinPos ». Cette absence de NaN simplifie les calculs et évite les comportements indéterminés dans les opérations arithmétiques, en garantissant toujours un résultat défini. Ceci étant, comme le signale l'article *Posits: the good, the bad and the ugly* [3], il est préférable d'avoir des dépassements de capacité qui soient clairement signalés par un infini plutôt qu'ils soient glissés sous le tapis par de l'arithmétique saturée.

3.4 Représentation sur l'anneau réel

La représentation en ringplot des Posits est une méthode visuelle pour comprendre comment les nombres Posit sont distribués suivant la valeur de leur bits, offrant ainsi une représentation intuitive de leur précision et de leur dynamique. Cette représentation nous permet de constater la richesse des Posits pour les nombres situés dans l'intervalle $[0,1]$, ce qui correspond à $[1: +\infty]$ dans le cas des Posits.

De plus, cette vue met en évidence la représentation symétrique des Posits. Ci-dessous est représenté un ringplot des Posits sur 4 bits.

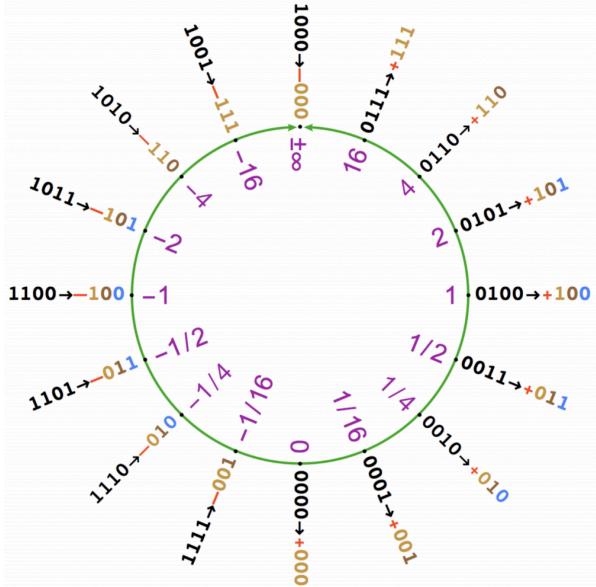


Fig. 8. – Ringplot de Posit 4bits, extrait de [2]

3.5 Posit sur quel matériel ?

Posit est un type de donnée comparable à « float », « int » ou « double », qui peut être intégré dans divers systèmes via l’émulation, y compris sur les GPGPU, les accélérateurs, ou les CPU. Toutefois, il est le plus souvent utilisé sur les FPGA (Field-Programmable Gate Arrays), car ceux-ci permettent une intégration native, évitant ainsi l’émulation et améliorant l’efficacité.

L’émulation des Posits peut se faire en utilisant des encodeurs/décodeurs entre le stockage des nombres au format Posit et les calculs en float/double, ce qui alourdit le temps de calcul. Une autre approche consiste à gérer directement les bits en binaire, de sorte que les calculs en Posit ne sont effectués qu’à l’utilisation de la variable. Cette méthode d’émulation sur des machines utilisant le standard IEEE 754 implique un surcoût de calcul, rendant difficile une comparaison directe des performances entre les formats. Par conséquent, nos comparaisons se concentrent sur la précision, la couverture et le coût de stockage. Pour évaluer les performances, nous avons réalisé un benchmark en créant et réduisant un million de valeurs, fournissant une estimation générale du temps de traitement.

Ainsi, pour la création des valeurs, nous utilisons la fonction `rand` fournie par la bibliothèque standard de C. Ces valeurs sont générées en format 32 bits afin d’avoir une entrée non identique pour Posit 32 bits et Float 32 bits.

```
float val = (float)rand()/(float)RAND_MAX * range;
```

Liste 1. – Génération des valeurs

La réduction effectuée correspond à l’accumulation d’un grand nombre de valeurs dans une seule.

```
for (int i = 0; i < n; i++)
    reduction += tableau[i];
```

Liste 2. – Calcul de la réduction

```

// Initialisation des valeurs
    Génération vers IEEE : 4271980.000000
    Génération vers Posit : 32886986.000000
    Conversion IEEE vers Posit : 20037265.000000

// Reduction des tableaux
    Somme des valeurs IEEE : 2305266.000000
    Somme des valeurs Posit : 52097284.000000

```

Liste 3. – Temps d’execusion en nsec pour un million de valeur, réalisé avec fastsigmoid [4]

Grâce à la littérature, nous avons identifié les domaines où Posit est supposé surpasser son équivalent dans le standard IEEE 754. Les points suivants seront donc ceux que nous vérifierons à travers les émulations de Posit ainsi qu’avec notre implémentation matérielle, RacEr :

- Adaptabilité de la précision grâce à la représentation dynamique
- Précision accrue autour de 1
- Zone de couverture plus étendue
- Élimination des overflow et underflow
- Réduction des NaN, simplifiée par un arrondi à Maxpos/Minpos
- Disponibilité potentielle de Posit sur tous les matériels

Il est tout de même important de noter que les représentations sur les avantages de Posit reposent sur des exemples permettant un gain vis à vis du standard. L’exemple Fig. 7 utilise une version particulière de Posit 8 ainsi que du FP8. Il serait tout à fait possible, par l’utilisation d’autres représentations, de montrer le FP8 avantageux par rapport au Posit 8.

4 Emulations de Posit

Ainsi, nous avons commencé par utiliser l'émulation pour émettre une première validation des Posits. Cette première partie, plus théorique, nous permettra de comprendre les véritables avantages des Posits par rapport au standard IEEE 754. Cela nous aidera à identifier les types d'applications qui pourraient en bénéficier. Nous avons sélectionné quatres émulations : Softposit, Ceralaine, Posit-GCC et FastSigmoid. Il faut noter que lors d'une émulation, le temps de calcul est fortement rallongé et est par conséquent moins efficace que sur un matériel implémentant la technologie nativement.

Comme expliqué précédemment, Liste 3, pour l'émulation de Posit, la majeur partie du temps de calcul se situe dans le décodage et l'encodage des données. Pour être traitée comme un Posit, la valeur d'entrée doit être convertie, puis traitée et enfin reconvertisse au format d'origine pour être lisible par le système lors de l'affichage. De plus, l'émulation ne permet pas un traitement fluide des données car le traitement d'une donnée au format IEEE diffère de celui d'un Posit.

4.1 SoftPosit

SoftPosit est une implémentation de la norme Posit développée par l'équipe « NGA Research ». Basée sur l'implémentation SoftFloat de l'université de Berkeley, elle permet de comparer les nombres Posit avec les nombres en format flottant (floats).

Certaines fonctionnalités facilitent l'intégration et la comparaison de la technologie Posit avec les formats de nombres flottants traditionnels. Cependant, SoftPosit prend en charge uniquement les versions Posit de 8, 16 et 32 bits. De plus, les niveaux d'exposants sont fixés, ce qui limite les options disponibles à trois versions spécifiques de Posit :

- **Posit<8,0>**
- **Posit<16,1>**
- **Posit<32,2>**

Ces limitations permettent une évaluation ciblée des performances et des caractéristiques de Posit dans des tailles de format bien définies, tout en simplifiant l'implémentation et l'utilisation dans des contextes spécifiques.

4.2 Ceralaine

Ceralaine est une implémentation dérivée de SoftPosit, étendant le support des versions Posit de 8 à 128 bits. Contrairement à SoftPosit, Ceralaine permet une gamme plus large de tailles de format, allant de **Posit<8,0>** à **Posit<128,4>**. Cependant, tout comme SoftPosit, Ceralaine utilise une seule configuration d'exposant par taille de format, ce qui limite les options disponibles à cinq versions spécifiques de Posit :

- **Posit<8,0>**
- **Posit<16,1>**
- **Posit<32,2>**
- **Posit<64,3>**
- **Posit<128,4>**

En tant qu'extension de SoftPosit, **Ceralaine** a été testé pour vérifier la concordance des résultats avec ceux obtenus par SoftPosit. Nos tests ont confirmé que les résultats fournis par Ceralaine sont identiques à ceux de SoftPosit, assurant ainsi une cohérence et une fiabilité dans les calculs entre les deux implémentations.

4.3 Posit-GCC

Posit GCC représente une approche légèrement différente des autres implémentations de Posit. Contrairement aux émulations précédentes, qui permettaient l'exécution parallèle des Posits et des

nombres flottants au format IEEE 754, Posit GCC intègre directement le traitement des Posits dans le processus de compilation. En utilisant un drapeau de compilation spécifique, Posit GCC remplace la gestion des types flottants standards (`float` et `double`) par des types Posit. Ainsi, le choix de traiter les nombres comme des Posits se fait au moment de la compilation, plutôt qu'en amont dans le code source.

Cette implémentation propose uniquement deux versions de Posit :

- **Posit<32,2>**
- **Posit<64,3>**

Ces versions remplacent respectivement les types `float` et `double` lors de la compilation. Le principal avantage de Posit GCC réside dans sa transparence d'utilisation : les types `float` et `double` sont directement interprétés comme des Posit 32 bits ou 64 bits, simplifiant ainsi l'intégration de Posit dans le code existant sans nécessiter de modifications supplémentaires.

4.4 Fastsigmoid

Enfin, **FastSigmoid** est une implémentation distincte qui ne repose pas sur SoftPosit, mais sur des nombres Unums de type III. Cette approche permet une plus grande flexibilité dans la gestion des versions de Posit. FastSigmoid supporte les formats Posit de 8, 16 et 32 bits, tout en exploitant la dynamique de l'exposant. En conséquence, il propose 10 versions différentes de Posit :

- **Posit<8,0> à Posit<8,2>**
- **Posit<16,0> à Posit<16,2>**
- **Posit<32,0> à Posit<32,3>**

Cette implémentation se distingue par ses fonctionnalités avancées, telles que la possibilité de désactiver la gestion de l'underflow ou des NaN (Not-a-Number). Le principal atout de FastSigmoid réside dans la grande diversité de versions de Posit qu'elle propose, permettant ainsi une meilleure prise en compte de la flexibilité de l'exposant par rapport aux autres implémentations. Cette diversité permet aux utilisateurs d'adapter les spécifications de Posit en fonction des besoins spécifiques des applications et des performances souhaitées.

4.5 Quel est l'impact de l'exposant ?

À travers la littérature, nous avons constaté que la représentation de Posit est dynamique, s'adaptant au nombre à représenter. Cependant, le paramètre de l'exposant doit être fixé lors de la définition de la variable. Comme mentionné précédemment, certaines implémentations de Posit n'incluent pas toutes les versions possibles des exposants. Il est donc crucial de comprendre pourquoi certaines implémentations choisissent de restreindre le nombre d'exposants, tandis que d'autres les incluent.

L'exposant détermine la zone de couverture de Posit. Plus l'exposant est grand, plus la plage de valeurs couvertes est étendue. Comme le montre le graphe ci-dessous, référence Fig. 9, la zone de couverture peut varier considérablement en fonction de l'exposant choisi. Il est également important de noter que, en plus d'affecter la couverture, l'exposant a un impact sur la précision des représentations. Le graphe Fig. 10 illustre les variations de précision autour de 1 en fonction de l'exposant sélectionné. Ces graphiques permettent de visualiser comment la flexibilité de l'exposant influence à la fois la couverture et la précision dans différentes configurations de Posit.

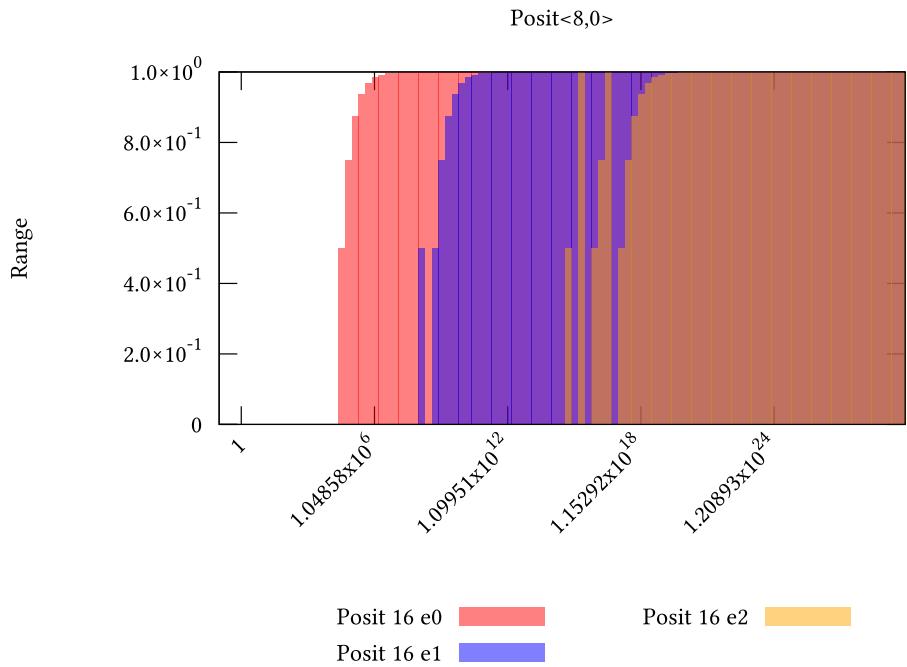


Fig. 9. – Mesure de la zone de couverture de Posit 16 en fonction de l'exposant

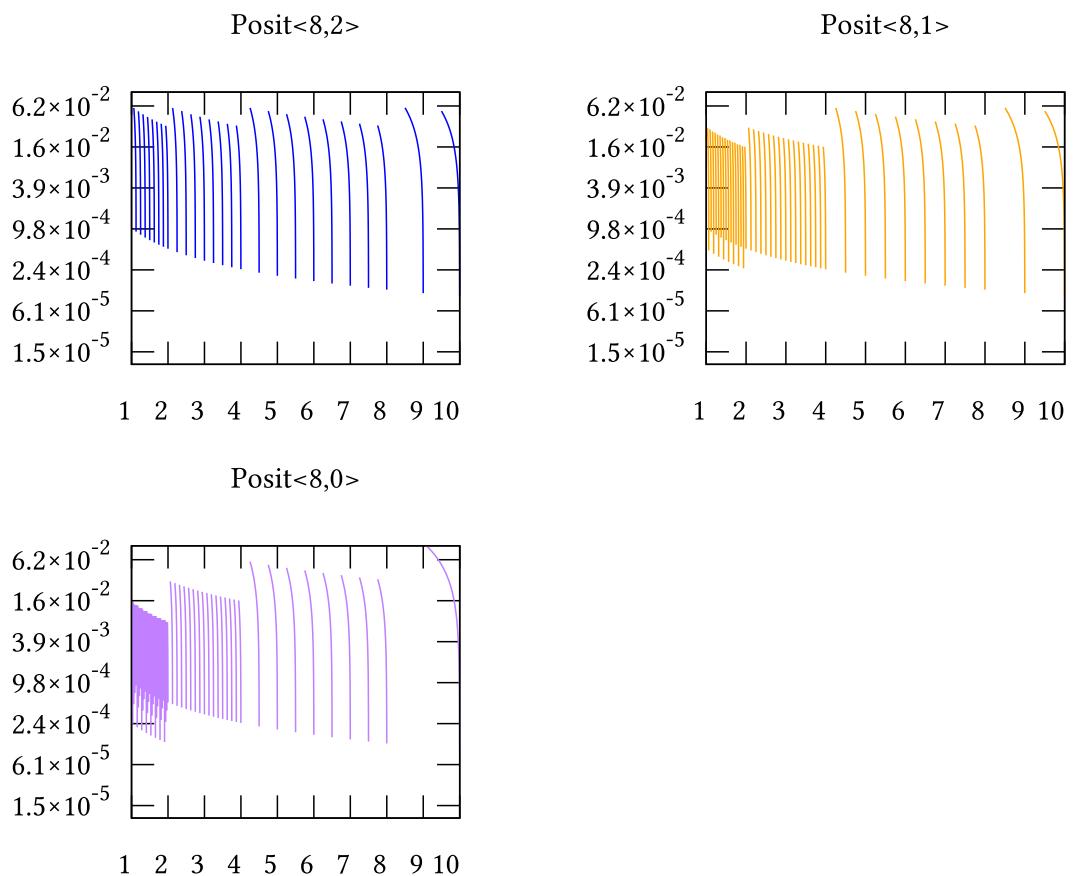


Fig. 10. – Mesure de la précision de Posit 8 en fonction de l'exposant entre 1 et 10

4.6 Validation des émulations

i Information

Il est important de souligner que nos mesures de précision montrent des variations en fonction des nombres à représenter. Aucune de ces mesures ne permet de garantir un avantage exclusif pour IEEE 754 ou Posit sur l'ensemble de l'espace de définition. De plus, nous n'avons pas réalisé de mesures temporelles en raison des différences d'implémentation entre le matériel natif du standard IEEE 754 et les implémentations émulées de Posit. Bien que la question de l'efficacité en terme de mémoire soit abordée, il convient de préciser que les gains d'espace mémoire se réfèrent uniquement à la taille de stockage des données, sans évaluer les implications sur la performance du traitement.

Dans un premier temps, nous avons dû vérifier si les implémentations existantes de Posit respectaient correctement les algorithmes définis pour ce format. Cette validation comprenait plusieurs points clés, notamment la gestion des underflows, des overflows et des valeurs NaN (Not-a-Number). Ces aspects devront également être vérifiés pour l'architecture RacEr.

De plus, nous avons évalué la précision des Posits dans la « golden zone » ainsi que la perte de précision en dehors de cette zone. Cette évaluation est essentielle pour comprendre les performances de Posit dans différents cas.

Enfin, nous avons examiné la facilité de portage des applications du standard IEEE 754 vers les formats Posit. Bien que cette partie soit importante pour les émulations, elle a été particulièrement pertinente pour l'architecture RacEr. Nous avons donc comparé l'implémentation matérielle de RacEr avec les émulations de Posit suivantes : Ceralaine, FastSigmoid et GCC-Posit.

4.6.1 NaN, under/overflow

D'après la littérature, Posit est souvent présenté comme le remplaçant idéal du standard IEEE 754. Cependant, en pratique, il existe des nuances qui en limitent la pertinence à un certain nombre d'applications. Par exemple, l'absence de NaN (Not-a-Number) dans Posit est conditionnelle, dépendant de la plage de données et de l'implémentation spécifique.

Pour évaluer la zone de définition des émulations Posit, nous avons testé des valeurs allant de 1 à 1×10^{50} . Selon la littérature, Posit<32,2> devrait pouvoir représenter des nombres jusqu'à 2×10^{37} . Au-delà de cette valeur, il est censé arrondir toutes les valeurs à son Maxpos, qui est de 2×10^{37} . Cependant, nos observations montrent des différences significatives entre les différentes implémentations :

- **Ceralaine** : Cette implémentation ne permet de représenter que des valeurs jusqu'à 1×10^{36} et renvoie un NaN pour des valeurs à partir de 1×10^{38} .
- **Posit GCC** : Les valeurs sont également limitées à 1×10^{36} , mais cette implémentation parvient à conserver cette valeur jusqu'à 1×10^{50} .
- **FastSigmoid** : Cette implémentation permet de représenter des valeurs allant jusqu'à 2×10^{37} , et, comme Posit GCC, elle conserve son Maxpos jusqu'à 1×10^{50} .
- **Softposit** : Cette implémentation ne permet de représenter que des valeurs jusqu'à 1×10^{36} et renvoie un NaN pour des valeurs à partir de 1×10^{38} .

Concernant le standard IEEE 754, les graphiques montrent que, pour des valeurs au-delà de 1×10^{38} , les nombres flottants (floats) passent à l'infini. Par conséquent, aucune valeur ne peut être représentée au-delà de ce seuil dans ce format.

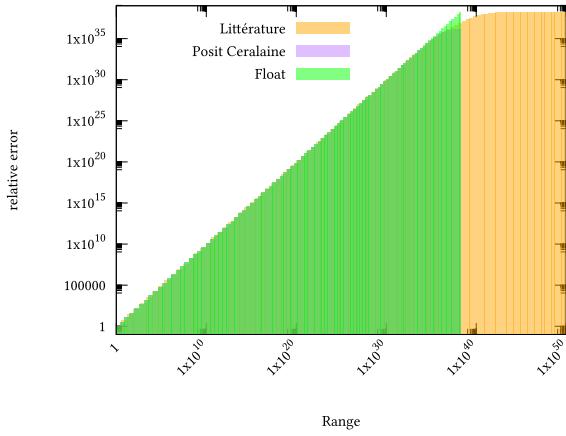


Fig. 11. – Représentation des nombres pour l’émulation Ceralaine

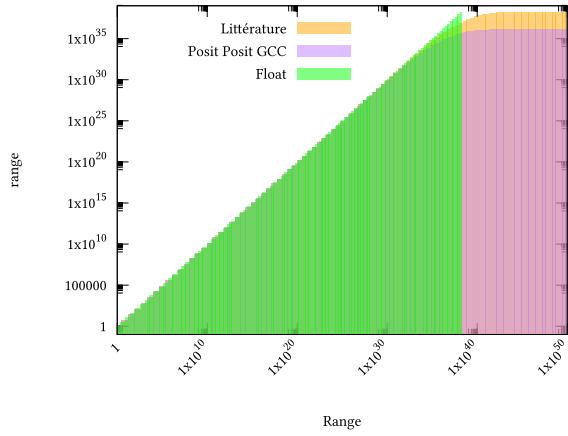


Fig. 12. – Représentation des nombres pour l’émulation posit gcc

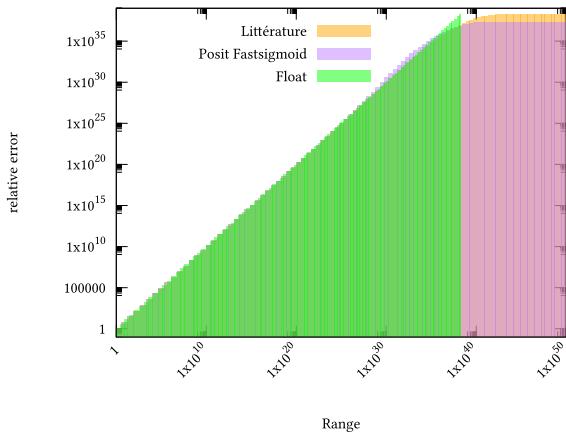


Fig. 13. – Représentation des nombres pour l’émulation fastsigmoid

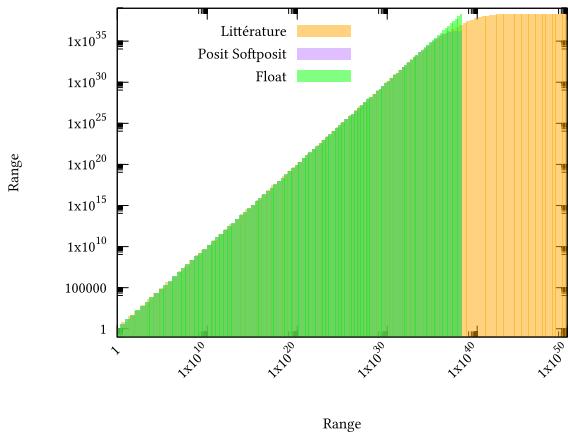


Fig. 14. – Représentation des nombres pour l’émulation Softposit

Nous constatons une différence nette entre les diverses implémentations de Posit lors de cette première vérification. Il est important de noter que l’implémentation de **SoftPosit** est identique à celle de **Ceralaine**.

Posit montre des avantages principalement avec les émulations de **FastSigmoid** et **Posit GCC**. En particulier, **FastSigmoid** est la seule implémentation à offrir une zone de couverture conforme aux attentes décrites dans la littérature.

La gestion des valeurs hors définition dans Posit permet d’éviter les comportements imprévisibles en remplaçant les valeurs non représentables par des valeurs arrondies plutôt que par des NaN ou des infinis. Cependant, cette approche dite « silencieuse » d’arrondir les valeurs peut masquer des erreurs de calcul, ce qui pourrait entraîner des résultats incorrects ou inattendus dans certaines situations.

4.6.2 Précision

La littérature indique que Posit<32,2> offre une précision particulièrement avantageuse pour les valeurs situées entre 1×10^{-6} et 1×10^6 (voir Tableau 4). Pour vérifier ces affirmations, nous avons conçu des tests pour évaluer plusieurs aspects :

- **La précision au-delà de 1×10^6 et en dessous de 1×10^{-6}** , où Posit est censé être équivalent ou moins performant.
- **La précision dans l’intervalle de 1×10^{-6} à 1×10^6** , où Posit devrait être meilleur ou équivalent au standard.

- **La précision autour de 1**, où Posit est supposé offrir une performance significativement supérieure, en validant aussi la limite.

Nous présentons ici uniquement les résultats de l'émulation de **FastSigmoid**. Les résultats pour les autres émulations sont disponibles en annexe : Chapitre 10.2.

i Information

Pour évaluer la précision, nous avons choisi de calculer l'erreur relative entre Posit et Float, en utilisant le double comme référence de la valeur « juste ». L'erreur relative a été calculée à l'aide de la formule suivante :

$$\text{err} = \frac{\text{Posit} - \text{Double}}{\text{Double}}$$

Cette approche permet de quantifier la différence entre les représentations Posit et Float par rapport à la précision offerte par le double.

4.6.2.a Impact sur les très grands nombres

Dans cette section, nous allons approfondir l'analyse en complétant les résultats du Chapitre 4.6.1, qui met en évidence la zone de définition de Posit<32,2>. La littérature ne signale pas de perte de précision pour les grands nombres, soulignant plutôt une précision élevée pour les valeurs proches de 0 et les grandes valeurs. Cependant, comme le montre le graphe ci-dessous, pour des valeurs dépassant la zone de $\pm 2 \times 10^7$, la précision de Posit<32,2> chute drastiquement. On parle ici de la perte d'un chiffre significatif.

Le seul avantage significatif de Posit pour les grands nombres apparaît lors d'un dépassement de la zone de définition. Dans ce cas, comme illustré au Chapitre 4.6.1, Posit arrondit les valeurs à Maxpos au lieu de renvoyer une valeur infinie, ce qui permet d'éviter les erreurs liées aux dépassements de capacité.

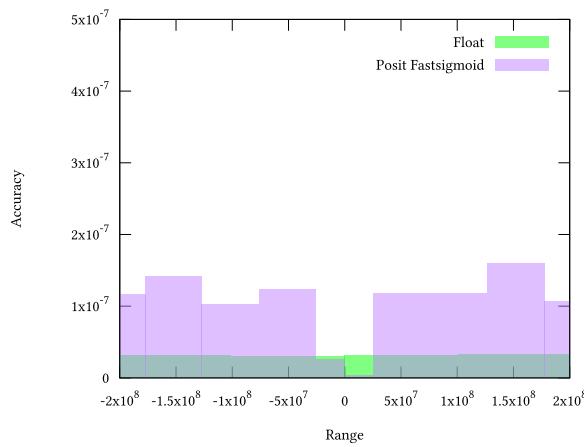


Fig. 15. – Emulation fastsigmoid sur -2×10^8 à 2×10^8

4.6.2.b Impact sur les nombres au centre

Comme indiqué dans la littérature, Posit<32,2> offre une précision supérieure à celle de son équivalent en floating point du standard IEEE 754, Float, dans la plage comprise entre -1×10^6 et 1×10^6 . Nous avons donc effectué des mesures sur des valeurs situées dans cet intervalle. Les résultats confirment les affirmations de la littérature : en moyenne, Posit se révèle effectivement plus précis dans cette plage.

Cependant, il est important de noter que ce gain de précision devient presque négligeable à partir de $\pm 1 \times 10^5$.

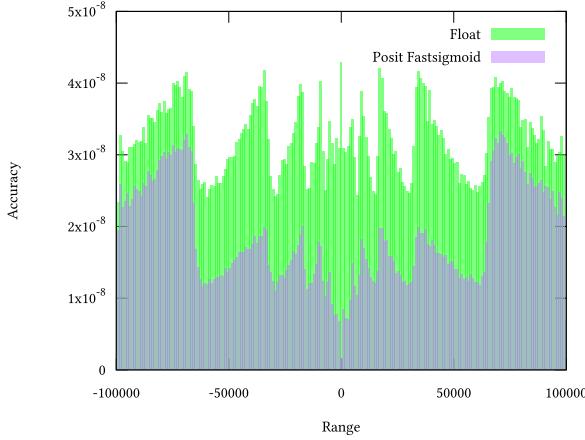


Fig. 16. – Emulation de Fastsigmoid sur -1×10^5 à 1×10^5

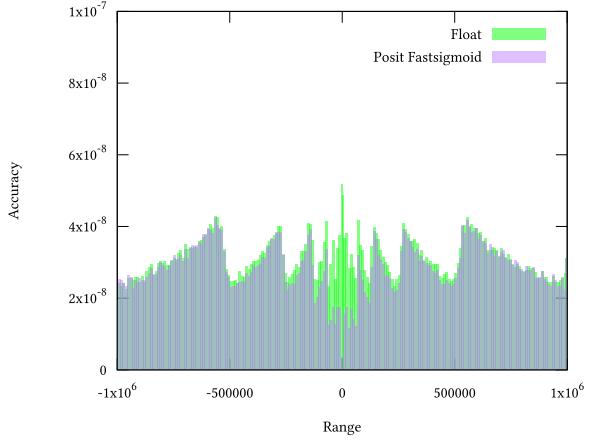


Fig. 17. – Emulation fastsigmoid sur -1×10^6 à 1×10^6

4.6.2.c Impact sur les nombres très petits

Pour la suite de nos analyses, nous nous sommes concentrés sur la précision conique de Posit, qui se caractérise par une plus grande précision autour de 0 et une précision décroissante au-delà d'une certaine valeur.

Dans l'intervalle $[-500 : 500]$, Posit démontre un avantage notable par rapport au format Float du standard IEEE 754, avec un gain de précision pouvant atteindre un facteur 10, soit un chiffre significatif supplémentaire. Cependant, la littérature souligne que, en deçà d'une certaine valeur, la précision de Posit devient comparable, voire inférieure, à celle du standard. Nous avons testé cette limite, annoncée à 1×10^{-6} pour la version Posit<32,2>, et nos mesures indiquent une démarcation plutôt autour de 1×10^{-5} pour l'implémentation émulée avec Fastsigmoid [4].

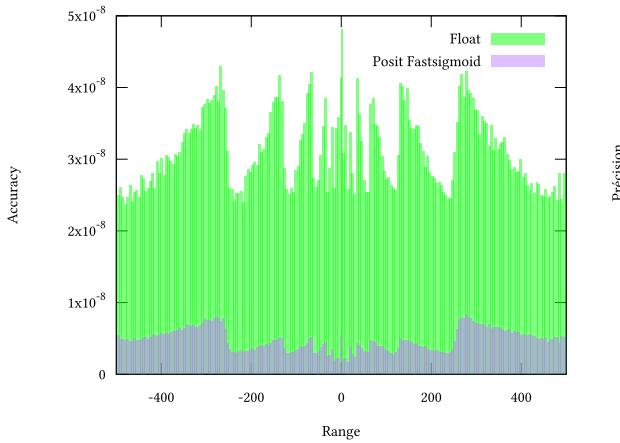


Fig. 18. – Emulation de Fastsigmoid autour de 0

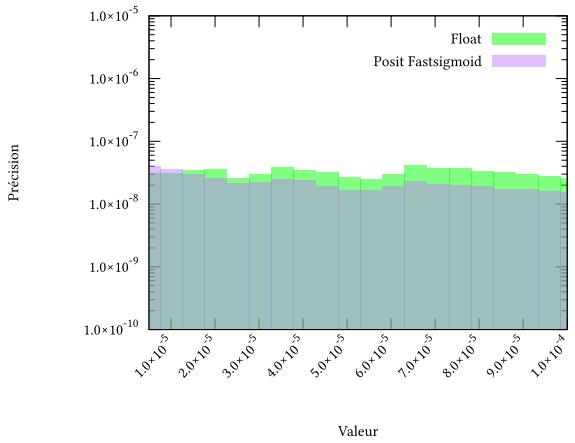


Fig. 19. – Limite de précision minimale de Posit 32 es 2 de Fastsigmoid

Comme le montrent les graphes, toutes les implémentations confirment un gain de précision de Posit pour des valeurs comprises entre -1×10^6 et 1×10^6 . Notamment, un gain d'un facteur 10 est observé pour les nombres calculés avec Posit autour de 1, dans l'intervalle $[-500 : 500]$. Il est également important de noter que les valeurs obtenues par les différentes émulations sont identiques pour toutes les versions de Posit.

Cependant, il est crucial de souligner la perte de précision de Posit en dehors de cette zone. Cette diminution, visible sur le graphe pour des valeurs allant de -1×10^9 à 1×10^9 , est d'une ampleur comparable au gain de précision offert par Posit dans la plage centrale. Au-delà de cette zone, les erreurs peuvent même s'aggraver, compromettant ainsi les avantages potentiels de Posit.

5 Orientation GPGPU / Autres alternatives au standard

5.1 CUDA

CUDA est une technologie propriétaire développée par NVIDIA qui permet de programmer sur des GPGPU (General-Purpose Graphics Processing Units). Un GPGPU est un GPU optimisé pour le calcul général, capable d'exécuter une large gamme d'algorithmes non graphiques, tels que l'apprentissage automatique, la simulation scientifique, et d'autres applications nécessitant des calculs massivement parallèles.

Étant donné que RacEr est une implémentation de Posit sur une architecture GPGPU, il est logique de la comparer à une implémentation CUDA utilisant le standard IEEE 754. Nous nous attendons donc à ce que les précisions de CUDA soient similaires à celles obtenues avec le standard IEEE 754 sur CPU, ce qui nous offre un point de comparaison pertinent pour évaluer les avantages potentiels de RacEr.

Nous avons entrepris de mesurer la précision de CUDA sur les mêmes points abordés par les émulations. Ces mesures ont été réalisées sur des P100. Nous avons ensuite vérifié les valeurs obtenues et les avons comparées aux résultats obtenus avec l'émulation Ceralaine, celle qui sera utilisée sur RacEr. Comme nous pouvons le constater, les écarts entre CUDA et Ceralaine sont cohérents avec les prévisions de Posit. Nous avons également validé les valeurs de CUDA en les comparant à celles collectées sur CPU, et ces deux ensembles de mesures sont identiques.

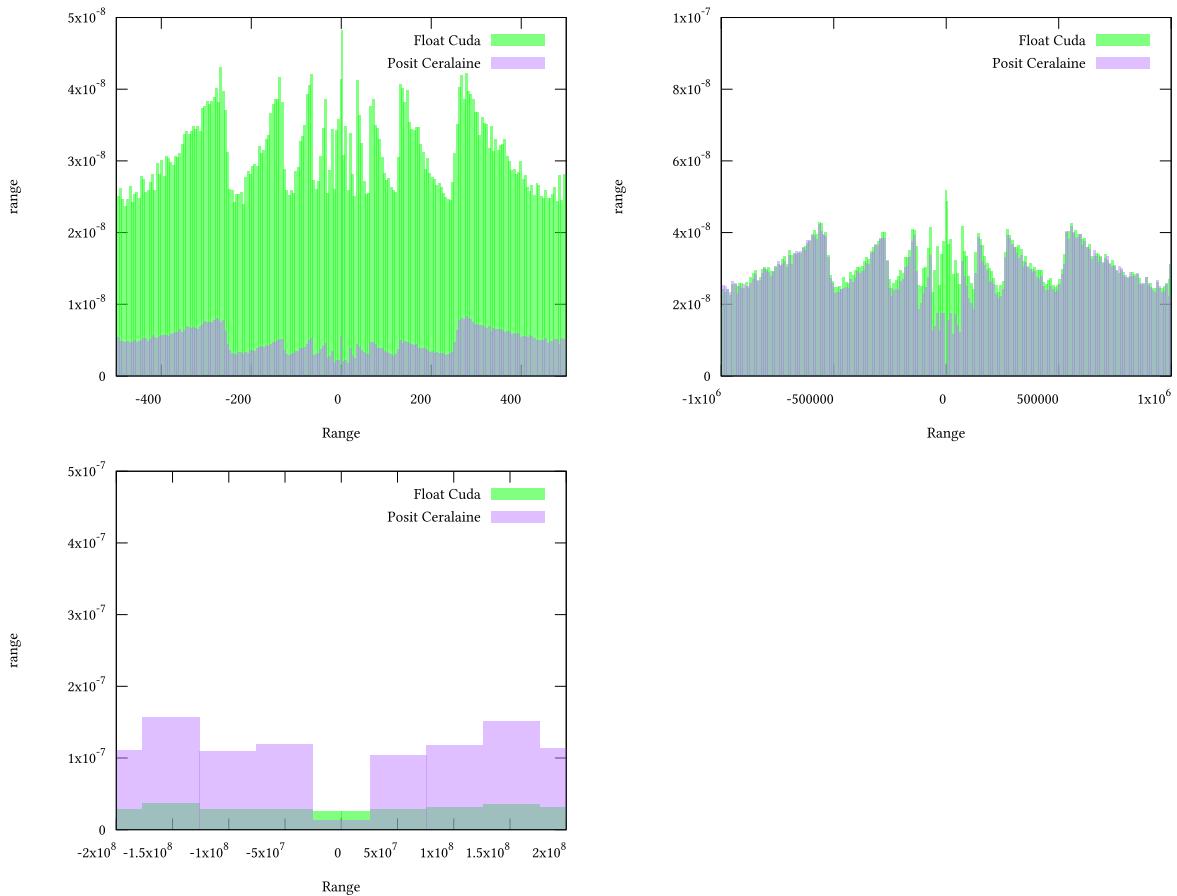


Fig. 20. – Opposition des mesures obtenues sur GPU CUDA et Posit Ceralaine

5.2 Représentation sur 16 bits

5.2.1 Tensor float 32 & Posit 16

Pour valider l'utilité de Posit, il est essentiel que la précision soit au moins équivalente à celle des autres formats pour un nombre de bits donné. Dans ce cadre, nous avons décidé de comparer sur différents points les implémentations de Posit, la version 16 bits avec plusieurs exposants, et le TensorFloat-32 [5] qui utilise 19 bits.

Parmi les implémentations à notre disposition, seule celle de Fastsigmoid permet d'utiliser des Posits avec une taille d'exposant variable. Nous comparerons donc les versions Posit 16 bits avec 0, 1 et 2 bits d'exposant avec le TensorFloat-32. Cependant, nous pouvons affirmer que les autres implémentations disposant d'une implémentation de Posit 16, c'est à dire Ceralaine et Softposit, ont une version équivalente à celle de Posit<16,1>(voir Fig. 46 en annexe). Posit GCC ne dispose d'aucune version avec 16 bits.

Pour les résultats suivants, nous avons donc utilisé uniquement l'implémentation de FastSigmoid, qui offre la possibilité d'ajuster le nombre d'exposants pour la version Posit 16 bits.

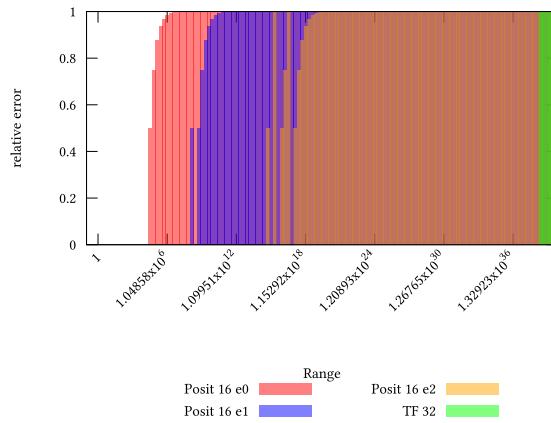


Fig. 21. – Comparaison de la zone de couverture de Posit 16 et TensorFloat 32

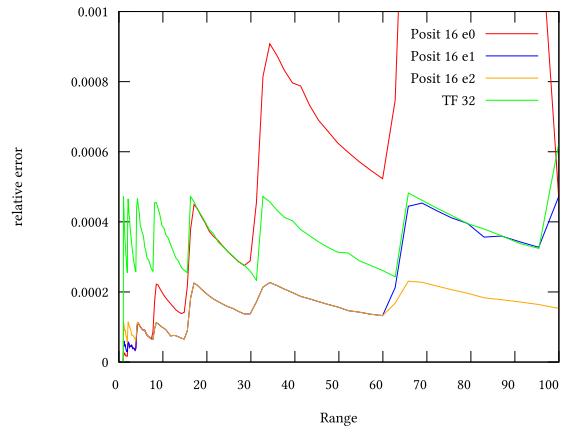


Fig. 22. – Comparaison de la précision de Posit 16 et TensorFloat 32

Comme le montrent les graphiques ci-dessus, la couverture du TensorFloat-32 est nettement plus étendue que celle des implémentations en Posit 16 bits. Cependant, il est important de noter que le Posit 16 bits offre une meilleure précision pour les valeurs proches de 1, particulièrement avec une version utilisant un exposant de 1 ou 2. Cela se traduit par un gain de précision dans l'intervalle $]0 \simeq 100]$ pour l'implémentation classique soit Posit 16 bits avec 1 bit d'exposant.

Cependant, aucune des implémentations de Posit ne parvient à couvrir l'intégralité de la plage du TensorFloat-32. En comparant la version de Posit 16 bits offrant la meilleure couverture, c'est-à-dire celle avec un exposant de 2, on observe une différence significative. Posit ne parvient plus à représenter les nombres aux alentours de $1e^{18}$, tandis que le TensorFloat-32, avec une plage de couverture comparable à celle du float, s'étend jusqu'à plus de $1e^{38}$.

5.2.2 Bfloat 16 & Posit 16

Pour la comparaison avec le BFloat16, nous avons choisi de le comparer directement avec TensorFloat-32, Posit<16,1>, et Float-32 du standard IEEE 754. Comme attendu, les graphiques révèlent que BFloat16 se distingue principalement par sa zone de couverture étendue. Toutefois, son implémentation qui n'atteint pas la précision de Posit, ni celle du TensorFloat-32. Malgré cela, BFloat16 parvient à couvrir une zone aussi large que le Float-32 tout en utilisant seulement 16 bits, avec une perte

de précision d'environ un demi-chiffre significatif par rapport au TensorFloat-32. Malheureusement, son implémentation sur 16 bits montre son défaut pour la précision des valeurs autour de 1 mais reste stable sur l'ensemble du domaine qu'il définit.

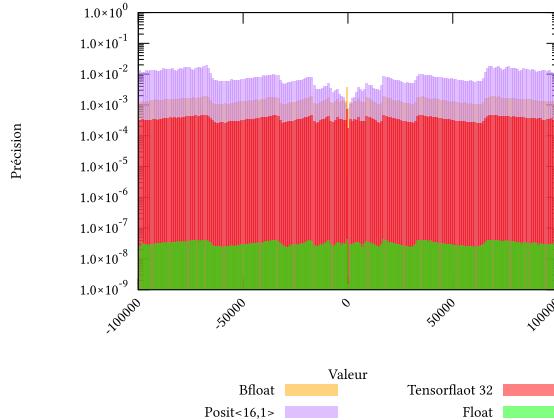


Fig. 23. – Comparaison de la zone de couverture de Posit 16 et Bfloat 16

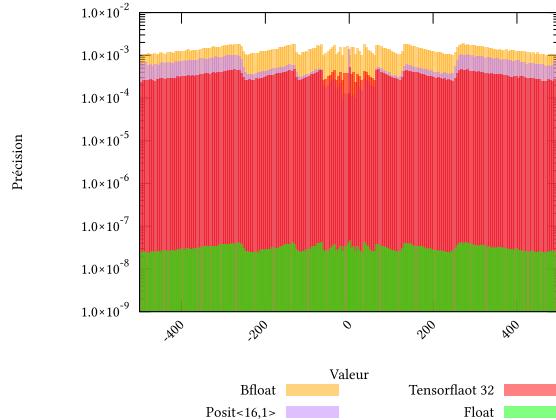


Fig. 24. – Comparaison de la précision de Posit 16 et Bfloat 16

Comme nous l'avons observé dans cette section, la différence de précision reste constante quel que soit le matériel utilisé. Ainsi, nous pouvons affirmer que la précision obtenue via une émulation sur CPU serait équivalente sur un GPGPU, et qu'une implémentation matérielle de Posit sur FPGA devrait être comparable à une implémentation matérielle sur GPGPU.

De plus, nous avons démontré que Posit n'est pas la seule alternative viable pour compléter ou remplacer le standard IEEE 754. En effet, comme le montrent TensorFloat-32 et BFloat16, chaque format est conçu pour répondre à des besoins spécifiques et possède des tailles différentes. Par conséquent, aucun de ces formats n'est globalement supérieur aux autres ; le choix du format dépendra des exigences particulières de chaque application.

5.3 Représentation sur 8 bits

Dans cette partie, nous aborderons les différentes représentations sur 8 bits qui peuvent exister. Notons que ces implémentations ne concernent pas exclusivement des nombres à virgules flottantes. La représentation sur 8 bits permet de répondre à un problème autour du domaine de l'IA. Durant le stage, nous n'avons pas mis en place d'application permettant de pouvoir valider la pertinence applicative des versions de Posit 8 bits. Cependant, à travers la littérature, nous avons pu noter certains points prouvant un gain possible grâce à la représentation Posit. Quelques essais ont montré, par le biais de l'utilisation de Posit 8 bits et 10 bits, un gain permettant l'obtention de résultats équivalents à un format 32 bits du standard IEEE 754. (article : *Deep Learning with approximate computing: an energy efficient approach* [6])

5.3.1 Posit 8 et FP8

N'ayant pas de version de Posit implémentée avec 10 bits, nous avons choisi d'utiliser FastSigmoid, qui permet de configurer une version 8 bits avec un exposant variant entre 0 et 2.

Pour la représentation des FP8, plusieurs variantes existent, tout comme pour Posit, car ce format n'est pas encore standardisé. Nous avons utilisé deux implémentations qui permettent, comme pour Posit, de choisir entre une plus grande zone de couverture ou une meilleure précision autour de 1.

L'implémentation de cette version a été fournie par umangyadav [7]. Elle implemente les versions suivantes :

- FP8e5m2 avec 5 bits d'exposant et 2 bits de mantisse
- FP8e4m3 avec 4 bits d'exposant et 3 bits de mantisse

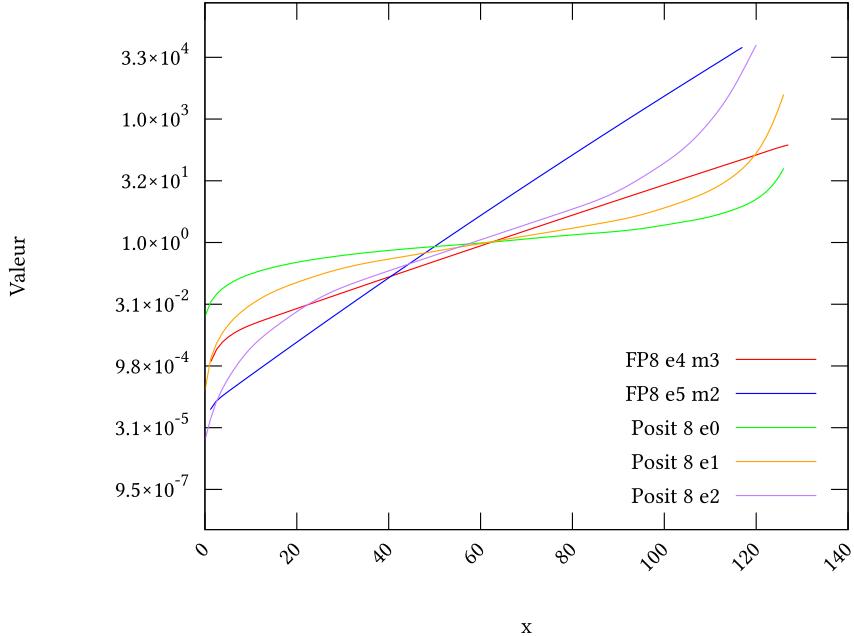


Fig. 25. – Comparaison des représentations 8 bits pour les valeurs positives (128 valeurs)

Les trois versions de FastSigmoid offrent des possibilités et des réponses à des besoins bien différents. Comme on a pu le voir avec Fig. 25, la version Posit<8,2> permet d'obtenir une zone de couverture plus importante que les versions de FP8. La chose la plus marquante sur le graphe est la linéarité des représentations FP8 et la représentation conique des versions de Posit.

Pour ce qui est d'une précision importante entre 0 et 1, Posit<8,0> reste bien meilleure que la version la plus précise de FP8 (4,3) avec Posit<8,0> qui à plus de la moitié de ses valeurs représentées entre 0 et 1.

5.3.2 INT 8

Certaines implémentations ont également permis, dans un souci de gain d'espace et de temps, de transformer la représentation des nombres en virgule flottante en entiers. Cette méthode, appelée quantification, permet de convertir les nombres à virgule flottante en entiers tout en conservant leur plage de définition. Plusieurs techniques de quantification existent, selon la nécessité de représenter ou non des nombres négatifs.

Il faut tout d'abord prendre la valeur dite maximale absolue, `amax`. Grâce à cette valeur, nous pouvons calculer le pas pour la représentation entre chaque valeur de INT8 `scale`. Enfin il ne nous reste plus qu'à convertir une valeur Float, x_f en INT 8, x_q .

$$\text{amax} = \max(|x_f|)$$

$$\text{scale} = \left(\frac{2 * \text{amax}}{256} \right)$$

$$x_q = \text{clip}\left(\left\lfloor \frac{x_f}{\text{scale}} \right\rfloor\right)$$

Comme le montre le graphique ci-dessous (Fig. 26), la représentation en INT8 présente à la fois des avantages et des inconvénients. D'une part, contrairement aux représentations en virgule flottante, elle offre une zone de couverture modulable en fonction des valeurs spécifiques de l'application ciblée. D'autre part, cette flexibilité implique que la précision est directement déterminée par la valeur maximale choisie, ce qui peut rendre la représentation INT8 inadaptée selon la plage de valeurs à représenter. Cependant, si ces valeurs sont uniformément réparties sur l'intervalle ($[-amax, amax]$), la représentation INT8 assure alors une couverture très précise.

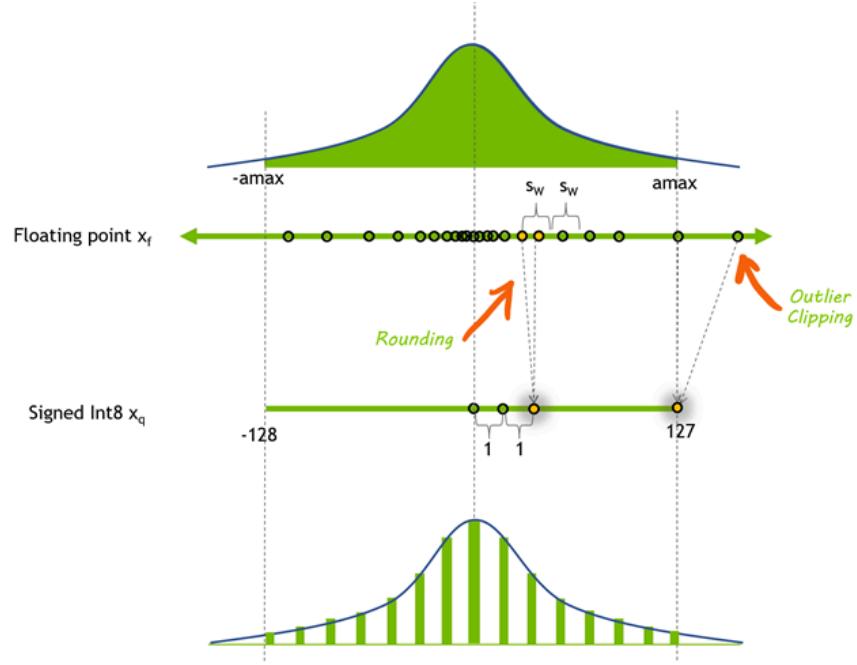


Fig. 26. – Quantification des floats vers int 8, source: NVIDIA

Un exemple illustrant la différence de représentation est disponible en annexe (voir Fig. 45).

À travers les parties précédentes, nous avons exploré diverses alternatives au standard IEEE 754 qui peuvent concurrencer Posit. Nous avons constaté que Posit est une solution avantageuse principalement lorsqu'elle est utilisée dans sa « golden zone », c'est-à-dire dans des plages de valeurs spécifiques où ses bénéfices sont maximisés.

Les alternatives proposées par NVIDIA, telles que TensorFloat-32, FP8 et INT8, répondent également à des besoins particuliers. Cependant, comparé à ces trois implémentations, Posit semble offrir des gains pour des applications comprennant des valeurs centrées sur 1 avec des extrêmes éloignées. Cependant, pour des équivalents FP32 avec des valeurs réparties sur la zone de couverture, il est souvent plus pertinent de choisir TensorFloat 32.

6 Posit implémentation matériel : RACER

À travers les parties précédentes, nous avons démontré l'intérêt que peut susciter Posit grâce à sa représentation numérique. Maintenant que nous comprenons ses limites théoriques et ses performances en émulation, nous allons valider son implémentation sur FPGA. RacEr, basé sur une architecture GPGPU de type CUDA, devrait offrir une précision indépendante de l'architecture utilisée, comme nous l'avons observé dans nos expérimentations antérieures. Par conséquent, nous nous attendons à des résultats similaires entre RacEr et les versions émulées.

L'implémentation de RacEr repose sur celle de Ceralaine, intégrant la majorité des opérations standards décrites dans le Chapitre 4.2. Cependant, il est important de noter que RacEr prend en charge uniquement une version spécifique de Posit : la version Posit<32,2>.

6.1 Architecture

D'un point de vue architectural, RacEr est construit de manière similaire aux architectures CUDA. Il est conçu pour supporter jusqu'à 512 coeurs de calcul Posit. Pour représenter ces coeurs de manière efficace, l'architecture de RacEr est organisée en plusieurs niveaux :

- **Les grilles**, qui représentent la plus grande entité virtuelle.
- **Les blocs**, qui sont des unités intermédiaires.
- **Les tuiles**, qui regroupent les unités de calcul.
- **Les unités de calcul** ou thread, les composants de calcul de base.

Cette architecture est très similaire à l'architecture Pascal de NVIDIA. Durant le stage, nous avons eu accès à un serveur AWS qui nous a permis de valider l'implémentation de RacEr. La machine mise à notre disposition nous a permis d'exécuter des opérations avec 4 unités de calcul réparties sur une seule tuile. De plus, un graphique montrant l'architecture d'un niveau puce et IO est disponible en annexe au Chapitre 10.3. Les puces CPU représentent une tuile comportant 4 threads.

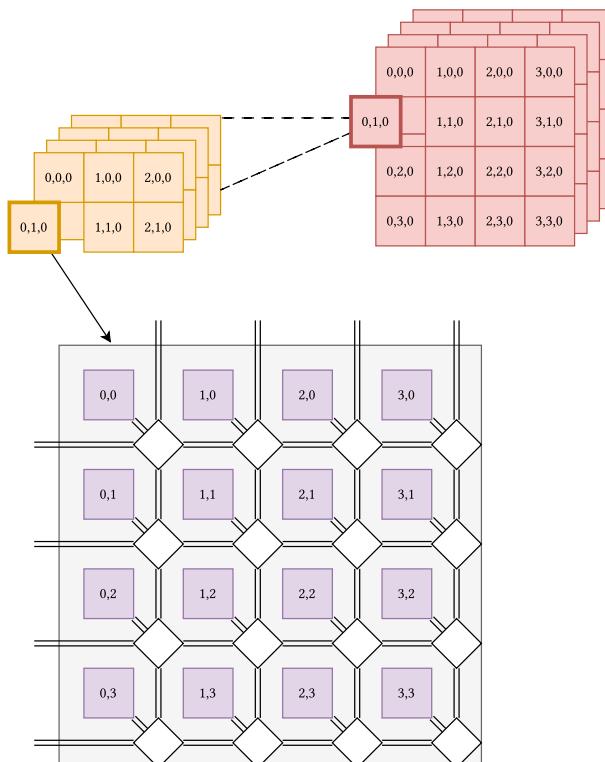


Fig. 27. – Schéma de l'architecture RacEr

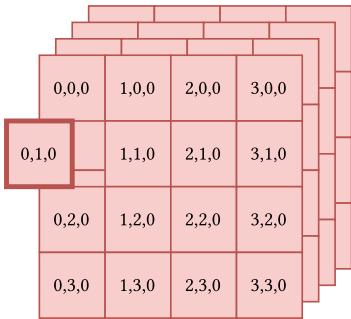


Fig. 28. – Représentation de la grille

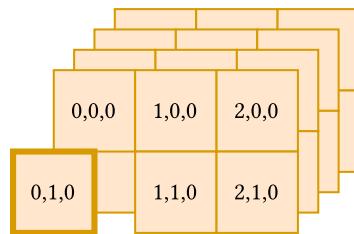


Fig. 29. – Représentation des blocs

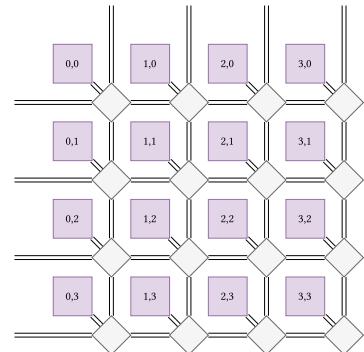


Fig. 30. – Représentation des tuiles

6.2 Compilation

Comme nous l'avons vu, l'architecture de RacEr suit les principes d'une architecture GPGPU, mais elle diffère de CUDA sur plusieurs aspects. Alors que CUDA facilite le développement rapide d'applications grâce à son écosystème intégré autour de C/C++, RacEr demande une attention particulière au développement, en raison de sa complexité accrue.

Cette complexité se manifeste notamment lors de la compilation d'une application, qui nécessite deux fichiers distincts : l'un pour le cœur de calcul exécuté sur le device (le kernel) et l'autre pour la partie main exécutée sur l'hôte (le CPU).

- Kernel : Le code du kernel doit être placé dans le répertoire « RacEr_manycore/software/spmd/RacEr_cuda_runtime_lite/ » sous un dossier spécifique. Le choix du nom du fichier est libre, mais il doit inclure un Makefile et un fichier main.c préparé à l'avance. Le nom du kernel doit être correctement référencé dans le Makefile, qui doit être configuré pour une architecture RISC-V.
- Main : Le fichier de la partie main doit être placé dans le répertoire « RacEr_f1/regression/cuda ». Pour que le test soit inclus dans le processus de compilation, il est également nécessaire d'ajouter le nom du fichier de test (sans l'extension .c) dans le fichier tests.mk.

Ainsi, contrairement à CUDA, le développement avec RacEr nécessite une configuration précise et une gestion rigoureuse des fichiers pour assurer une compilation correcte.

Par défaut, les fichiers suivent la convention de nommage suivante :

- Fichier principal (main) : `test_operation.c`
- Kernel :
 - Nom du dossier : `operation/`
 - Nom du fichier : `kernel_operation.c`

L'implémentation de RacEr est conçue pour faciliter une transition rapide depuis du code CUDA. La partie main nécessite uniquement quelques modifications mineures, telles que le changement de noms de fonctions et l'ajout de descriptions du matériel utilisé. Cette adaptation est presque transparente pour l'utilisateur, permettant une migration rapide et fluide vers l'utilisation de Posit.

En revanche, les fonctions kernel dans RacEr nécessitent une attention plus approfondie en raison de différences fondamentales par rapport à CUDA.

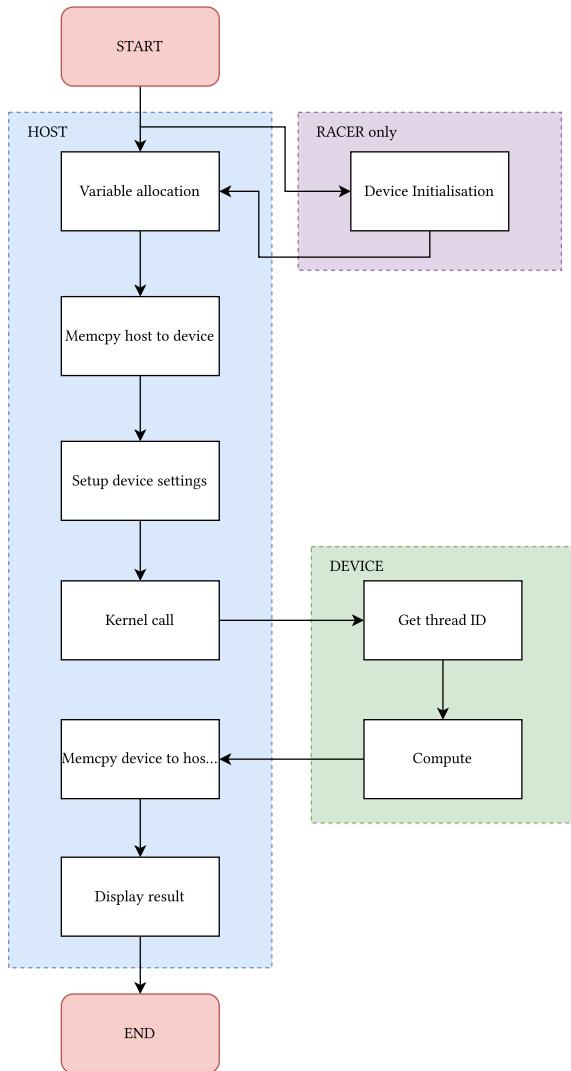
6.2.1 Approche CUDA

CUDA adopte une approche de programmation à un niveau très granulaire. Chaque kernel est exécuté par un grand nombre de threads, chacun ayant une vue locale de la mémoire et des ressources. Le

code kernel est écrit avec une perspective qui considère le traitement parallèle et la gestion des threads individuels.

6.2.2 Approche RacEr

À l'inverse, RacEr adopte une approche plus abstraite et moins centrée sur les détails des threads individuels. RacEr s'inspire des architectures CPU en fournissant une vue plus globale du calcul. Les kernels sont décrits à un niveau d'abstraction supérieur, se concentrant sur les opérations à un niveau de parallélisme plus large plutôt que sur la gestion des threads individuels. Cette approche est plus proche de la façon dont les CPU traitent les instructions, ce qui peut simplifier la conception des algorithmes mais nécessite une adaptation du code kernel écrit initialement pour CUDA.



Ce graphique décrit le flux de développement pour les deux architectures, RacEr et CUDA. Le flux est très similaire, les principales différences concernent le noyau, l'initialisation des périphériques et la compilation.

Comme le montre clairement le graphique, RacEr nécessite une étape supplémentaire. Cette étape nous permet de spécifier plusieurs informations telles que l'identifiant du périphérique, le chemin du noyau et l'allocateur utilisé.

Vous pourrez trouver en annexe, un exemple de portage sur une application de reduction vectorielle. Tous les points différents entre CUDA et RacEr y sont mentionnés et explicités, voir le Chapitre 10.4 en annexe. Il y a également un manuel expliquant les fonctions que nous avons pu référencer pour les besoins du portage des applications au Chapitre 10.5.

6.3 Problèmes rencontrés

L'approche que nous avons adoptée a été de convertir des applications de plus en plus complexes afin de pouvoir identifier les points manquants dans le portage. Il a également été important d'estimer le temps nécessaire pour le portage afin de déterminer si le coût humain ne serait pas démesuré pour certaines applications, en fonction du gain potentiel en précision. Il faut noter qu'il y a eu un manque assez important d'informations pour que le portage soit simple et rapide. Il nous a d'abord fallu retrouver, par la relecture de certains exemples, des fonctions d'initialisation et d'allocation mémoire. Une fois l'ensemble des fonctions retrouvées, le cœur de calcul nous a posé problème en raison de ses définitions d'indices dans chaque boucle parallèle. Enfin, le problème majeur fut la compilation, qui nécessitait

des fichiers bien nommés et bien placés, sans aucune liberté possible. Compte tenu de la proposition d'une API de type CUDA, nous pensions qu'il ne nous faudrait que deux jours maximum pour réaliser le portage d'une application simple. C'est finalement plus de deux semaines qui auront été nécessaires. De plus, lors des sessions de débogage, nous avons pu remarquer une erreur avec certaines fonctions de débogage, notamment dans l'affichage de valeurs et dans les valeurs des indices d'une boucle.

6.4 Résultats

Nous avons commencé à effectuer différentes mesures afin de valider l'implémentation matérielle de RacEr. Comme observé précédemment, cf. Chapitre 4, les implémentations de Posit peuvent varier et ainsi fournir des résultats différents. RacEr utilise l'implémentation de Ceralaine, voir Chapitre 4.2, pour la description de ses Posits.

Cependant, à l'heure actuelle, seule la version Posit<32,2> est implémentée et fonctionnelle sur RacEr. Ce choix est jugé correct d'après les mesures effectuées sur l'émulée de Ceralaine et la littérature, mais il pourrait présenter quelques lacunes. La représentation des nombres en Posit semble correcte avec l'émulée, mais l'arrondi en dehors de la zone de couverture mérite une attention particulière lors de la vérification de cette implémentation sur le matériel RacEr.

Nous avons pu voir une évolution dans l'implémentation de RacEr. En effet, lors de nos premières mesures, nous avons remarqué une perte significative de précision numérique de RacEr. Nous nous attendions à obtenir des résultats équivalents à l'émulée de Ceralaine, Chapitre 4.2.

6.4.1 Précision

Les mesures réalisées sur RacEr compareront donc 3 implémentations : le float 32 bits du standard IEEE 754, le Posit<32,2> de la version émulée de Ceralaine et le Posit <32,2> de la version Ceralaine implémentée sur RacEr. Tout comme pour la vérification des émulées, Chapitre 4.6, nous cherchons à obtenir les valeurs données par la littérature, ou du moins celle du Chapitre 4.2.

6.4.1.a Impact sur les nombres très petits

Lors de nos premières mesures, les résultats Posit obtenus grâce au calcul fait par RacEr avaient une erreur de 1×10^3 , soit 3 chiffres significatifs de moins par rapport à la version émulée. Cette précision faible tant bien même que Posit est censé être dans sa « golden zone » est inquiétante et annonce des résultats pas à la hauteur de la technologie. Cependant, comme on peut le voir sur la figure 30, la précision de RacEr est meilleure pour les valeurs très proche de 0, $[1 \times 10^{-5}; 1 \times 10^{-4}]$.

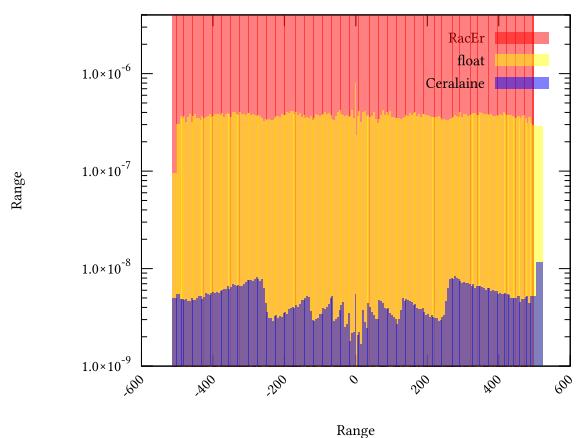


Fig. 32. – Précision de RacEr entre $[-500 : 500]$

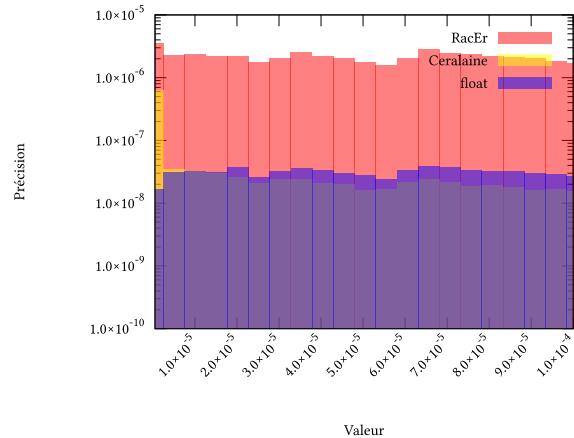


Fig. 33. – Précision de RacEr entre $[1 \times 10^{-5}; 1 \times 10^{-4}]$

6.4.2 Impact sur les nombres au centre

Nous avons continué à faire des mesures en élargissant les intervalles des nombres à calculer. Le résultat obtenu est très surprenant. Comme on peut le remarquer, la précision de Posit est nettement moins bonne que la littérature et que le standard pour les nombres centrés sur 1 mais en s'éloignant et en regardant dans l'intervalle $[-1 \times 10^6; 1 \times 10^6]$ on s'aperçoit qu'à partir d'environ $\pm 1 \times 10^5$ la précision de RacEr s'améliore jusqu'à quasiment concurrencer l'émulation de Ceralaine. Cela lui permet également de montrer un gain sur le standard IEEE 754.

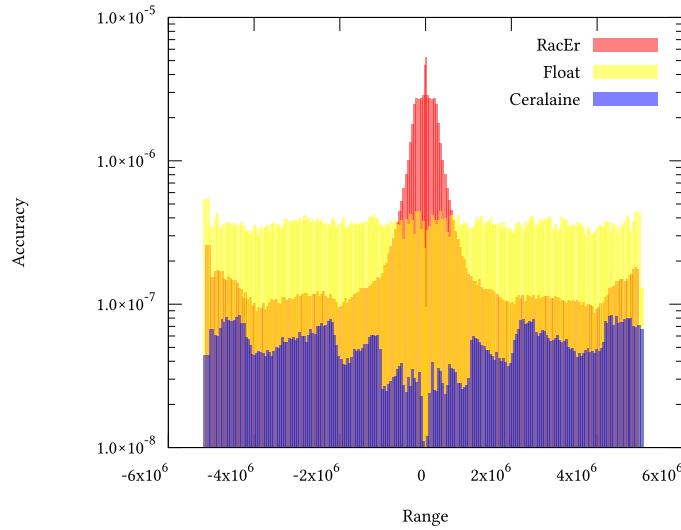


Fig. 34. – Précision de RacEr entre $[-5 \times 10^6; 5 \times 10^6]$

6.4.3 Impact sur les très grands nombres

La version de Posit sur RacEr montre des comportements surprenants en maintenant une précision supérieure à celle de Ceralaine et des valeurs du standard IEEE même en dehors de la zone où Posit est censé être meilleur. Selon la littérature, Posit est conçu pour offrir une meilleure précision autour de valeurs proches de 1, et ses avantages devraient décroître au-delà de cette zone.

Cependant, dans les résultats obtenus avec RacEr, il semble que cette implémentation de Posit parvienne à maintenir une précision élevée même lorsque les valeurs sont hors de la « golden zone ». Cela contraste avec Ceralaine qui montrent une dégradation plus rapide de la précision autour de $\pm 4 \times 10^9$.

De plus, il est important de noter que sur le graphe fourni, aucune valeur dans la plage de $[-1 \times 10^5, 1 \times 10^5]$ n'est représentée. Cela donne l'impression que la précision de RacEr est meilleure autour de 1, alors que ce ne serait pas le cas si l'on prenait en compte ces valeurs manquantes.

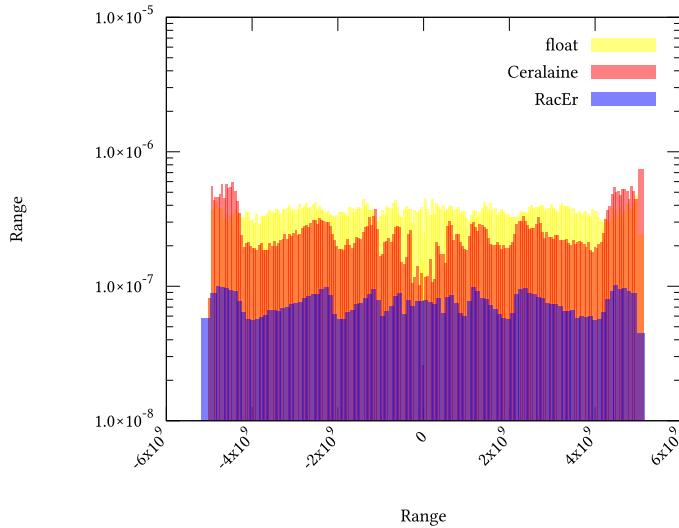


Fig. 35. – Précision de RacEr entre $[-5 \times 10^9; 5 \times 10^9]$

6.4.4 Overflow, Underflow, Not a Number

La version de Posit disponible sur RacEr utilise un mécanisme d’arrondi automatique à `MaxPos` lorsque les valeurs dépassent `5.192297e+33`. Cependant, cette valeur de seuil ne correspond pas à celle attendue d’après la littérature, qui indique généralement un seuil d’overflow autour de `2e+37` pour la version de Posit considérée. Cette différence suggère une implémentation ayant des limitations particulières dans RacEr, ce qui peut affecter la précision et la robustesse des calculs avec des valeurs extrêmes. Cela souligne l’importance de bien comprendre les comportements d’overflow propres à chaque implémentation de Posit.

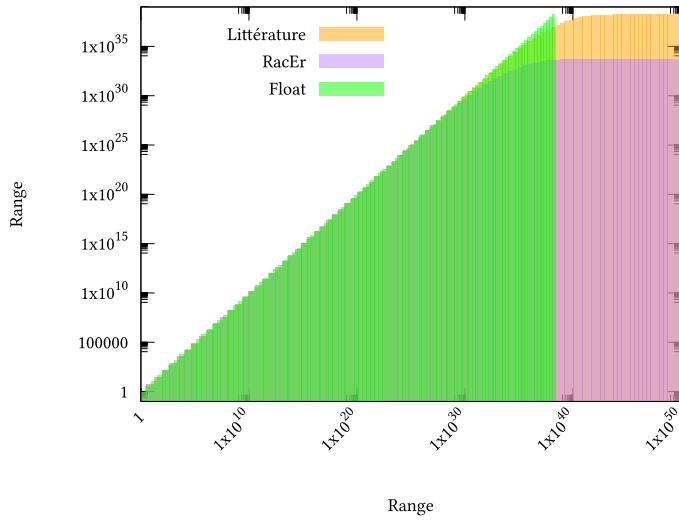


Fig. 36. –

6.4.5 Index des matrices

Des tests sur la gestion des indices dans RacEr ont révélé une erreur dans son implémentation. Lors d’une utilisation en dimension 1D, les indices sont correctement gérés en fonction de l’axe X. Cependant, lorsqu’on passe à une utilisation en dimension 2D, avec des indices sur les axes X et Y, nous avons observé que les indices sur l’axe Y sont décalés de 1. Ce décalage rend actuellement l’utilisation d’appels matriciels en 2D, impliquant à la fois X et Y, impossible.

En raison du décalage des indices sur l'axe Y, les résultats pour les appels matriciels en 2D ne correspondent pas aux attentes, et les indices ne sont pas correctement alignés avec les valeurs attendues.

</> **Code**

```
int __attribute__ ((noinline))
kernel_float_vec_reduce (posit *src, posit *dst, int block_size_x){
    for (int iter_x = _RacEr_id; iter_x < block_size_x;
        iter_x += RacEr_tiles_X * RacEr_tiles_Y){
        dst[iter_x] += src[iter_x];
    }
    RacEr_tile_group_barrier (&r_barrier, &c_barrier);
}
```

Liste 4. – Kernel d'une réduction sur RacEr

La proposition de RacEr est particulièrement intéressante dans un monde où les GPGPUs sont grandissant pour leur efficacité sur un large domaine de calcul. Posit tente d'apporter par le biais de RacEr une réponse aux défis actuels de consommation de temps et d'énergie tout en obtenant des résultats précis. Cependant, les performances observées sont loin d'être satisfaisantes et sont même décevantes par rapport aux émulations que nous avons testées. De plus, d'autres problèmes subsistent dans l'implémentation de l'architecture RacEr, ce qui complique davantage sa mise en pratique.

Enfin, nous avons pu essayer la programmation sur FPGA. Vous trouverez en annexe au Chapitre 10.6 une explication détaillée des applications d'exemples que nous avons implémenté et des problèmes qui se sont heurtés à nous.

7 Cas d'études

Enfin, nous avons réalisé plusieurs études de cas, montrant ou non l'intérêt de Posit pour certaines applications. Nous avons choisi d'en présenter deux : une simulation moléculaire et un calcul d'algèbre linéaire portant sur une inversion de matrice. Ces cas ont été réalisés grâce aux émulations de Posit et démontrent un réel intérêt pour cette représentation, notamment en raison du gain significatif en précision qu'elle permet. Rappelons qu'il n'existe encore aucune implémentation matérielle sur ASIC, et que cela permettrait également de valider les performances de Posit du point de vue du temps d'exécution.

7.1 Simulation moléculaire

Pour ce premier cas d'étude, nous avons utilisé un code de simulation microscopique basé sur le potentiel de Lennard-Jones pour étudier un fluide composé de particules homogènes, c'est-à-dire que toutes les particules sont identiques. Nous avons réparti aléatoirement 1000 particules dans un espace pseudo-infini en simulant des cubes adjacents à notre cube d'étude, afin de modéliser les interactions aux limites du cube.

Pour simuler l'évolution temporelle, nous avons utilisé l'algorithme de Verlet pour les vitesses, en l'initialisant avec des valeurs également tirées aléatoirement. Lors de cette simulation, nous nous sommes concentrés sur deux paramètres principaux :

- L'énergie globale du système, calculée à partir de l'énergie cinétique et de l'énergie potentielle, en utilisant un rayon de coupure.
- La température du système, ainsi que la vérification que la somme des forces du système est égale à zéro.

Pour cette simulation, l'ordre de grandeur des variables est compris entre 0 et $1e^4$.

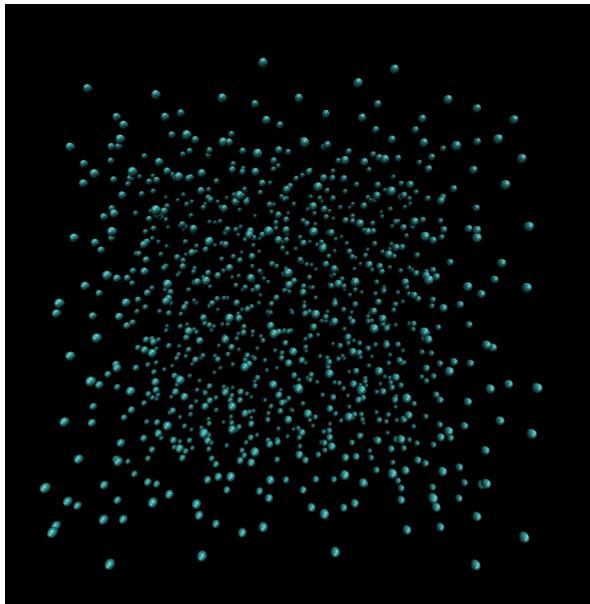


Fig. 37. – Début de la simulation

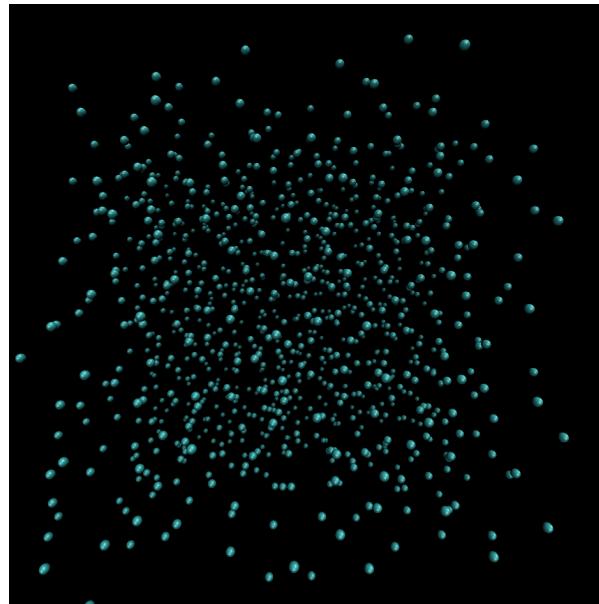


Fig. 38. – Fin de la simulation

</> Résultats

```
Energy verlet: 797.305560 = -1046.684419 + 1843.989979 Temperature: 309.185229
Sum of verlet forces: 1.932676e-12
```

Liste 5. – Posit 64 bits

</> Résultats

```
Energy verlet: 758.659959 = -1085.330020 + 1843.989979 Temperature: 309.185229
Sum of verlet forces: -1.134026e-11
```

Liste 6. – Float 64 bits IEEE

Deux changements importants sont à noter. Le premier concerne l'énergie totale du système. Après 1000 itérations, cette énergie varie d'environ 40, ce qui représente environ 7% de l'énergie totale.

Le second changement concerne la somme des forces. Nous nous attendons à ce que cette somme soit égale à zéro. Toute déviation de cette valeur indique une erreur due aux calculs. Nous observons que l'utilisation de Posit permet d'obtenir un chiffre significatif supplémentaire pour le calcul de cette somme.

7.2 Algèbre linéaire

En algèbre linéaire, une propriété intéressante des matrices est que l'inverse de l'inverse d'une matrice A , noté $(A^{-1})^{-1}$, est toujours la matrice A elle-même. Cela signifie que, après avoir inversé une matrice, puis inversé le résultat, nous retrouvons la matrice d'origine.

Cette propriété est essentielle pour évaluer la précision numérique des calculs. Toute déviation par rapport à la matrice originale après une double inversion indique une erreur dans le processus de calcul. Ainsi, la vérification par l'inversion inverse constitue un excellent test de la robustesse des algorithmes et de leur capacité à maintenir la précision des données.

</> Résultats

```
A = [ 8.401877 3.943829
      7.830992 7.984400 ]
B = [ 8.401877 3.943829
      7.830992 7.984400 ]
res = 0.000000, rms = 0.000000
```

Liste 7. – Float 64 bits IEEE

</> Résultats

```
A = [ 8.401877 3.943829
      7.830992 7.984400 ]
B = [ 8.401987 3.943966
      7.831672 7.984796 ]
res = -0.000466, rms = 0.000540
```

Liste 8. – Float 32 bits IEEE

</> Résultats

```
A = [ 8.401877 3.943829
      7.830992 7.984400 ]
B = [ 8.401852 3.943811
      7.830964 7.984383 ]
res = 0.000015, rms = 0.000025
```

Liste 9. – Posit 32 bits ES 2

</> Résultats

```
A = [ 8.401877 3.943829
      7.830992 7.984400 ]
B = [ 8.401908 3.943855
      7.830888 7.984372 ]
res = 0.000005, rms = 0.000028
```

Liste 10. – Posit 32 bits ES 1

Comme illustré, l'utilisation de Posit permet de réduire l'erreur lors de la double inversion de matrice. Considérons $B = A + A_\delta$, où A_δ représente une petite perturbation. Nous avons évalué deux types d'erreurs : l'erreur relative et l'erreur quadratique moyenne. Les résultats montrent que Posit offre une robustesse supérieure pour chacune de ces mesures, assurant ainsi une meilleure précision dans le calcul de l'inversion inverse. On remarque également la possibilité d'utiliser la représentation de Posit<32,2> à la place de celle 64 bits du standard IEEE 754. Cela engendrerait une perte de précision légère pour un gain en temps et espace de mémoire d'un facteur 2.

8 Conclusion

À travers notre étude, nous avons exploré les domaines où Posit est supposé surpasser le standard IEEE 754, notamment grâce à son adaptabilité de précision, sa représentation dynamique, sa précision accrue autour de 1, sa zone de couverture étendue, et sa gestion plus efficace des valeurs hors définition, notamment en éliminant les overflows, underflows et les NaN. Nos tests et émulations ont permis de vérifier certaines de ces caractéristiques.

Les résultats montrent que Posit offre un avantage significatif dans des plages de valeurs spécifiques, notamment entre -1×10^6 et 1×10^6 , avec un gain de précision d'un facteur 10 autour de 1. Cependant, en dehors de cette zone, la perte de précision devient notable, et peut même s'aggraver, annulant les bénéfices de Posit dans ces cas-là. Il est crucial de noter que cette perte est comparable à l'amélioration observée dans la plage centrale, ce qui limite l'intérêt de Posit à des cas d'utilisation spécifiques.

Nous avons également constaté que les performances et les précisions des différentes implémentations de Posit varient. L'implémentation FastSigmoid est la seule à offrir une zone de couverture conforme aux attentes théoriques, alors que d'autres, comme SoftPosit et Ceralaine, montrent des comportements identiques mais avec des limitations notables. Par ailleurs, la stratégie d'arrondi silencieux de Posit, bien qu'elle réduise les occurrences de NaN, peut masquer des erreurs de calcul, compromettant la fiabilité dans certaines situations.

Notre analyse montre que, même si Posit présente des avantages théoriques significatifs, ceux-ci ne se traduisent pas toujours par des bénéfices pratiques universels. D'autres formats, comme TensorFloat-32, BFloat16, FP8, et INT8, conçus pour des applications spécifiques, offrent des avantages équivalents ou supérieurs selon le contexte. En particulier, TensorFloat-32 s'avère souvent plus pertinent pour les valeurs élevées, tandis que INT8 est plus adaptés aux valeurs proches de 1.

L'implémentation matérielle de Posit via RacEr offre une perspective prometteuse pour répondre aux défis actuels de consommation d'énergie et de temps de calcul. Cependant, nos expérimentations ont montré que les performances de RacEr sont inférieures à celles des émulations logicielles testées. De plus, des problèmes d'implémentation compliquent encore son intégration dans des workflows existants.

En conclusion, bien que Posit et son implémentation via RacEr apportent des solutions innovantes, leur adoption à grande échelle dépend de la résolution de plusieurs limitations techniques et d'une évaluation approfondie des contextes d'application où ils pourraient être les plus avantageux. Dans le paysage actuel, des alternatives comme TensorFloat-32 et BFloat16 ont déjà prouvé leur efficacité dans des scénarios spécifiques, et le choix entre ces différentes approches devra être guidé par les besoins précis de chaque application.

Il semble donc peu probable que Posit puisse devenir un nouveau standard, remplaçant l'IEEE 754. En effet, le standard IEEE 754 offre une bonne précision sur l'ensemble de ses valeurs, sans zones spécifiques de meilleure ou de moindre représentation, et sa couverture est étendue, surpassant celles des alternatives proposées. Cette uniformité et cette robustesse font de l'IEEE 754 la meilleure option pour un standard, capable de répondre à une grande diversité de besoins sans les inconvénients des représentations plus spécialisées comme Posit.

Cependant, dans le domaine de l'intelligence artificielle, des essais très encouragants montrent un gain d'espace mémoire allant jusqu'à un facteur 4 pour Posit. C'est pourquoi, Posit pourrait être une bonne piste dans ce domaine pour s'abstraire d'une représentation en INT 8.

9. Bibliographie

- [1] J. L. Gustafson et I. T. Yonemoto, « Beating floating point at its own game: Posit arithmetic », *Supercomputing frontiers and innovations*, vol. 4, n° 2, p. 71-86, 2017.
- [2] J. Gustafson, « Posit arithmetic », *Mathematica Notebook describing the posit number system*, 2017.
- [3] F. De Dinechin, L. Forget, J.-M. Muller, et Y. Uguen, « Posits: the good, the bad and the ugly », in *Proceedings of the Conference for Next Generation Arithmetic 2019*, 2019, p. 1-10.
- [4] I. Yonemoto, « Implémentation fastsigmoids ».
- [5] tancheng, « Implémentation Tensorfloat 32 ».
- [6] G. E. C. Raposo, « Deep Learning with approximate computing: an energy efficient approach », 2021.
- [7] umangyadav, « Implémentation FP8 ».
- [8] P. Lindstrom, S. Lloyd, et J. Hittinger, « Universal coding of the reals: Alternatives to IEEE floating point », in *Proceedings of the Conference for Next Generation Arithmetic*, 2018, p. 1-14.
- [9] R. Murillo, A. A. Del Barrio, et G. Botella, « Customized posit adders and multipliers using the FloPoCo core generator », in *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2020, p. 1-5. doi: 10.1109/ISCAS45731.2020.9180771.
- [10] G. Raposo, P. Tomás, et N. Roma, « Positnn: Training deep neural networks with mixed low-precision posit », in *ICASSP 2021-2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2021, p. 7908-7912.
- [11] M. Cococcioni, F. Rossi, E. Ruffaldi, et S. Saponara, « Vectorizing posit operations on RISC-V for faster deep neural networks: experiments and comparison with ARM SVE », *Neural computing & applications*, vol. 33, n° 16, p. 10575-10585, 2021, doi: 10.1007/s00521-021-05814-0.
- [12] F. de Dinechin, A. V. Ershov, et N. Cast, « Towards the post-ultimate libm », in *17th IEEE Symposium on Computer Arithmetic (ARITH'05)*, 2005, p. 288-295.
- [13] R. Chaurasiya *et al.*, « Parameterized posit arithmetic hardware generator », in *2018 IEEE 36th International Conference on Computer Design (ICCD)*, 2018, p. 334-341.
- [14] M. K. Jaiswal et H. K.-H. So, « Architecture generator for type-3 unum posit adder/subtractor », in *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2018, p. 1-5.
- [15] L. Crespo, P. Tomás, N. Roma, et N. Neves, « Unified posit/IEEE-754 vector MAC unit for transprecision computing », *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 69, n° 5, p. 2478-2482, 2022.
- [16] C. W. Clenshaw et F. W. Olver, « Beyond floating point », *Journal of the ACM (JACM)*, vol. 31, n° 2, p. 319-328, 1984.
- [17] N. Neves, P. Tomás, et N. Roma, « Dynamic fused multiply-accumulate posit unit with variable exponent size for low-precision DSP applications », in *2020 IEEE Workshop on Signal Processing Systems (SiPS)*, 2020, p. 1-6.
- [18] Y. Uguen, L. Forget, et F. de Dinechin, « Evaluating the hardware cost of the posit number system », in *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, 2019, p. 106-113.

- [19] L. SRAVANI et M. M. ADISESHAIAH, « DESIGN OF A POWER EFFICIENT 32-BIT POSIT MULTIPLIER ».
- [20] A. A. Jonnalagadda, U. A. Kumar, R. Thotli, S. Sardesai, S. Veeramachaneni, et S. E. Ahmed, « ADEPNET: A Dynamic-Precision Efficient Posit Multiplier for Neural Networks », *IEEE Access*, vol. 12, p. 31036-31046, 2024.
- [21] M. K. Jaiswal et H. K.-H. So, « PACoGen: A hardware posit arithmetic core generator », *IEEE access*, vol. 7, p. 74586-74601, 2019.
- [22] Q. Li, C. Fang, et Z. Wang, « PDPU: An Open-Source Posit Dot-Product Unit for Deep Learning Applications », in *2023 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2023, p. 1-5.
- [23] S. Mach, F. Schuiki, F. Zaruba, et L. Benini, « FPnew: An open-source multiformat floating-point unit architecture for energy-proportional transprecision computing », *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 29, n° 4, p. 774-787, 2020.
- [24] N. Nakasato, Y. Murakami, F. Kono, et M. Nakata, « Evaluation of POSIT Arithmetic with Accelerators », in *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*, 2024, p. 62-72.
- [25] M. Cococcioni, F. Rossi, E. Ruffaldi, S. Saponara, et B. D. de Dinechin, « Novel arithmetics in deep neural networks signal processing for autonomous driving: Challenges and opportunities », *IEEE Signal Processing Magazine*, vol. 38, n° 1, p. 97-110, 2020.
- [26] A. Y. Romanov *et al.*, « Analysis of posit and bfloat arithmetic of real numbers for machine learning », *IEEE Access*, vol. 9, p. 82318-82324, 2021.
- [27] R. Murillo, A. A. Del Barrio, G. Botella, M. S. Kim, H. Kim, et N. Bagherzadeh, « PLAM: A posit logarithm-approximate multiplier », *IEEE Transactions on Emerging Topics in Computing*, vol. 10, n° 4, p. 2079-2085, 2021.
- [28] S. H. F. Langrouri, T. Pandit, et D. Kudithipudi, « Deep learning inference on embedded devices: Fixed-point vs posit », in *2018 1st Workshop on Energy Efficient Machine Learning and Cognitive Computing for Embedded Applications (EMC2)*, 2018, p. 19-23.
- [29] F. Rossi, F. Urbani, M. Cococcioni, E. Ruffaldi, et S. Saponara, « FPPU: Design and Implementation of a Pipelined Full Posit Processing Unit », *arXiv preprint arXiv:2308.03425*, 2023.
- [30] « RacEr: Posit GPGPU ». [En ligne]. Disponible sur: <https://vivid-sparks.com/>
- [31] A*Star, « Implémentation softposit ».

10 Annexe

10.1 Glossaire

P<8,1> : Posit 8 bits avec 1 bit d'exposant

FP8 : Floating Point 8 bits

FP(8,3) : Floating Point 8 bits avec 3 bits d'exposants

FP8 E5M2 : Equivalent à FP(8,5), signifiant Floating Point 8 bits avec 5 bits d'exposants et 2 pour la mantisse.

GPGPU : General-Purpose Graphic Processing Units

FPGA : Field Programmable Gate Arrays

IP : Intellectual Property

IA : Intelligence artificielle

API : Application Programming Interface (interface de programmation en français)

10.2 Mesures des émulations

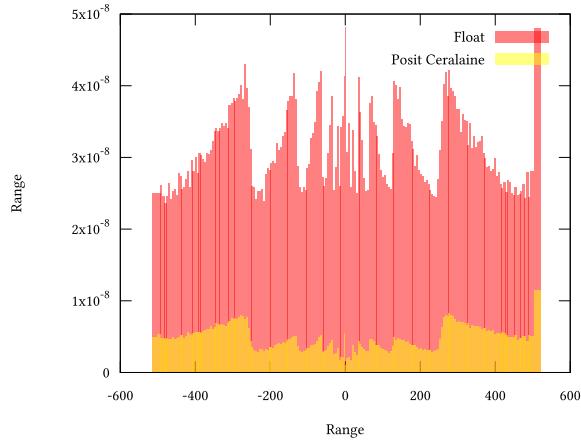


Fig. 39. – Emulation de Ceralaine autour de 0

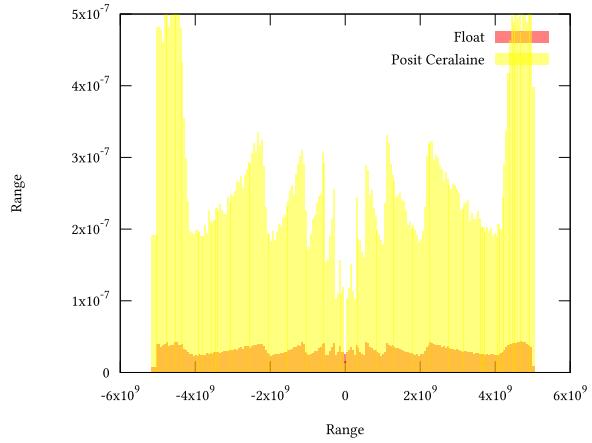


Fig. 40. – Emulation Ceralaine sur -5×10^9 à 5×10^9

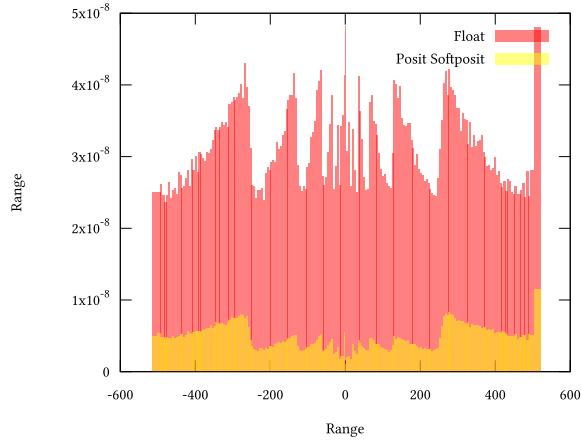


Fig. 41. – Emulation de softposit autour de 0

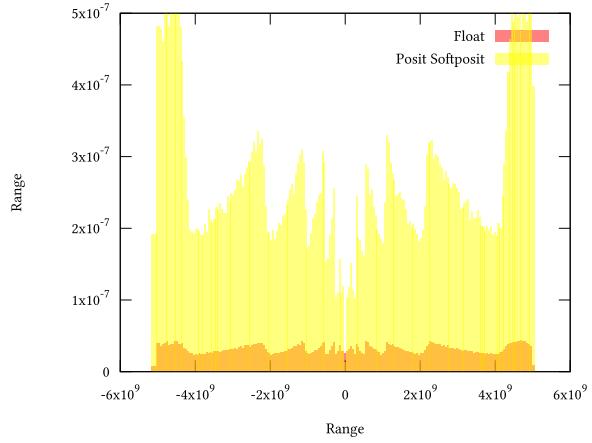


Fig. 42. – Emulation softposit sur -5×10^9 à 5×10^9

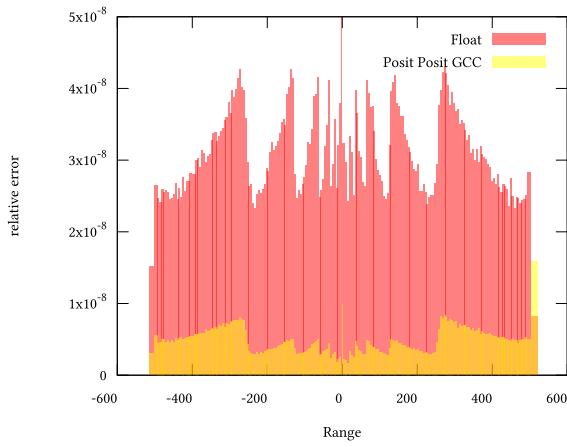


Fig. 43. – Emulation de Posit-GCC autour de 0

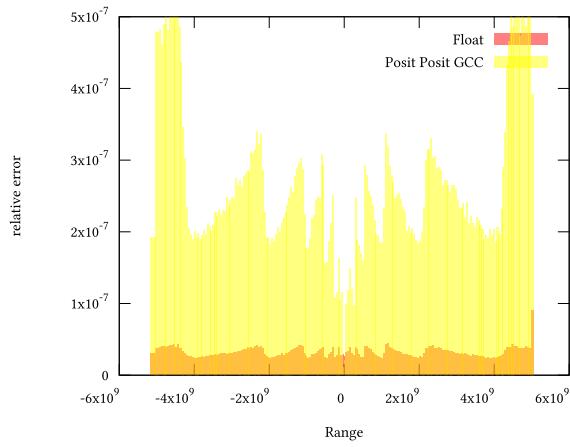


Fig. 44. – Emulation Posit_GCC sur -5×10^9 à 5×10^9

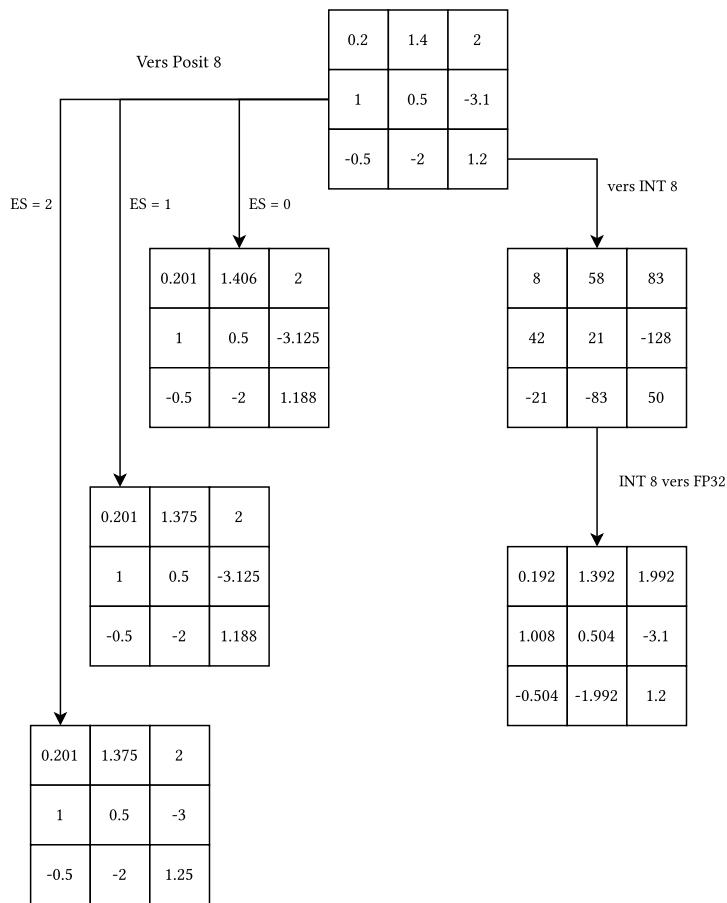


Fig. 45. – Exemple de conversion de matrice FP32 vers Posit 8 et INT 8

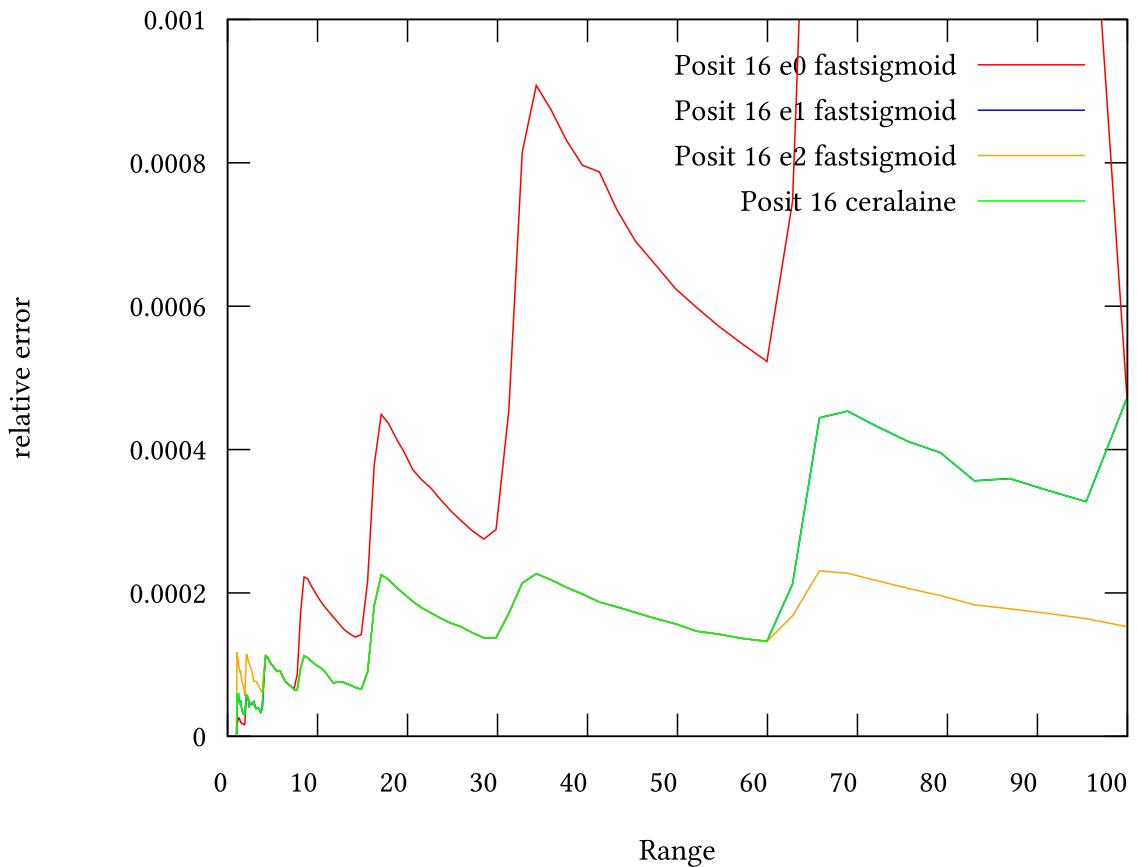


Fig. 46. – Comparaison de Fastsigmoid et Ceralaine pour Posit 16

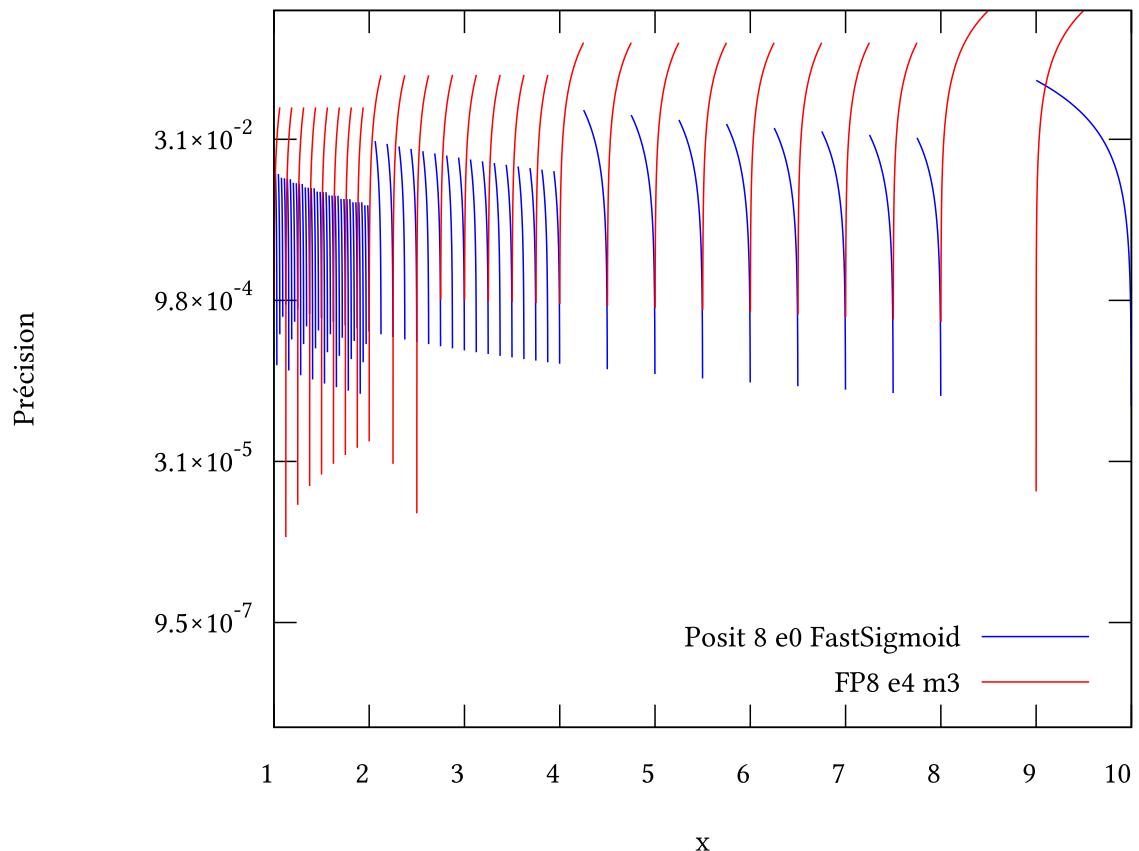


Fig. 47. – Comparaison FP8 e4m3 et Posit <8,0> entre 1 et 10

10.3 Représentation architecture RacEr

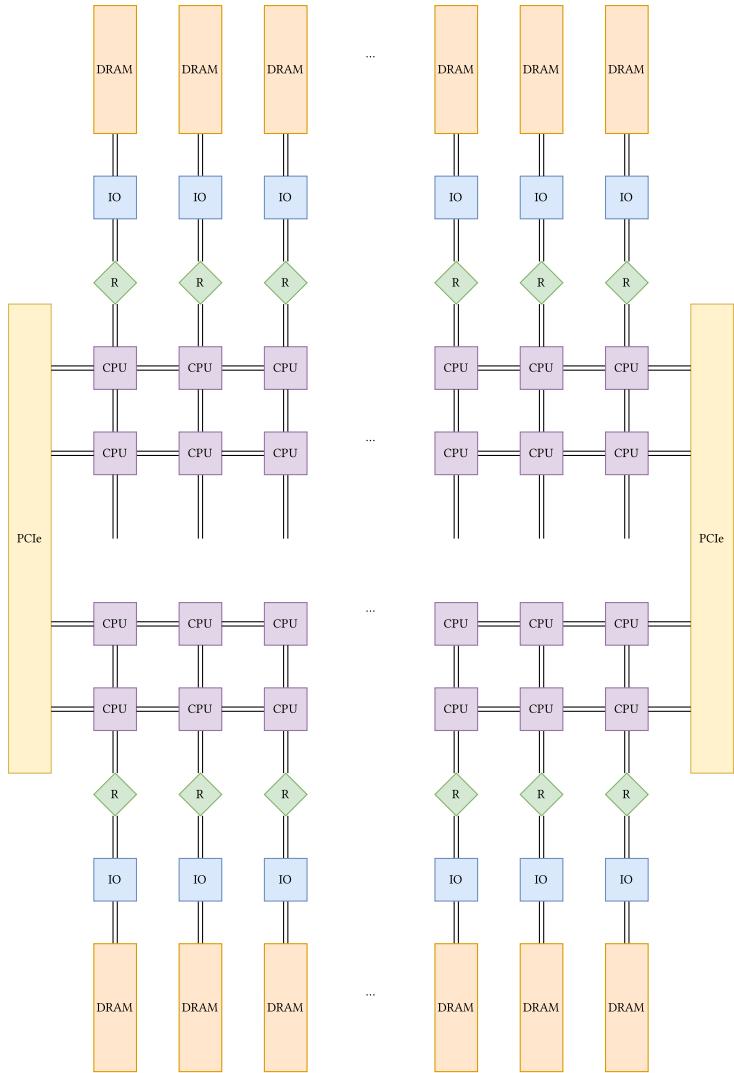


Fig. 48. – Schéma de l'architecture RacEr, 1 CPU correspond à 1 tuile de 4 threads

10.4 Portage d'une application

10.4.1 Initialisation du device

Pour cette étape supplémentaire requise par RacEr, nous pouvons voir, sur l'exemple : Liste 11, toutes les variables et fonctions nécessaires. Ces variables sont utilisées pour stocker le chemin binaire du noyau, le nom du test et l'allocateur. Pour les fonctions, l'une est utilisée pour attacher le périphérique en utilisant l'identifiant du périphérique et l'autre est utilisée pour lier le noyau sur le périphérique. Cette étape requise pour RacEr n'existe pas pour CUDA.

```

// RacEr
#define ALLOC_NAME "default_allocator"
char *bin_path, *test_name;
struct arguments_path args = { NULL, NULL };

argp_parse (&argp_path, argc, argv, 0, 0, &args);
bin_path = args.path;
test_name = args.name;

RacEr_mc_device_t device;
RacEr_mc_device_init (&device, test_name, 0);
RacEr_mc_device_program_init (&device, bin_path, ALLOC_NAME, 0);

```

Liste 11. – Device initialization on RacEr

10.4.2 Différences principales :

- La plupart des fonctions sont similaires, sauf qu'il faut remplacer le mot-clé « cuda » par « RacEr_mc_device » et préciser le périphérique comme premier argument.

```

// RacEr
RacEr_mc_device_memcpy (&device, dst, src, size, HB_MC_MEMCPY_KIND);

// CUDA
RacEr_mc_device_memcpy (dst, src, size, cudaMemcpyKind);

```

Liste 12. – API in main file

La façon d'exécuter un noyau diffère pour les deux api. RacEr utilise comme CUDA, une grille et une description de bloc mais dans l'appel de fonction, RacEr utilise 2 fonctions. La première est utilisée pour mettre en file d'attente le noyau sur le périphérique. Elle utilise une table avec les paramètres requis du noyau. La seconde fonction est juste un appel pour exécuter le noyau.

```

// RacEr
RacEr_mc_dimension_t tg_dim = { .x = 2, .y = 2 };
RacEr_mc_dimension_t grid_dim = { .x = 1, .y = 1 };
int cuda_argv[3] = { d_A, d_B, n };
RacEr_mc_kernel_enqueue (&device, grid_dim, tg_dim, "vector_add", 3, cuda_argv);
RacEr_mc_device_tile_groups_execute (&device);

// CUDA
int block_size = 256;
int numBlocks = (n + block_size - 1) / block_size;
vector_add<<<numBlocks, block_size>>> (d_A, d_B, n);

```

Liste 13. – Run a kernel

Pour gérer les erreurs, RacEr utilise des codes de retour. Tous les codes d'erreurs sont référencés dans le *user guide*

```

// RacEr
rc = RacEr_mc_device_tile_groups_execute (&device);
if (rc != HB_MC_SUCCESS)
{
    return rc;
}

```

Liste 14. – Error management

10.4.3 Différences au niveau des coeurs de calcul :

Sur RacEr, il n'y a pas de float IEEE-754. A la place, nous utilisons Posit. Pour utiliser Posit, il suffit de remplacer « float » par « posit »

```

// RacEr
int __attribute__ ((noinline)) vector_add (posit *A, posit *B, int n);

// CUDA
__global__ void vector_add(float *A, float *B, int n);

```

Liste 15. – How to use Posit instead of IEEE-754 float

Pour définir les ID de chaque thread, la méthode est assez similaire à l'exception du nom de chaque partie (grille, bloc)

```

// RacEr
int start_x = block_size_x * (_RacEr_tile_group_id_y * _RacEr_grid_dim_x +
_RacEr_tile_group_id_x);

// CUDA
int id = threadIdx.y * (gridDim.x * blockDim.x) + threadIdx.x;

```

Liste 16. – Get thread ID

Sur RacEr, le noyau est écrit comme sur un CPU. Cela signifie que vous devez spécifier la plage de couverture de la boucle for et simplement incrémenter par « RacEr_tiles_X x RacEr_tiles_Y » pour diviser le travail avec tous les threads.

```

// CPU
for (int iter_x = 0; iter_x < n; iter_x)
{
    A[iter_x] = A[iter_x] + B[iter_x];
}

// RacEr
for (int iter_x = _RacEr_id; iter_x < block_size_x;
    iter_x += RacEr_tiles_X * RacEr_tiles_Y)
{
    A[iter_x] = A[iter_x] + B[iter_x];
}

// CUDA
if (id < n)
    A[id] = A[id] + B[id];

```

Liste 17. – Kernel loop

- Définition des barrières

```
INIT_TILE_GROUP_BARRIER (r_barrier, c_barrier, 0, RacEr_tiles_X - 1, 0, RacEr_tiles_Y - 1);
```

Liste 18. – Barrier

10.5 Manuel RacEr

10.5.1 Fonctions fichiers kernel Racer

RacEr Kernel Function	Definition
RacEr_set_tile_x_y()	Sets RacEr_X and RacEr_Y to the X and Y coordinate of the tile.
RacEr_x_y_to_id(int x, int y)	Calculates tile's flat id using its (x,y) coordinates.
RacEr_id_to_x(int x)	Calculates tile's x coordinate using its flat id.
RacEr_id_to_y(int y)	Calculates tile's y coordinate using its flat id.

<code>RacEr_remote_ptr (int x, int y, void *addr)</code>	Forms a remote address by taking in x and y coordinates of remote tile and the address to local variable. Used in RacEr_remote_store and RacEr_remote_load.
<code>RacEr_remote_store (int x, int y, void *addr, int val)</code>	Stores val to the local address addr in the memory space of the tile at (x,y)
<code>RacEr_remote_store_uint8 (int x, int y, void *addr, unsigned char val)</code>	Stores the 1-byte val to the local address addr in the memory space of the tile at (x,y)
<code>RacEr_remote_store_uint16 (int x, int y, void *addr, unsigned short val)</code>	Stores the 2-byte val to the local address addr in the memory space of the tile at (x,y)
<code>RacEr_remote_load (int x, int y, void *addr)</code>	Loads from the local address addr in the memory space of the tile at (x,y)
<code>RacEr_dram_ptr (void *addr)</code>	Forms a pointer to an element on the DRAM attached to the bottom of tile's column using the local address.
<code>RacEr_dram_load (void *addr, val)</code>	Loads from DRAM into val by using RacEr_dram_ptr.
<code>RacEr_dram_store (void *addr, val)</code>	Stores val into dram by using RacEr_dram_ptr.
<code>RacEr_tilegroup_ptr (void *addr, int index)</code>	Takes in the local address of tilegroup-shared memory, and the array index. Calculates the coordinates of the tile holding that index, and returns a RacEr_remote_ptr to that element.
<code>RacEr_tilegroup_load (void *addr, int index, val)</code>	Loads from tilegroup-shared memory into val by taking in its local address and index, using RacEr_tilegroup_ptr.
<code>RacEr_tilegroup_store (void *addr, int index)</code>	Stores local val to tilegroup-shared memory by taking in its local address and index, using RacEr_tilegroup_ptr.
<code>RacEr_remote_control_store (int x, int y, void *addr, val)</code>	Remote stores the value into the instruction memory of tile (x,y) using local address.
<code>RacEr_remote_freeze (int x, int y)</code>	Starts the execution of tile (x,y) using RacEr_remote_control_store.
<code>RacEr_remote_unfreeze (int x, int y)</code>	Stops the execution of tile (x,y) using RacEr_remote_control_store.
<code>INIT_TILE_GROUP_BARRIER (ROW_BARRIER_NAME, COL_BARRIER_NAME, int x_cord_start, int x_cord_end, int y_cord_start, int y_cord_end)</code>	Initializes parameters for a barrier instruction for all tiles within tilegroup using the start and end coordinates of the tilegroup.
<code>RacEr_tile_group_barrier (ROW_BARRIER_NAME, COL_BARRIER_NAME)</code>	Synchronizes all tiles within the tilegroup by taking in the row and column barrier names generated by <code>INIT_TILE_GROUP_BARRIER</code> .
<code>RacEr_wait_while(int cond)</code>	Wait for condition to be true
<code>poll_range(int range, unsigned char *ptr_value)</code>	Check if no 0 value in <code>ptr_value</code>
<code>RacEr_print_float(posit f)</code>	Print posit value

10.5.2 Fonctions fichiers main Racer

RacEr main function	Definition
<code>RacEr_mc_device_malloc (RacEr_mc_device_t *device, int size, void *src)</code>	allocate memory <code>size</code> on targeted <code>device</code> accessible with the <code>src</code>
<code>RacEr_mc_device_memcpy (RacEr_mc_device_t *device, void *dst, void *src, int size, hb_mc_memcpy_kind kind)</code>	Copy memory <code>size</code> from <code>src</code> to <code>dst</code> on the <code>device</code> . You must specify how : <code>HB_MC_MEMCPY_TO_DEVICE</code> or <code>HB_MC_MEMCPY_TO_HOST</code>
<code>RacEr_mc_device_memset (RacEr_mc_device_t *device, void *src, int value, int size)</code>	Set memory <code>size</code> for the <code>src</code> to <code>value</code> on the <code>device</code>
<code>RacEr_printf (format, ...)</code>	print
<code>RacEr_mc_kernel_enqueue (RacEr_mc_device_t *device, RacEr_mc_dimension_t grid_dim, RacEr_mc_dimension_t tg_dim, char *function_name, int function_argc, int *function_argv)</code>	Enqueue function with a <code>function_name</code> on <code>device</code> . Specify how the <code>grid_dim</code> and <code>tg_dim</code> . Pass the function parameter throw <code>function_argc</code> and <code>function_argv</code> in a <code>int kernel_args[] = {function_arg, ...}</code>
<code>RacEr_mc_device_tile_groups_execute (RacEr_mc_device_t *device)</code>	Execute the function enqueue on the <code>device</code>
<code>RacEr_mc_device_init (RacEr_mc_device_t *device, char *name, int device_id)</code>	initialise the <code>device</code> with the <code>name</code> on a specific <code>device_id</code>
<code>RacEr_mc_device_program_init (RacEr_mc_device_t *device, char *bin_path, char *allocator, int device_id)</code>	initialise the program on the <code>device</code> with the <code>bin_path</code> program. It requires an <code>allocator</code> and the <code>device_id</code>
<code>argp_parse (&argp_path, int argc, char *argv, 0, 0, struct arguments_path *args)</code>	Function to parse args from Command line and store it in <code>args</code> which is an <code>arguments_path</code> type
<code>RacEr_mc_device_finish (RacEr_mc_device_t *device)</code>	Terminate the simulation on the <code>device</code>
<code>RacEr_mc_device_free(RacEr_mc_device_t *device, void *src)</code>	free allocated memory on <code>device</code>

10.5.3 Types dans RacEr

RacEr Macro	Definition
<code>RacEr_mc_dimension_t tg_dim = { .x = x, .y = y }</code> <code>RacEr_mc_dimension_t grid_dim = { .x = value / block_size_x, .y = value / block_size_y }</code>	2-dimension type for <code>tg_dim</code> and <code>grid_dim</code>
<code>enum hb_mc_memcpy_kind { HB_MC_MEMCPY_TO_DEVICE, HB_MC_MEMCPY_TO_HOST }</code>	Enum to define the copy mode
<code>RacEr_mc_device_t device</code>	Device type
<code>struct arguments_path args = {name, path}</code>	Struct to store function <code>name</code> and <code>path</code>

10.5.4 Macros dans RacEr

RacEr Macro	Definition
<pre>#define ALLOC_NAME "default_allocator"</pre> <pre>#define HB_MC_SUCCESS (0) #define HB_MC_FAIL (-1) #define HB_MC_TIMEOUT (-2) #define HB_MC_UNINITIALIZED (-3) #define HB_MC_INVALID (-4) #define HB_MC_INITIALIZED_TWICE (-4) // same as invalid #define HB_MC_NOMEM (-5) #define HB_MC_NOIMPL (-6) #define HB_MC_NOTFOUND (-7) #define HB_MC_BUSY (-8) #define HB_MC_UNALIGNED (-9)</pre>	Definition of the <code>default_allocator</code> Errno macro
<pre>#define RacEr_TILE_GROUP_X_DIM RacEr_tiles_X int start_x = __RacEr_tile_group_id_x * block_size_x int end_x = start_x + block_size_x</pre>	Get a thread action field

10.6 Implémentation sur carte FPGA

Durant le stage, nous avons eu la chance de pouvoir développer des applications autour du FPGA. Nous avons utilisés les outils fournis par Xilinx qui sont Vivado, Vitis HLS et Vitis pour faciliter la conception et le développement sur FPGA. Vivado est une suite de conception complète permettant la synthèse, l'implémentation, et la vérification des circuits logiques sur FPGA. Vitis HLS (High-Level Synthesis) permet de convertir du code en C/C++ en RTL (Register Transfer Level) optimisé pour FPGA, simplifiant ainsi le développement matériel. Vitis, quant à lui, est une plateforme unifiée qui intègre les outils de développement matériel et logiciel, permettant de créer des applications complexes en tirant parti à la fois du processeur et du FPGA, et de les optimiser pour des performances maximales.

10.6.1 Kria KV260

La carte FPGA KV260, conçue par Xilinx, est une plateforme de développement idéale pour l'intelligence artificielle et le edge computing. Basée sur le SoC Zynq UltraScale+, elle combine un FPGA programmable avec des coeurs ARM, offrant puissance et flexibilité pour des applications exigeantes comme la vision par ordinateur. La KV260 permet un prototypage rapide et une accélération matérielle efficace, tout en supportant les principaux frameworks d'IA, ce qui en fait une solution robuste pour les projets modernes nécessitant des performances élevées et une faible latence.

10.6.1.a Hello World

Dans un premier temps, pour prendre en main la carte FPGA et les outils de programmation, nous avons réalisé un « Hello World ». Pour ce faire, nous avons d'abord conçu un design sur Vivado en utilisant l'IP spécifique à la carte.

i Info

Le modèle de l'IP fournit par AMD correspondant à la carte Kria KV260 possèdait quelques erreurs dans la définition de la DDR. Cette erreur empêchait tout fonctionnement de la carte lors de l'envoie du design matériel. Le correctif du design de l'IP original est disponible en annexe. Chapitre 10.7

Ensuite, à partir de l'application Vitis Classic, nous avons conçu un programme simple en utilisant le design matériel précédemment créé sur Vivado. Enfin, après avoir déployé le modèle et le programme tout en spécifiant correctement le port d'écoute, nous avons obtenu notre « Hello World », confirmant ainsi la réussite de l'exécution du programme sur le design créé.

10.6.1.b Déplacement de données

Après avoir maîtrisé les outils de programmation pour FPGA, nous avons complexifié le programme en créant notre première IP. Celle-ci nous permet de lire une donnée à un emplacement de la mémoire et de la réécrire à un autre. Grâce à cette IP, nous avons pu programmer en VHDL et nous familiariser avec les communications AXI, indispensables pour faire interagir les différentes IPs. Le code est disponible en annexe sous la référence Liste 19. Nous avons mis en place un AXI Stream Master/Slave minimal, comprenant 3 bits pour indiquer si le port est prêt, valide, et s'il a terminé, ainsi qu'un vecteur de données de 32 bits. Cette IP nous servira à interfaçer correctement notre code Posit avec le reste du design.

Il a ensuite fallu adresser correctement la mémoire pour savoir où lire et écrire les données. Une fois ces questions de mémoire résolues, nous sommes passés à Vitis. Le code écrit nécessitait de connaître les adresses de lecture et d'écriture, ainsi que les décalages. Nous avons ensuite compilé et déployé le programme sur le FPGA, comme précédemment.

Malheureusement, nous n'avons pas réussi à établir une communication correcte sur les ports adressés. Lors de l'exécution de notre code sous Vitis, l'envoi des données n'a pas abouti, celles-ci étant restées bloquées en attente d'une confirmation positive de la réception.

10.6.2 Posit 32

10.6.2.a Flo-Posit

Nous avons utilisé pour se faire le repertoire Github de RaulMurillo, [Flo-Posit](#). Celui ci à généré des instances de PAU(Posit Arithmetic Units) en VHDL à partir de l'outil FloPoCo. FloPoCo est un outil de génération de cœur arithmetique pour les FPGAs. Par défaut, FloPoCo ne génère que des cœur floating point avec un exposant et une mantisse, voir Fig. 2, en permettant de changer le nombre de bits alloué pour chaque partie. FloPoCo a été développé par une équipe de l'inria et est accessible ici : [FloPoCo](#)

Lors de l'utilisation du code VHDL Flo-Posit, plusieurs problèmes ont été rencontrés. Tout d'abord, la connexion des entrées et sorties du bloc Posit avec la partie PS (Processing System) a posé des difficultés, en particulier pour assurer une intégration correcte avec le reste du système. Ensuite, l'assignation des adresses de début en PL (Programmable Logic) n'a pas été triviale, nécessitant une attention particulière pour garantir la correspondance avec les adresses utilisées dans le design. L'importation du fichier VHDL fourni par RaulMurillo et la création d'un bloc design PS/PL sur la carte Kria ont également présenté des défis, notamment en termes de compatibilité et d'intégration. Pour gérer les données, la mise en place d'une FIFO (First-In-First-Out) via FIFO Generator s'est révélée complexe, car elle devait correctement ajouter les résultats à la mémoire. Enfin, le transfert des octets entre la partie PS et la partie PL a soulevé des problèmes, nécessitant une gestion précise des données pour assurer la cohérence et l'efficacité de la communication entre les deux systèmes. Ces problèmes ont entravé la mise en œuvre fluide du code VHDL Flo-Posit, nécessitant des ajustements et des validations approfondies pour les surmonter.

Comme pour la partie précédente, après avoir réussi à obtenir un bitstream avec Vivado, nous avons rencontré des difficultés avec Vitis, ce qui nous a empêchés de finaliser notre implémentation. Malgré l'obtention d'un bitstream, les problèmes survenus lors de l'intégration et de la configuration sous Vitis nous ont empêché de faire fonctionner notre architecture et ainsi d'en obtenir des résultats.

10.6.2.b Vitis HLS

Nous avons rencontré des difficultés à rendre l'IP exportée depuis Vitis HLS compatible avec Vivado. L'erreur principale indiquait l'utilisation d'une architecture cible incorrecte lors de la compilation du code écrit sous Vitis HLS. Bien que nous ayons initialement sélectionné la carte Kria KV260 comme cible, le problème persistait. Nous avons tenté plusieurs approches pour résoudre cette incompatibilité : d'abord, en remplaçant l'architecture cible par d'autres architectures similaires, puis en sélectionnant directement la puce spécifique de la carte Kria KV260. Malheureusement, aucune de ces tentatives n'a permis d'intégrer correctement notre IP dans Vivado. Cette incompatibilité met en évidence un problème dans la manière dont Vitis HLS gère certaines cibles matérielles, particulièrement lorsque l'on tente de déployer des IPs sur des configurations spécifiques comme celle de la KV260. En conséquence, nous n'avons pas pu utiliser notre IP au sein de l'environnement Vivado, ce qui a limité notre capacité à poursuivre le développement d'architecture Posit.

Cette expérience sur FPGA a été très enrichissante pour moi, car elle m'a permis d'explorer les défis liés au design matériel. Bien que mon travail ait été relativement limité, j'ai tenté de maximiser mes essais pour approfondir mes résultats. Cependant, il est frustrant de constater qu'en dehors des programmes très basiques, je n'ai pas réussi à faire fonctionner pleinement des projets plus complexes. Cela souligne les difficultés inhérentes au développement matériel.

10.7 Modification IP Zynq Ultrascale pour la carte KV260

DDR :

- enable DDR controller
- Requested device frequency : 1200 MHz
- Memory type : DDR4
- Components : Components
- Effective DRAM bus width : 64 bits
- ECC : Disable
- Speed Bin : DDR4 2400P
- Cas latency : 16
- RAS to CAS : 16
- precharge time : 16
- Cas write latency : 16
- tRC : 47.06
- tRASmin : 33.0
- tFAW : 30.0
- additive latency : 0
- DRAM IC Bus width : 16 bits
- DRAM Device Capacity : 8192 MBytes
- Bank group address count : 1
- Bank address count : 2
- Row address count : 16
- Column address count : 10
- uncheck Dual Rank

IO :

GPIO :

- GPIO0 MIO MIO 0..25
- GPIO1 MIO MIO 26..51

QSPI :

- QSPI : single

SD/eMMC :

- SD 1 MIO 39..51

SPI :

- SPI 1 MIO 6..11

UART :

- UART 1 36..37

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity axis_adder is
    Port (
        clk           : in  std_logic;
        resetn        : in  std_logic;
        -- Slave interface
        s_axis_tvalid : in  std_logic;
        s_axis_tdata  : in  std_logic_vector(31 downto 0);
        s_axis_tlast   : in  std_logic;
        s_axis_tready  : out std_logic;
        -- Master interface
        m_axis_tready : in  std_logic;
        m_axis_tvalid  : out std_logic;
        m_axis_tdata  : out std_logic_vector(31 downto 0);
        m_axis_tlast   : out std_logic
    );
end axis_adder;

architecture Behavioral of axis_adder is
    signal tvalid_int : std_logic := '0';
    signal tdata_int  : std_logic_vector(31 downto 0) := (others => '0');
    signal tlast_int  : std_logic := '0';
    signal tready_int : std_logic := '0';
begin

    process(clk, resetn)
    begin
        if resetn = '0' then
            tvalid_int <= '0';
            tdata_int <= (others => '0');
            tlast_int <= '0';
            tready_int <= '0';
        elsif rising_edge(clk) then
            if s_axis_tvalid = '1' and m_axis_tready = '1' then
                tvalid_int <= '1';
                tdata_int <= std_logic_vector(unsigned(s_axis_tdata) + 1);
                tlast_int <= s_axis_tlast;
                tready_int <= '1';
            else
                tvalid_int <= '0';
                tlast_int <= '0';
                tready_int <= '0';
            end if;
        end if;
    end process;

    s_axis_tready <= tready_int;
    m_axis_tvalid <= tvalid_int;
    m_axis_tdata <= tdata_int;
    m_axis_tlast <= tlast_int;

end Behavioral;

```

Liste 19. – Code de communication AXI en VHDL