Université P. et M. Curie

Master spécialité informatique

Calcul scientifique précis et efficace sur le processeur CELL

NGUYEN Hong Diep

Rapport de stage recherche de master 2

effectué au laboratoire LIP6

Encadrants : Stef Graillat, Jean-Luc Lamotte

Table des matières

Introduction

Le thème de mon stage est l'implémentation d'opérateurs à virgule flottante en précision étendue sur le processeur CELL. Ce processeur permet d'atteindre plus de 200 GFLOPs en simple précision mais avec uniquement un mode d'arrondi vers zéro et quelques autres restrictions par rapport à la norme IEEE 754. La simple précision ne permet pas toujours résoudre tous les problèmes numériques, il est nécessaire d'utiliser des précisions étendues sur 64, 128 ou 256 bits. L'objectif du stage consiste dans un premier temps à étudier le comportement numérique du processeur CELL. Ensuite, nous développerons une bibliothèque double simple précision et une bibliothèque quad simple précision sur le modèle de la bibliothèque proposée par Yozo Hida, Xiaoye S. Li, et David H. Bailey dans l'article "Algorithmes for quad-double precision floating point arithmetic". Le stage se terminera par une étude des performances respectives de ces deux bibliothèques.

Ce rapport est organisé de la manière suivante :

- le chapitre 1 introduit l'arithmétique flottante et les transformations exactes qui nous permettent d'obtenir le résultat exact d'une opération flottante quelconque en utilisant les précisions de la norme IEEE 754.
- le chapitre 2 présente le processeur CELL, son architecture et sa capacité de parallélisme.
- dans le chapitre 3 nous présentons la double-simple et les opérations de base sur des double-simples. Puis nous présentons le processus d'implémentation de la librairie double-simple.
- dans le chapitre 4 nous présentons la quad-simple.
- dans le chapitre 5, on présente les résultats expérimentaux sur les performances de la librairie double-simple puis on analyse les résultats obtenus.
- le dernier chapitre résume les résultats obtenus de mon stage.

Chapitre 1

L'arithmétique flottante et les EFTs (Error-Free Transformation)

1.1 L'arithmétique flottante

Les nombres à virgule flottante sont les nombres les plus souvent utilisés dans un ordinateur pour représenter des valeurs non entières.

Un nombre flottant normalisé $x \in \mathbb{F}$ est un nombre qui s'écrit sous la forme

$$x = (\pm) \underbrace{x_0.x_1 \dots x_{p-1}}_{mantisse} \times b^e, \quad 0 \leq x_i \leq b-1, \quad x_0 \neq 0.$$

Avec b la base, p la précision et e l'exposant de x. En faisant varier e, on fait "flotter" la virgule décimale. La valeur $\varepsilon = b^{1-p}$ est appelé la précision machine ou bien la borne haute de l'erreur relative.

Les nombres à virgule flottante sont des approximations de nombres réels. Soit x un nombre réel ($x \in \mathbb{R}$), l'approximation de x dans le domaine des nombres à virgule flottante, notée $fl(x)$, est déterminée de la manière suivante :

- S'il existe un nombre à virgule flottante $f \in \mathbb{F}$ tel que $x = f$ alors $fl(x) = f$
- Sinon, il existe 2 nombres consécutifs à virgule flottante $x^-, x^+ \in \mathcal{F}$ tels que : $x^- < x < x^+$. Alors

  $$fl(x) \in \{x^-, x^+\}.$$

  La valeur de $fl(x)$ va être choisie à partir de ces deux valeurs d'après le mode d'arrondi sélectionné. Il y en a quatre modes d'arrondi :

  1. au plus près : la valeur la plus proche de x,
  2. vers infini positif : la valeur la plus proche de l'infini positif,
  3. vers infini négatif : la valeur la plus proche de l'infini négatif,
  4. vers zéro : la valeur la plus proche de zéro.

Les différences de représentation interne et de comportement des nombres flottants d'un ordinateur à un autre obligeaient à reprendre finement les programmes de calcul scientifique pour les porter d'une machine à une autre jusqu'à ce qu'une norme soit proposée par l'IEEE.

La norme IEEE 754 spécifie deux formats de nombres à virgule flottante (et deux formats étendus optionnels) et les opérations associées. Ces deux formats sont sur 32 bits ("simple précision") et 64 bits ("double précision"). La répartition des bits est la suivante :

Précision	Encodage	Signe	Exposant	Mantisse	Domaine de valeur
Simple	32 bits	1 bit	8 bits	23 bits	$[-3.402823e+38, 3.402823e+38]$
Double	64 bits	1 bit	11 bits	52 bits	$[-1.79769e+308, 1.79769e+308]$

TAB. 1.1 – La norme IEEE 754

Une autre représentation des flottants binaires Soit x un nombre flottant binaire de p-bit, on a :

$$x = s \times 1.m \times 2^e,$$

où s, m, e sont respectivement le signe, la mantisse $p-1$ bits et l'exposant.
x peut aussi être représenter par

$$x = s \times 1m \times 2^{e-p+1},$$

– $1m$ est un entier tel que $2^{p-1} \leq 1m < 2^p$.
– 2^{e-p+1} est nommé $ulp(x)$ (unit in the last place).
Comme la borne de l'erreur relative est $\varepsilon = 2^{1-p}$

$$\begin{aligned} ulp(x) &= 2^e\varepsilon \\ &= \frac{|x|}{1.m}\varepsilon, \end{aligned}$$

d'où

$$\frac{\varepsilon}{2}|x| < ulp(x) \leq \varepsilon|x|.$$

On présente une série de lemmes utilisées par la suite.

Lemme 1. *Soit $b = m \times ulp(b)$ un nombre flottant de t-bit, et k un entier tel que $|k| \leq |m|$, alors $k \times ulp(b)$ est représentable par un flottant de t-bit.*

Lemme 2. *Soit a et b deux flottants de t-bit tels que $1/2 \leq a/b \leq 2$, la soustraction de a par b est représentable par un nombre flottant de t-bit. Cela veut dire que $fl(a-b) = a-b$.*

Lemme 3. *Si a et b sont deux flottants de t-bit et que $|a| \leq |b|$, alors : $ulp(b) \geq ulp(a)$ et il existe une valuer non négative m tel que $ulp(b) = ulp(a) \times 2^m$.*

Lemme 4. *Soit $\circ$ une opération flottante. On a toujours*

$$|err(a \circ b)| < ulp(fl(a \circ b)) < \varepsilon|a \circ b|.$$

1.2 Les "transformations exactes" (EFT)

Soit $\circ$ une opération quelconque sur des flottants, soit $a \circ b$ le résultat de cette opération sur deux flottants a et b. Dans plusieurs cas, ce résultat n'est pas représentable par un flottant mais se trouve entre deux flottants consécutifs x^- et x^+. Une de ces deux valeurs va être choisie pour représenter le résultat selon l'arrondi sélectionné (il en existe quatre). La différence entre le résultat exact et le résultat informatique correspond aux erreurs d'arrondi. Elle sont très petites par rapport au résultat et elles sont acceptables dans beaucoup de domaines (par exemple pour des applications multimédia).

Dans le domaine de calcul scientifique, l'exactitude du résultat est très importante. La variation du résultat doit respecter une borne supérieure en valeur absolue en fonction du domaine d'application. Avec un grand nombre d'opérations, chaque opération engendre une erreur d'arrondi. Ces erreurs s'accumulent et peuvent rendre le résultat très différent du résultat exact. Plusieurs méthodes sont proposées pour minimiser ces erreurs de calcul, tel que les méthodes compensées, l'augmentation de précision, précision arbitraire, etc. Toutes ces méthodes se basent sur des EFTs (error-free transformation).

Soit a et b deux nombres flottants et $\circ$ une opération quelconque. Il a été démontré [8, p. 14] que l'erreur effectuée lors de l'opération $a \circ b$ avec le mode d'arrondi au plus près est représentable par un flottant pour $\circ \in (+, -, \times, /)$. Avec les autres modes d'arrondi, dans la plupart de cas cette erreur est représentable mais il existe quelques cas où cette erreur n'est pas représentable. Dans ces cas, pour récupérer la valeur exacte de $a \circ b$ il faut simuler le mode d'arrondi au plus près. Les EFTs permettent d'obtenir pour une opération $a \circ b$ un couple (s, e) dont la somme est exactement égale à $a \circ b$.

1.2.1 Notation

Soit :
- a et b deux nombres flottants.
- $fl(\circ)$ la représentation flottante du résultat d'une opération $\circ$ quelconque $(+, -, \times, /)$
- $err(\circ)$ l'erreur de cette opération

L'opération $a \circ b$ donne comme résultat $fl(a \circ b) + err(a \circ b)$.

Dans la suite du document, la complexité des algorithmes sera mesurée en nombres de FLOP[1].

1.2.2 La sommation

Il existe plusieurs algorithmes pour calculer l'erreur d'arrondi d'une somme de deux flottants. On présente ci-dessous des algorithmes qui sont utilisés dans plusieurs bibliothèques existantes.

Algorithmes

Algorithme de Knuth [6]

```
1  Two-Sum (a, b)
2      s = fl(a + b)
```

[1]FLOP : FLoating-point OPeration

```
3       v = fl(s - a)
4       e = fl((a - (s - v)) + (b - v))
5   return (s,e)
```

Cette méthode coûte 6 FLOPs. Si $|a| \leq |b|$ on a une autre méthode proposée par Dekker [3] qui coûte seulement 3 FLOPs appelée le Quick-Sum.

```
1   Quick-Sum (a,b)
2       s = fl(a + b)
3       e = fl(b - (s - a))
4   return (s,e)
```

On peut calculer la somme en utilisant le Quick-Sum avec un branchement

```
1   Two-Sum1 (a,b)
2       s = fl(a + b)
3       if (|a| <= |b|) then
4           e = fl(b - (s - a))
5       else
6           e = fl(a - (s - b))
7   return (s,e)
```

On propose ci-dessous un autre algorithme donnant le même résultat.

```
1   Two-Sum2 (a,b)
2       if (|a| < |b|)
3           swap(a,b)
4       s = fl(a + b)
5       e = fl(b - (s - a))
6   return (s,e)
```

Le `Two-Sum1` est le plus rapide (si on compte un branchement comme une opération normale, cette méthode coûte 4 FLOPs), et le `Two-Sum` est le plus lente avec 6 FLOPs. Cette comparaison est simple et intuitive en utilisant le nombre d'opérations. En réalité, les contraintes d'implémentation doivent être prises en compte. On verra dans la partie **Implémentation** de ce rapport que `Two-Sum2` est le plus performant.

Les algorithmes présentés ci-dessus sont utilisables uniquement avec le mode d'arrondi au plus près. Si on utilise le mode d'arrondi vers zéro, il faut les modifier. L'algorithme suivant a été proposé par Priest [8].

```
1       Two-Sum-toward-zero (a,b)
2           if (|a| < |b|)
3               swap(a,b)
4           s = fl(a + b)
5           d = fl(s - a)
6           e = fl(b - d)
7           if (e + d != b)
8               s = a, e = b
9           return (s,e)
```

Du fait de jeu d'instructions du processeur CELL nous compterons le test et le `swap` à 1 FLOP (voir la partie **Implémentation**). Donc l'algorithme `Two-Sum-toward-zero` coûte 6 FLOPs.

Pour le `Quick-Sum-toward-zero` il nous suffit d'enlever les deux lignes 2 et 3 parce qu'on a déjà l'hypothèse que $|a| \geq |b|$. Donc le `Quick-Sum-toward-zero` coûte 5 FLOPs.

L'algorithme de Priest utilise une comparaison entre la somme $e + d$ et b. Cela veut dire que cette comparaison doit attendre la fin de toutes les instructions précédentes pour se dérouler. Nous proposons ici une modification qui remplace cette comparaison par une autre comparaison en utilisant seulement la valeur de b et d.

```
1       Two–Sum–toward–zero2 (a,b)
2           if (|a| < |b|)
3               swap(a,b)
4           s = fl(a + b)
5           d = fl(s - a)
6           e = fl(b - d)
7           if (|2 * b| < |d|)
8               s = a, e = b
9           return (s,e)
```

Il n'y a pas de grande différence entre l'algorithme `Two-Sum-toward-zero` proposé par Priest et notre algorithme, sauf que l'instruction de comparaison de la ligne 7 est remplacée pour relâcher la dépendance entre les instructions des lignes 6 et 7 qui nous permet améliorer un peu la performance de la bibliotheque. Les bénifices vont être expliquées dans la partie **Implémentation**.

Ces deux derniers algorithmes seront implémentés dans notre bibliothèque. L'exactitude du `Two-Sum-toward-zero` peut être trouvée dans [8]. Nous avons fait la preuve de l'exactitude du `Two-Sum-toward-zero2` :

Démonstration

Après deux instructions des lignes 2 et 3 de l'algorithme `Two-Sum-toward-zero2` on a : $|a| \geq |b|$. On va faire la démonstration en considérant le cas $a > 0$. Le cas $a < 0$ est symétrique et la démonstration est identique.

Trois cas sont possibles : $b \geq 0$, $-a \leq b \leq -a/2$, $-a/2 < b < 0$. Nous allons maintenant traiter ces trois cas.

Cas b $\geq$ 0 Il est évident que $a+b > 0$. Avec l'arrondi vers zéro on a $err(a+b) \geq 0$. Et puis $a+b = fl(a+b)+err(a+b)$, on déduit : $b \leq a \leq fl(a+b) \leq a+b$ et $0 \leq err(a+b) \leq b$.

Posons $b = h \times ulp(b)$, avec h un entier positif.

$a > b > 0$ d'où $ulp(a) = x \times ulp(b) \rightarrow a = k \times ulp(b)$ avec k un entier positif.

D'après l'affectation dans la ligne 4 de l'algorithme `Two-Sum-toward-zero` $s = fl(a+$

$b) \geq b$ d'où $s = fl(a+b) = l \times ulp(b)$, l est un entier positif.

$$\begin{aligned} err(a+b) &= a+b-fl(a+b) \\ &= h \times ulp(b) + k \times ulp(b) - l \times ulp(b) \\ &= (h+k-l) \times ulp(b) \\ &= m \times ulp(b). \end{aligned}$$

De plus $0 \leq err(a+b) \leq b$ d'où $err(a+b)$ est représentable (cf lemme 1).
D'après l'affectation dans la ligne 5 de l'algorithme `Two-Sum-toward-zero`

$$\begin{aligned} d &= fl(s-a) \\ &= fl(a+b-err(a+b)-a) \\ &= fl(b-err(a+b)) \\ &= fl((h-m) \times ulp(b)). \end{aligned}$$

On a $0 \leq err(a+b) \leq b$ d'où $0 \leq b - err(a+b) \leq b$
$\rightarrow b - err(a+b) = (h-m) \times ulp(b)$ est représentable
$\rightarrow b = (h-m) \times ulp(b)$ est le résultat exact.
Finalement

$$\begin{aligned} e &= fl(b-v) \\ &= fl(h \times ulp(b) - (h-m) \times ulp(b)) \\ &= fl(m \times ulp(b)) \\ &= err(a+b), \end{aligned}$$

d'où e est le résultat exact.
De plus

$$\begin{aligned} |d| &= (h-m) \times ulp(b) \\ &< h \times ulp(b) \\ &< b \\ &< |2 * b|, \end{aligned}$$

d'où la comparaison de la ligne 7 de l'algorithme `Two-Sum-toward-zero2` n'est pas satisfaite.
$\Rightarrow$ Le résultat retourné est : $(s = fl(a+b), e = err(a+b))$.

Cas -a $\leq$ b $\leq$ -a/2 d'où
$1/2 \leq -b/a \leq 1$
$\rightarrow a + b = a - (-b)$ est représentable (cf lemme 2)
$\rightarrow s = fl(a+b) = a+b.$
$\rightarrow d = fl(s-a) = b$
$\rightarrow e = fl(b-d) = 0.$
$d = b$ donc comparaison de la ligne 7 de l'algorithme `Two-Sum-toward-zero2` n'est pas satisfaite. d'où le résultat retourné est : $(s = a+b, e = 0)$.

Cas -a/2 < b < 0 d'où

$a > a + b > a/2 > |b| > 0 \rightarrow err(a + b) > 0$

comme $a/2$ est un flottant représentable, on a : $fl(a + b) \geq a/2$

$\rightarrow a > s \geq a/2$.

$\rightarrow 1/2 \leq s/a < 1$.

$\rightarrow s - a$ est représentable d'où

$$\begin{aligned} d &= fl(s - a) \\ &= s - a \\ &= a + b - err(a + b) - a \\ &= b - err(a + b), \end{aligned}$$

et

$$\begin{aligned} e &= fl(b - d) \\ &= fl(b - (b - err(a + b))) \\ &= fl(err(a + b)). \end{aligned}$$

Comme $a > a + b > |b|$ on déduit : $a > s = fl(a + b) \geq |b|$

$\rightarrow a = h \times ulp(b), s = k \times ulp(b), b = -l \times ulp(b)$, h, k, l sont des entiers positifs et que $h > k > l > 0$.

$$\begin{aligned} \rightarrow err(a + b) &= a + b - fl(a + b) \\ &= h \times ulp(b) - l \times ulp(b) - k \times ulp(b) \\ &= (h - l - k) \times ulp(b) \end{aligned}$$

Comme $b < 0$ et $err(a+b) \geq 0$ la comparaison de la ligne 7 peut s'écrire de la manière suivante :

$$\begin{aligned} |2 * b| &< |d| \\ &< |b - err(a + b)| \\ \Leftrightarrow 2 * b &> b - err(a + b) \\ \Leftrightarrow |b| &< err(a + b). \end{aligned}$$

Si la comparaison de la ligne 7 est satisfaite, le résultat retourné est $(s = a, e = b)$. Comme $0 < a + b < a$ on a

$$|e| = |b| < err(a + b) < ulp(fl(a + b)) \leq ulp(a) = ulp(s).$$

Si la comparaison de la ligne 7 n'est satisfaite, cela veut dire $|b| \geq err(a + b) \geq 0$. De plus $err(a + b) = (h - l - k) \times ulp(b)$, d'après le lemme 1 $err(a + b)$ est représentable. D'où $e = fl(err(a + b)) = err(a + b)$. Donc le résultat retourné par cet algorithme est $(s = fl(a + b), e = err(a + b))$

Dans les deux cas le fait que $e + s = a + b$ et $e < ulp(s)$ est toujours valide. Donc (s, e) est la transformation exacte de la somme de a et b.

1.2.3 Le produit

Algorithmes

Le calcul par les EFTs est plus compliqué pour le produit. Pour trouver l'erreur d'arrondi d'un produit flottant, on utilise la méthode de Dekker [3] pour couper un nombre flottant de p-bit en deux flottants de $[p/2]$-bit.

```
1  split (a)
2      facteur = 2^([(p + 1 / 2)])
3      c = fl(facteur * a)
4      x = fl(c - (c - a))
5      y = fl(a - x)
6  return (x,y)
```

Le *facteur* étant connu, il peut être calculé à l'avance. Dans ce cas, cette méthode coûte 4 FLOPs. Les deux nombres flottants du résultat représentent les deux parties haute et basse du nombre original. En utilisant ce découpage, Veltkamp [3] a proposé un algorithme pour calculer l'erreur d'arrondi du produit de deux flottants en arrondi au plus près.

```
1      Two-Product (a,b)
2          p = fl(a * b)
3          (a1,a2) = split (a)
4          (b1,b2) = split (b)
5          e = fl(a2 * b2 -
6                  (((p - a1 * b1) - a2 * b1) - a1 * b2))
7          return (p,e)
```

Comme le `Two-Sum`, cette méthode permet de calculer le produit de deux nombres flottants et d'obtenir le résultat informatique et l'erreur de ce produit. Elle est très coûteuse parce qu'elle appelle deux fois la fonction `split` qui coûte 4 FLOPs. Au total la méthode `Two-Product` coûte 17 FLOPs, ce qui est beaucoup plus que la fonction `Two-Sum`. Mais si le processeur est équipé d'un FMA (Fused Multiply-Add) qui calcule le terme $a \times b + c$ à la précision infinie et puis arrondit le résultat vers la précision courante, on peut calculer l'erreur d'un produit avec une seule opération avec l'algorithme classique suivant.

```
1      Two-Product-FMA (a,b)
2          p = fl(a * b)
3          e = fma(a,b,-p)
4          return (p,e)
```

La fonction `Two-Product-FMA` nous donne exactement le même résultat que la fonction `Two-Product` mais coûte seulement 2 FLOPs en mode d'arrondi au plus près. Cet algorithme fonctionne aussi en mode d'arrondi vers zéro. Le FMA étant implémenté sur le CELL, c'est donc la fonction `Two-Product-FMA` qui sera programmée dans notre bibliothèque. Nous allons faire la démonstration de l'exactitude du `Two-Product-FMA` en arrondi vers zéro.

Démonstration

Soient a et b deux flottants de t-bit. Posons

$a = s_1 \times 1m_1 \times 2^{e1-t}$

$b = s_2 \times 1m_2 \times 2^{e2-t}$

avec $2^t \leq 1m_1, 1m_2 < 2^{t+1}$

$a \times b = (s_1 \times s_2) \times (1m_1 \times 1m_2) \times 2^{e1+e2-2t}$

Comme $2^t \leq 1m_1, 1m_2 < 2^{t+1}$ on a aussi

$2^{2t} \leq 1m_1 \times 1m_2 < 2^{2t+2}$

Donc le résultat intermédiaire du produit de a et de b est un flottant de $(2t+1)$-bit (on ne compte pas le premier bit 1). Dans le mode de calcul le plus rapide, le processeur CELL implémente seulement le mode d'arrondi vers zéro donc le résultat informatique de $a \times b$ représente exactement les $t+1$ premiers bits du résultat intermédiaire. Alors la soustraction de $fl(a \times b)$ par $a \times b$ est exactement les $(t+1)$ derniers bits du résultat intermédiaire, cela veut dire qu'il est représentable par un flottant de t-bit et que $a \times b - fl(a \times b) = err(a \times b)$. Donc la fonction `Two-Product-FMA` est utilisable aussi bien en arrondi vers zéro qu'en arrondi au plus près.

Chapitre 2

Introduction au processeur CELL

Le CELL est un processeur conçu conjointement par IBM, Sony et Toshiba, révélé en février 2005. Il équipe notamment la console de jeu vidéo Playstation 3 de Sony. Des ordinateurs à base de CELL sont produits par IBM et Mercury Computer Systems notamment.

Les processeurs traditionnels n'évoluent plus fondamentalement depuis des années. Ils se contentaient essentiellement d'exploiter les nouveaux procédés de gravure pour monter en fréquence. Cette ère est terminée depuis le 90 nm, qui ne permet plus de monter en fréquence autant que par le passé. C'est pour cette raison que l'on assiste à l'émergence des processeurs double coeur : on exploite la nouvelle finesse de gravure pour mettre plus de transistors et développer le SMP à l'intérieur même du processeur, sans grande augmentation de fréquence.

Le CELL d'IBM adopte la même idée. Il ne tente pas d'augmenter les performances des processeurs par l'augmentation de la fréquence, puisqu'on va très bientôt atteindre la limite physique de la fréquence. Mais il consiste à tirer parti de la performance de plusieurs processeurs dans le même temps.

Il y a deux particularités qui rendent le processeur CELL très performant :

1. Le Cell n'utilise pas le "Out of Order", il ne fait aucun travail de réorganisation du code, il est dit "in order". Il ne doit pas implémenter des unités pour tracer des instructions. Cela libère beaucoup de place pour ajouter de nouvelles unités d'exécution. Le travail d'optimisation est à la charge du programmeur et du compilateur.
2. Le CELL est optimisé pour le calcul distribué : plusieurs processeurs peuvent communiquer entre eux et partager leur charge de travail.

Il dispose de 234 millions de transistors gravés sur une surface de 235mm^2 en 90 nm SOI (Silicon On Insulator). La version finale tourne à 3,2 GHz en 0,9 V et est formée de huit couches de cuivre inter-connectées. Le contrôle de la température est dynamique : dix capteurs thermiques numériques et un capteur linéaire.

Un processeur CBE (Cell Broadband Engine) est composé de :

* 1 PPE (PowerPC Processing Element) : unité généraliste simplifiée, “in order”
* 8 SPEs (Synergistic Processing Element)
* Un cache de niveau 2 de 512 KB partagé
* l'EIB (Element Interconnect Bus) qui gère les communications internes entre les différents éléments
* MIC (Memory Interface Controler) : contrôleur mémoire partagé

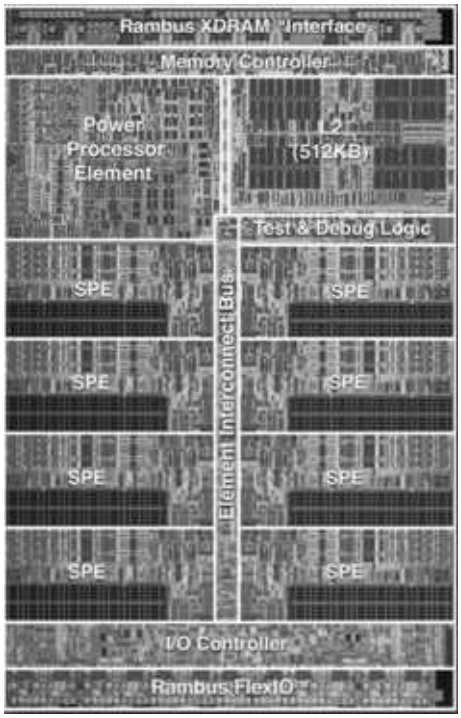

FIG. 2.1 – Un processeur CELL

* Flex I/O interface : contrôleur d'entrées-sorties (E/S)

2.1 Le PPE (PowerPC Processing Element)

Le PPE est un core basé sur l'architecture POWER et à lecture multiple de deux-voies. Le PPE contient un cache de niveau 1 d'instruction de 32 KB, un cache de niveau 1 de donnée de 32 KB et un cache de niveau 2 de 512 KB. Le coeur (PPE) utilise les jeux d'instructions POWER et AltiVec qui est entièrement pipeliné pour des calculs en double précision. Même si le PPE est très puissant et peut être utilisé comme un processeur de métier pour faire des calculs, il fonctionne plutôt comme un contrôleur pour les huit SPEs, qui gèrent presque toute la charge de calcul.

2.2 Les SPEs (Synergistic Processing Element)

Chaque SPE se compose d'un SPU "Synergistic Processing Unit", et un MFC "Memory Flow Controller" qui fait les transferts mémoire via des accès DMA. Un SPE est un processeur RISC auquel s'ajoute un processeur SIMD de 128-bit avec des instructions de calcul en simple et double précision. Chaque SPE possède une mémoire locale de 256 KB pour les instructions et les données (appelé le "local store" LS) qui est visible du PPE et peut être adressée directement par les programmes. Le LS peut être considéré comme un cache, mais il n'opère pas exactement comme un cache parce qu'il n'est pas transparent et qu'il n'y a pas de structures pour prédire quelles données sont à charger. Le programme a accès à toute la mémoire locale et est chargé de gérer toutes les communications de

données **à partir de et vers le LS**. Pour accéder à la mémoire du système le processeur SPU doit passer des adresses 64-bit au contrôleur de flux de mémoire (MFC) pour démarrer un transfert DMA dans l'espace d'adresse du système. Les SPEs contiennent un ensemble de 128 registres 128-bit. Le SPE fournit une grosse puissance de calcul sur des vecteurs. Il peut effectuer simultanément des opérations sur 16 entiers de 8-bit, 8 entiers de 16-bit, 4 entier de 32-bit ou 4 nombres flottants de simple précision. Il peut aussi effectuer une opération de mémoire dans le même temps. Il est intéressant de noter que le SPE peut effectuer en même temps des calculs sur le processeur SIMD et d'autres opérations. Le SPE contient deux pipelines d'instructions, chaque instruction est pré-assignée à un unique pipeline (tableau 2.1). Une partie du travail de lancement d'instruction est effectuée par le compilateur.

Deux instructions peuvent être lancées pour chaque cycle d'horloge à condition que [2] :

* aucune dépendance entre ces deux instructions
* les opérandes sont disponibles
* l'instruction à l'adresse paire (les 3 derniers bits d'adresse sont 000) est une instruction de pipeline 0
* l'instruction à l'adresse impaire (les 3 derniers bits d'adresse sont 100) est une instruction de pipeline 1 et
* les instructions sont dans l'ordre que pipeline 0 suivi par pipeline 1.

Instructions	Pipeline	latence
arithmétique, logique, comparaison, sélection	pair	2
opérations flottantes en simple précision	pair	6
FMA sur des entiers de 16-bits	pair	7
opérations flottantes en double précision	pair	6 + 7
décalage/rotation, estimation	impair	4
charge, stockage, canal	impair	6
débranchement	impair	1-18

TAB. 2.1 – pipelines et instructions

Le SPU du SPE est complètement pipeliné et peut effectuer une opération vectorielle à chaque cycle d'horloge. Il est basé sur un FMA simple précision, qui permet d'accomplir deux opérations flottantes dans le même temps. Avec une fréquence de 3.2 GHz, chaque SPU peut atteindre une performance théorique de $2 \times 4 \times 3.2 = 25.6$ GFLOPs de simple précision. [1]

Malheureusement, ce n'est pas la même chose avec des flottants de double précision. Les SPEs supportent l'arithmétique double précision, mais les instructions de double précision ne sont pas entièrement pipelinées. En particulier, l'opération FMA de double précision a une latence de 7 cycles. Donc la crête théorique de performance qu'un SPE peut atteindre est : $2 \times 2 \times 3.2/7 = 1.8$ GFLOPs.

[1] une addition devient $1*a+b$, une multiplication s'écrit $a*b+0$. Les divisions ne sont pas implémentées dans le matériel, elles ne sont pas performantes.

2.3 L'EIB (Element Interconnect Bus)

Toutes les composantes d'un processeur CELL telles que le PPE, les SPEs, la mémoire principale et Entrée/Sortie sont inter-connnectées par l'EIB. L'EIB est composé de quatre boucles unidirectionnelles, deux en chaque direction. Chaque participant dans l'EIB dispose d'une porte 16B d'entrée et une porte 16B de sortie. La limite pour un participant est de lire et écrire à une vitesse de 16B par cycle d'horloge EIB.

2.4 Parallélisme

Le processeur CELL a été retenu dans le cadre du projet RoadRunner du Los Alamos, qui devrait atteindre plus de 1 PFLOPs. La structure du processeur CELL nous permet d'avoir 3 niveaux de parallélisme :

1. le SPE dispose d'un processeur SIMD qui nous permet d'effectuer 4 opérations en simple précision en même temps.
2. entre le PPE et les SPEs, dont le PPE joue le rôle d'un contrôleur et les huit SPEs sont les processeurs de métier.
3. entre des processeurs CELL.

Chapitre 3

La bibliothèque double-simple

Dans ce chapitre, nous présentons la méthodologie pour augmenter la précision courante en double précision et la façon de l'implémenter dans une bibliothèque. Cette nouvelle précision, appelée le double-simple, est différente de la vraie double précision de la machine. Elle est moins bien avec moins bits de précision. L'intérêt du quasi-double est d'exploiter la puissance du SPFPU [1] au lieu d'utiliser directement le DPFPU [2]. De plus, la prochaine version du CELL supportera la double précision, mais ne supportera pas la quad précision. Dans ce cas, on pourra suivre cette méthodologie pour augmenter la précision courante en quad précision.

Comme le FPU supporte seulement l'arrondi vers zéro, toutes les opérations utilisées dans les méthodes de la bibliothèque double-simple suivent le mode d'arrondi vers zéro. Les performances de cette bibliothèque vont être testées et analysées dans le chapitre suivant.

3.1 Représentation

Dans la fonction `split`, on a décomposé un nombre flottant de p-bit en deux flottants de $[p/2]$-bit qui représentent les deux parties de $[p/2]$-bit, haute et basse, du nombre.

Inversement, on peut utiliser deux nombres flottants de p-bit pour représenter un nombre flottant de $2p$-bit. Un double-simple est une somme non-évaluée de deux nombres flottants IEEE de simple précision. Le double-simple représente la somme exacte de ces deux flottants : $a = a_0 + a_1$. Mais il existe peut-être plusieurs couples de deux flottants qui ont la même sommation, donc il y a peut-être plusieurs représentations d'un même nombre. La redondance engendre beaucoup de complexité dans l'algorithme. Pour assurer l'unicité de la représentation de double-simple on impose la normalisation des deux flottants a_1 et a_0,

$$|a_1| \leq \frac{1}{2} ulp(a_0). \tag{3.1}$$

Cette normalisation proposée par Yozo Hida [5, p. 7] ressemble au mode d'arrondi au plus près qui assure que a_0 est le nombre flottant la plus proche de $a_0 + a_1$. Nous

[1] SPFPU : Single-Precision Floating Point Unit
[2] DPFPU : Double-Precision Floating Point Unit

proposons une autre normalisation qui suit le mode d'arrondi vers zéro :

$$a_1, a_0 \text{ ont les mêmes signes et } |a_1| < ulp(a_0). \tag{3.2}$$

Cette contrainte entre les deux composantes assure que a_0 est le nombre flottant la plus proche de $a_0 + a_1$ qui se trouve entre 0 et $a_0 + a_1$.

Avec la normalisation il n'y a pas de chevauchement de bits entre les deux parties d'un double-simple.

Chaque nombre flottant membre dispose de 24 [3] bits de précision, donc le double-simple est idéalement de 48-bit précision. Donc ce n'est pas égal à un vrai double de la machine qui a 53 bits de précision. Il y a d'autres différences entre le double-simple et le vrai double de la machine à côté du nombre de bits représentatives

* Le domaine de valeur : comme le double-simple est une somme de deux flottants de simple précision son domaine de valeur dépend du domaine de valeur d'un simple :
  - La plus petite valeur absolue représentable est aussi la plus petite valeur représentable d'un flottant de simple précision, soit 2^{-127}.
  - La plus grande valeur absolue représentable est un peu plus grande que la borne haute d'un flottant de simple précision : $(1 + \varepsilon) \times Max(simple\_précision)$.
* La précision :
  - L'intervalle entre deux double-simples successifs près de zéro est plus grand que l'intervalle entre deux vrais doubles successifs près de zéro de la machine.
  - L'intervalle entre deux grands double-simples successifs est plus petit que l'intervalle entre deux grands vrais doubles successifs de la machine. En théorie, on utilise 2 flottants de simple précision pour représenter deux parties d'un double précision. Mais en fait, en utilisant une somme de 2 flottants avec seulement une restriction sur les valeurs absolues on peut représenter des nombres de plus de bits en ignorant des zéros entre les deux parties.

3.2 Opérations de base sur des double-simples

3.2.1 Notations

Dans cette section, on va utiliser des notations de la figure 3.1 pour décrire des schémas de calcul.

3.2.2 Renormalisation

Très souvent, les algorithmes numériques sur des double-simple produisent un résultat de 2 ou 3 flottants mais il reste encore des chevauchement entre ces flottants. Il faut donc transformer le résultat dans la représentation standard. Cette phase est appelée la renormalisation.

Si le résultat intermédiaire se compose de 2 flottants, l'algorithme de renormalisation est appelé `Renormalise2`. Comme on a deux types de normalisation, on a aussi deux versions correspondantes de renormalisation sur deux nombre flottants d'entrée qui sont appelées `Renormalise2-nearest` et `Renormalise2-toward-zero` respectivement.

[3] 23 bits + le bit caché.

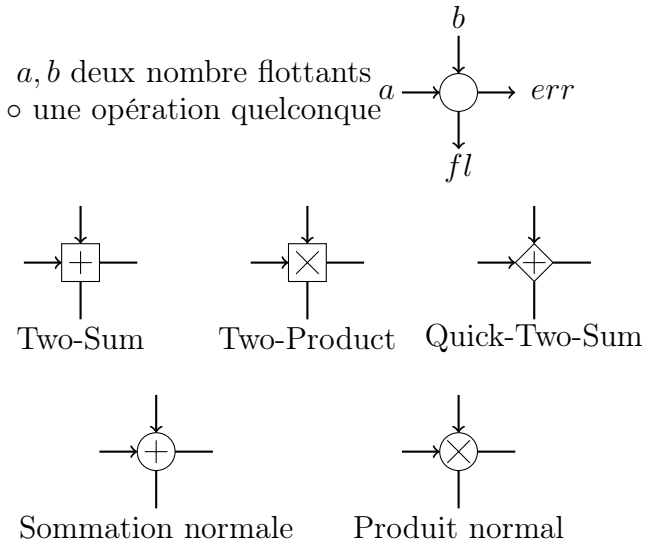

FIG. 3.1 – Notations des opérations

Renormalise2-nearest

L'EFT sur la sommation de ces 2 flottants nous renvoie un couple de deux flottants de même somme. Mais la relation entre les deux membres de ce couple est moins forte que la normalisation sous le mode d'arrondi au plus près imposée sur un double-simple. On n'a plus la relation (3.1). On peut réutiliser l'EFT sur la sommation de deux flottants en ajoutant quelques modifications pour implémenter le `Renormalise2-nearest`.

```
1   Renormalise2-nearest (a,b)
2       epsilon = 2^(-23)
3       if (|a| < |b|)
4           swap(a,b)
5       s = fl(a + b)
6       d = fl(s - a)
7       e = fl(b - d)
8       splus = fl(s + s * epsilon)
9       ulp = splus - s
10      eplus = e - ulp
11      if (|e| > |eplus|)
12          s = splus, e = eplus
13      if (|2 * b| < |d|)
14          s = a, e = b
15  return (s,e)
```

THÉORÈME 5. *Soit a et b deux nombre flottants de simple précision. Le résultat de l'algorithme* `Renormalise2-nearest` *sur ces deux flottants est un couple de deux flottants*

de simple précision (s, e) *qui satisfait :*

$$s + e = a + b \text{ et } |e| \leq \frac{1}{2} ulp(s)$$

L'égalité de la deuxième relation se trouve quand e *et* s *ont la même signe.*

Nous avons fait la démonstration de ce théorème qui se trouve dans l'Annexe A.

Il est évident que cet algorithme nous permet d'avoir une autre transformation exacte qui réagit exactement comme une transformation exacte avec le mode d'arrondi au plus près. Cet algorithme nous donne un résultat plus propre que l'algorithme `Two-Sum-toward-zero` parce que la contrainte entre les deux éléments du résultat est plus forte :

$$|e| \leq \frac{1}{2} ulp(s) < \frac{1}{2} \varepsilon |s|.$$

Dans la plupart de cas, les deux entrées de la renormalisation satisfont bien la condition $|a_0| \geq |a_1|$. Cela nous permet d'enlever les deux instructions 3, 4 de l'algorithme `Renormalise2-nearest`. Donc l'algorithme `Renormalise2-nearest` coûte 9 FLOPs.

Renormalise2-toward-zero

L'algorithme `Renormalise2-nearest` nous donne un résultat exact mais coûte très cher parce qu'il est implémenté sous le mode d'arrondi vers zéro. Pour le `Renormalise2-toward-zero`, on peut réutiliser le résultat de la transformation exacte sur la sommation mode d'arrondi vers zéro sans ajouter de nouvelle instructions.

```
1  Renormalise2-toward-zero (a,b)
2      if (|a| < |b|)
3          swap(a,b)
4      s = fl(a + b)
5      d = fl(s - a)
6      e = fl(b - d)
7  return (s,e)
```

L'algorithme `Renormalise2-toward-zero` est beaucoup plus simple que l'algorithme `Renormalise2-nearest`. Il coûte seulement 4 FLOPs. Dans la plupart de cas, les deux entrées de la renormalisation satisfont bien la condition $|a_0| \geq |a_1|$. Cela nous permet d'enlever les deux instructions 2, 3 de l'algorithme `Renormalise2-toward-zero`. Donc l'algorithme `Renormalise2-toward-zero` coûte 3 FLOPs.

D'après la démonstration de l'algorithme `Two-Sum-toward-zero` on peut facilement déduire que :

$$\begin{aligned} s &= fl(a + b) \\ e &= fl(err(a + b)). \end{aligned}$$

Avec le mode d'arrondi vers zéro on a :

$$|e| \leq |err(a + b)| < ulp(s),$$

et e a le même signe que $err(a+b)$, $err(a+b)$ a le même signe que s. D'où s et e ont le même signe.

Cela nous permet de constater que (s, e) satisfait la normalisation sur le double-simple sous le mode d'arrondi vers zéro. Le problème est que $err(a+b)$ n'est pas toujours représentable par un nombre flottant (par exemple $a = 2^{24}, b = 2^{-24}$). Donc (s, e) n'est pas le résultat exact. L'erreur de cet algorithme est causée par la troncation de $err(a+b)$. L'erreur de cet algorithme peut être estimé par le théorème suivant :

Théorème 6. *Soient a, b deux flottants de simple précision. Le résultat retourné par* `Renormalise2-toward-zero` *est un double-simple (s, e) qui satisfait :*

- *s, e ont le même signe,*
- *$|e| < ulp(s)$,*
- *$a + b = s + e + \delta$, avec δ est l'erreur de l'algorithme* `Renormalise2-toward-zero` *et*

$$\begin{aligned} |\delta| &\leq \frac{1}{2}\varepsilon^2|a+b| \\ &\leq 2^{-45}|a+b|. \end{aligned}$$

Démonstration Les deux première propriétés sont déjà démontrées juste avant. Soient sl, ml, el respectivement le signe, la matisse et l'exposant de e. On a :

$$e = sl \times ml \times 2^{el} \text{ et } 1 \leq ml < 2, ulp(r_l) = 2^{el} \times \varepsilon.$$

Soit eh l'exposant de s, on a :

$$ulp(s) = 2^{eh} \times \varepsilon.$$

$|r_l| < ulp(r_h)$ d'où :

$$\begin{aligned} |sl \times ml \times 2^{el}| &< 2^{eh} \times \varepsilon \\ \Leftrightarrow ml \times 2^{el} &< 2^{eh} \times \varepsilon \\ \rightarrow 2^{el} &< 2^{eh} \times 2^{1-p} \\ \rightarrow el &< eh + (1-p) \\ \rightarrow el &\leq eh + (1-p) - 1 \\ \rightarrow 2^{el} &\leq 2^{eh} \times 2^{1-p} \times 2^{-1} \\ \rightarrow 2^{el} &\leq \frac{1}{2} \times ulp(s) \\ \rightarrow 2^{el} &< \frac{1}{2} \times \varepsilon|s| \\ \rightarrow 2^{el} \times \varepsilon &< \frac{1}{2} \times \varepsilon^2|s| \\ \rightarrow ulp(e) &< \frac{1}{2} \times \varepsilon^2|s|. \end{aligned}$$

Avec le mode d'arrondi vers zéro on a :

$$\begin{aligned} s &= fl(a+b) \\ |s| &\leq |a+b|. \end{aligned}$$

D'après l'algorithme `Renormalise2-toward-zero`, l'erreur de cet algorithme est causée par la troncation de $err(a+b)$ ver e donc :

$$\begin{aligned} |\delta| &< ulp(e) \\ &< \frac{1}{2} \times \varepsilon^2 |s| \\ &< \frac{1}{2} \times \varepsilon^2 |a+b|. \end{aligned}$$

Dans la partie suivante, on va voir que la borne d'erreur produite dans le processus de calcul des algorithmes qui suivent est beaucoup plus grande que la borne d'erreur de `Renormalise2-toward-zero`. Donc nous avons décidé d'implémenter `Renormalise2-toward-zero` dans la bibliothèque double-simple pour augmenter la performance de la bibliothèque mais ne pas perdre beaucoup sa précision.

Renormalise3

Si le résultat intermédiaire se compose de 3 flottants, la renormalisation suit l'algorithme `Renormalise3` suivant qui est une variante de la méthode de renormalisation proposée par Yozo Hida, Xiaoye S.Li et David H.Bailey [5, p. 7].

```
1  Renormalise3 (a0, a1, a2)
2      (s, t2) = Quick-Sum-toward-zero (a1, a2)
3      (t0, t1) = Quick-Sum-toward-zero (a0, s)
4      t1 = fl(t1 + t2)
5      (b0, b1) = Renormalise2 (t0, t1)
6  return (b0, b1)
```

La fonction Renormalise2 peut être remplacée par une de ses deux versions en fonction du type de normalisation choisi. Chaque appel à `Quick-Sum-toward-zero` coûte 5 FLOPs, avec la renormalisation vers zéro cette méthode de renormalisation va coûter 13 FLOPs, qui est beaucoup plus que le `Renormalise2-toward-zero`. Malheureusement les conditions nécessaires pour que cet algorithme fonctionne bien sont inconnues [5, p. 7]. Mais il est indiqué aussi dans [5, p. 7] que cet algorithme est applicable sur des résultats produits par l'ensemble des algorithmes qui suivent.

3.2.3 La sommation

Double-simple + simple : la sommation d'un double-simple a avec un simple b suit le schéma de la figure 3.2.

Après les appels de `Two-Sum-toward-zero`, on a le résultat exact de la somme mais représenté par 3 flottants. Ce résultat exact est renormalisé pour récupérer le résultat en forme d'un double-simple. Donc l'exactitude de cet algorithme dépend de l'exactitude de la renormalisation. Si le `Renormalise3` fonctionne bien, le résultat de cet algorithme est exact dans le domaine des flottants de $[2p]$-bits.

Cet algorithme utilise 2 appels de `Two-Sum-toward-zero` et un appel de `Renormalise3`, donc au total il coûte 25 Flops. La performance de cet algorithme est déterminée par la performance de l'algorithme `Two-Sum-toward-zero` et la renormalisation. Si on enlève

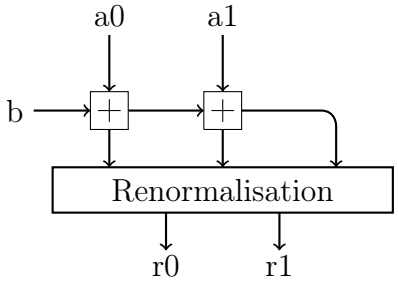

FIG. 3.2 – La sommation entre un double-simple et un simple

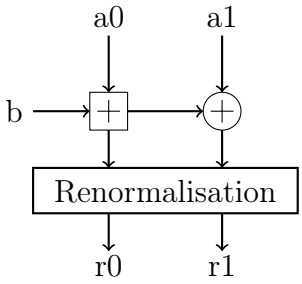

FIG. 3.3 – Algorithme réduit de la sommation entre un double-simple et un simple

le dernier terme du résultat intermédiaire, qui est l'erreur de la deuxième EFT sur la somme, la renormalisation va être réalisée par l'algorithme `Renormalise2-toward-zero` au lieu de `Renormalise3`, et le deuxième `Two-Sum-toward-zero` peut être remplacé par une sommation normale (parce que l'erreur de cette sommation n'est plus utilisée) (figure 3.3)

L'algorithme de la figure 3.3 est beaucoup plus simple, coûte seulement 9 FLOPs, et peut être décrit comme suit :

```
1       add_ds_s (a0, a1, b)
2           (t0, t1) = Two-Sum-toward-zero (a0, b)
3           t1 = t1 + b
4           (b0, b1) = Renormalise2-toward-zero (t0, t1)
5       return (b0, b1)
```

L'erreur de l'algorithme `add_ds_s` est exactement l'erreur de la deuxième sommation et la renormalisation. En négligeant les termes d'ordre supérieur on peut doubler la vitesse de cet algorithme.

THÉORÈME 7. *L'erreur de calcul de l'algorithme* `add_ds_s` *satisfait*

$$|err(a+b)| < \frac{2\varepsilon^2}{1-\varepsilon} \times |fl(a+b)| + \frac{\varepsilon^2}{2} \times |fl(a+b)|.$$

où ε est l'erreur relative en simple précision.

Double-simple + double-simple : la sommation de deux double-simples a et b suit le schéma dans la figure 3.4.

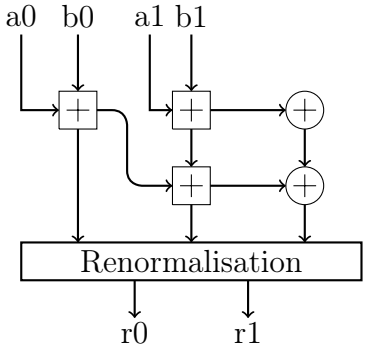

FIG. 3.4 – La sommation de deux double-simples

Cet algorithme utilise 3 `Two-Sum-toward-zero` et une sommation normale. À cause de cette sommation normale, on ne peut pas parvenir au résultat exact après la première phase avant la renormalisation. L'erreur causée par cette sommation normale est : $err0 = err(err(a_1+b_1)+err(fl(a_1+b_1)+err(a_0+b_0)))$. Cette erreur est très petite par rapport au résultat parce que :

- Si $err(a_0+b_0) = 0$ d'où $err(fl(a_1+b_1)+err(a_0+b_0)) = 0 \rightarrow err0 = 0$
- Si $err(a_0+b_0) \neq 0$ d'où on ne peut pas avoir la relation $0 < 1/2max(a_0,b_0) < min(a_0,b_0) < max(a_0,b_0)$ ni la relation $0 > 1/2max(a_0,b_0) > min(a_0,b_0) > max(a_0,b_0)$. Cela donne la relation $|a_0+b_0| \geq min(|a_0|,|b_0|)$ et $|a_0+b_0| \geq 1/2max(|a_0|,|b_0|)$. Donc $|err0|$ est un terme de $0(\varepsilon^4 \times |a_0+b_0|)$

Au total, cette fonction coûte 32 FLOPs (dont 18 pour trois `Two-Sum-toward-zero`, 13 pour la renormalisation et 1 pour la sommation normale).

Comme pour la sommation entre un double-simple et un simple, la mauvaise performance de cet algorithme est causée par le troisième terme du résultat intermédiaire. Donc l'amélioration de cet algorithme peut être réalisée aussi en enlevant ce terme. Cela veut dire que les deux derniers EFTs sont remplacés par deux sommations normales, et la renormalisation est réalisée par l'algorithme `Renormalise2-toward-zero` (figure 3.5). Cet algorithme réduit est le suivant :

```
1       add_ds_ds (a0, a1, b0, b1)
2           (t0, t1) = Two–Sum–toward–zero (a0, b0)
3           t2 = fl(a1 + b1)
4           t1 = fl(t1 + t2)
5           (b0, b1) = Renormalise2–toward–zero (t0, t1)
```

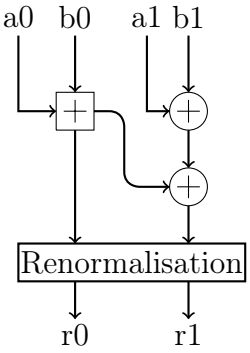

FIG. 3.5 – Algorithme réduit de la sommation de deux double-simples

```
6       return (b0, b1)
```

Avec deux sommations normales, un `Two-Sum-toward-zero` et un `Renormalise2-toward-zero`, cet algorithme coûte 11 FLOPs. L'erreur de cet algorithme est bornée par le théorème suivant [7, p. 18].

THÉORÈME 8. *Soit $a_h + a_l$ et $b_h + b_l$ deux double-simples. Soit $r_h + r_l$ le résultat retourné par l'algorithme* `add_ds_ds` *sur $a_h + a_l$ et $b_h + b_l$. L'erreur de cet algorithme δ satisfait :*

$$r_h + r_l = (a_h + a_l) + (b_h + b_l) + \delta$$

et est bornée par :

$$|\delta| < max(2^{-23} * |a_l + b_l|, 2^{-43} * |a_h + a_l + b_h + b_l|) + 2^{-45} * |a_h + a_l + b_h + b_l|.$$

3.2.4 La soustraction

La soustraction de deux double-simple $a-b$ est implémentée par la sommation $a+(-b)$. Pour cela, il nous faut seulement inverser le signe de toutes les composantes de b. Et puis, en suivant le même algorithme de sommation on peut récupérer la soustraction de deux double-simples.

3.2.5 Le produit

Le produit entre deux double-simples a et b est le produit entre deux sommes $a_0 + a_1$ et $b_0 + b_1$, donc le produit exact dispose de 4 composantes :

$$\begin{aligned} c &= (a_0 + a_1) \times (b_0 + b_1) \\ &= a_0 \times b_0 + a_1 \times b_0 + a_0 \times b_1 + a_1 \times b_1. \end{aligned}$$

La première composante $a_0 \times b_0$ est la composante la plus significative du résultat. C'est un terme en $\mathcal{O}(1)$. Utilisant l'EFT sur le produit, on transforme ce terme en une somme de deux simples :

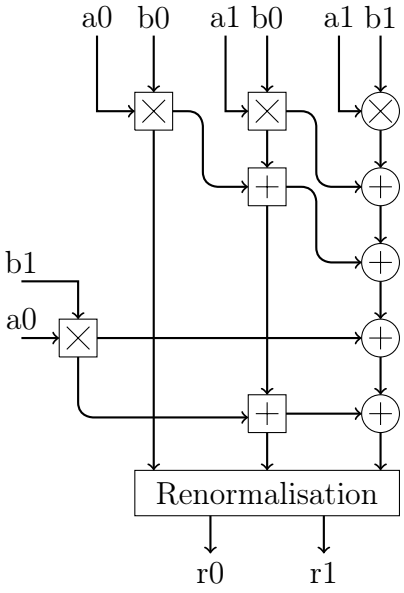

FIG. 3.6 – Le produit de deux double-simples

– $fl(a_0 \times b_0)$ un terme en $\mathcal{O}(1)$ qui est une estimation du résultat,
– $err(a_0 \times b_0)$ un terme en $\mathcal{O}(\varepsilon)$.

Les deux composantes suivantes sont des termes en $\mathcal{O}(\varepsilon)$. Cela signifie qu'en utilisant l'EFT sur le produit, on transforme chaque composante de ces deux nombres en une somme d'un terme en $\mathcal{O}(\varepsilon)$ et un terme $0(\varepsilon^2)$.

- $a_1 \times b_0 = fl(a_1 \times b_0) + err(a_1 \times b_0)$
- $a_0 \times b_1 = fl(a_0 \times b_1) + err(a_0 \times b_1)$

Avec le terme $err(a_0 \times b_0)$ on a trois termes en $\mathcal{O}(\varepsilon)$. Puisqu'on a seulement la transformation exacte pour une somme de deux flottants on doit utiliser deux `Two-Sum-toward-zero` pour transformer la somme de trois flottants qui nous donne un terme en $\mathcal{O}(\varepsilon)$ et deux termes $\mathcal{O}(\varepsilon^2)$ (correspondants aux deux erreurs des deux `Two-Sum-toward-zero`).

Au total, il nous reste cinq termes en $\mathcal{O}(\varepsilon^2)$. Pour cela, on se contente d'utiliser seulement des opérations normales (4 opérations) qui nous donnent un gain sur la performance (car le `Two-Sum-toward-zero` coûte 6 FLOPs) en acceptant une erreur de terme en $\mathcal{O}(\varepsilon^2)$.

Finalement, en appliquant le processus de renormalisation sur les trois termes récupérés on obtient le résultat du produit entre deux double-simples. Le schéma du processus entier est décrit dans la figure 3.6.

Selon le schéma de la figure 3.6, on peut facilement calculer le coût de cette fonction. On a trois `Two-Product`, deux `Two-Sum-toward-zero`, un produit normal, quatre sommations normales et une renormalisation. Donc le coût total est : $3 \times 2 + 2 \times 6 + 5 + 13 = 36$ FLOPs.

De la même façon que pour la sommation, on utilise la renormalisation de deux termes (`Renormalise2-toward-zero`) au lieu de trois termes (`Renormalise3`) pour améliorer cet algorithme. Cela veut dire que seule l'erreur du produit $a_0 \times b_0$ est intéressante, les autres

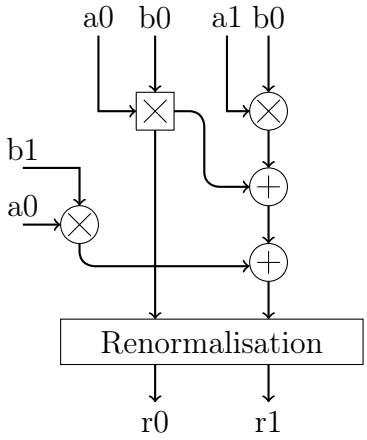

FIG. 3.7 – Algorithme réduit du produit de deux double-simples

erreurs étant ignorées afin d'améliorer la vitesse de calcul. Donc, excepté l'EFT sur le produit $a_0 \times b_0$ qui coûte 2 FLOPs, seules des opérations basiques sur les flottants sont utilisées (figure 3.7).

Cet algorithme réduit, appelé le `mul_ds_ds`, qui coûte seulement 8 FLOPs, sera utilisé pour implémenter notre bibliothèque. Il peut être décrit de la manière suivante :

```
1       mul_ds_ds (a0, a1, b0, b1)
2           (t0, t1) = Two-Product-FMA (a0, b0)
3           t2 = fl(a1 * b0)
4           t2 = fl(a0 * b1 + t2)
5           t1 = fl(t1 + t2)
6           (b0, b1) = Renormalise2-toward-zero (t0, t1)
7       return (b0, b1)
```

L'erreur de cet algorithme est bornée par la théorème suivante.

THÉORÈME 9. *Soit $a_h + a_l$ et $b_h + b_l$ deux double-simples. Soit $r_h + r_l$ le résultat retourné par l'algorithme `mul_ds_ds` sur $a_h + a_l$ et $b_h + b_l$. L'erreur de cet algorithme δ satisfait :*

$$|(r_h+r_l)-(a_h+a_l)\times(b_h+b_l)| < 2^{-43}\times|(a_h+a_l)\times(b_h+b_l)|+9\times 2^{-68}\times|(a_h+a_l)\times(b_h+b_l)|$$

3.2.6 La division

Pour calculer la division de deux double-simples, nous utilisons l'algorithme traditionnelement enseigné à l'école primaire.

Soit $a = (a_0, a_1)$ et $b = (b_0, b_1)$ deux double-simples. Pour calculer la division de a par b, on calcule d'abord le quotient approximatif par :

$$q_0 = a_0/b_0.$$

C'est une division en simple précision. Nous calculons ensuite le reste : $r = a - q_0 \times b$, qui nous permet de calculer le terme de correction par :

$$q_1 = r/b_0.$$

Comme pour la sommation et le produit, on s'arrête a deux termes du résultat pour économiser le temps de calcul en compromettant quelque bits de précision.

Dans l'algorithme décrit précédemment, il faut utiliser un produit d'un double-simple avec un simple et une soustraction pour le $a - q_0 \times b$. Comme avec le `fma_ds_ds`, on n'utilise pas le produit et la sommation de double-simples mais on les implémente directement pour éviter d'utiliser plusieurs fois le processus de renormalisation. De plus, l'algorithme suivant se base sur la fonction FMA.

```
1  div_ds_ds(a,b)
2      p0 = fl(a0 / b0)
3      tmp1 = fl(a0 - q0 * b0)
4      tmp2 = fl(a1 - q0 * b1)
5      r = fl(tmp1 + tmp2)
6      p1 = fl(r / b_0)
7      (q0,q1) = Renormalise2-toward-zero(p0, p1)
8  return (q0,q1)
```

Nous proposons le théorème suivant qui estime la précision de cet algorithme.

Théorème 10. *Soit $a = (a_0, a_1)$ et $b = (b_0, b_1)$ deux double-simples, ε l'erreur relative en simple précision, et ε_1 la borne d'erreur relative de la division en simple précision avec $\mathcal{O}(\varepsilon_1) = \mathcal{O}(\varepsilon)$. L'erreur de l'algorithme `div_ds_ds` est bornée par :*

$$\frac{|div\_ds\_ds(a,b)) - a/b|}{|a/b|} < \varepsilon^2 \times (6.5 + 7 \times \varepsilon_1/\varepsilon + 2 \times (\varepsilon_1/\varepsilon)^2) + \mathcal{O}(\varepsilon^3)).$$

Nous avons fait la preuve de ce théorème. Elle se trouve dans l'Annexe A.

Dans plusieurs cas, la borne d'erreur relative de la division en simple précision est aussi l'erreur relative en simple précision. Cela veut dire $\varepsilon_1 = \varepsilon$. Dans ce cas, la borne d'erreur relative de la division de double-simples est :

$$\begin{aligned} \frac{|q - a/b|}{|a/b|} &< \varepsilon^2 \times (6.5 + 7 + 2) + \mathcal{O}(\varepsilon^3) \\ &< 15.5 \times \varepsilon^2 + \mathcal{O}(\varepsilon^3) \end{aligned}$$

Cette inéquation signifie que notre algorithme de division de double-simples permet de récupérer un résultat précis à 42 bits. Pour augmenter la précision de l'algorithme de division un terme de correction q_2 est calculé de la même façon : $r = (a - q \times b); q_2 = r_0/b_0$ ce qui augmente beaucoup la complexité de l'algorithme ou bien diminue la performance de cette opération. Donc pour l'intérêt de la performance, comme avec les deux autres opérations (sommation et produit) la vitesse de l'algorithme est améliorée en compromettant la précision de l'algorithme.

3.3 Implémentation

Le SPE (Synergistic Processor Element) d'un processeur CELL contient un processeur SIMD (Single Input Multiple Data) 32bits 4-voies. Il supporte le calcul vectoriel de petite dimension. Il dispose d'un ensemble important de registres de 128 bits (128 registres). Il supporte des calculs sur des vecteurs de

- 16 char / unsigned char
- 4 int / unsigned int
- 4 float
- 2 double

Chaque opération FMA sur des vecteurs coûte un FLOP en simple précision. Dans la pratique, le SPE n'implémente pas des opérations arithmétiques sur des scalaires. Des opérations scalaires sont implémentées en utilisant l'intervalle préférée qui est une position dans un vecteur. Cette procédure exige des opérations supplémentaires pour décaler l'élément dans l'intervalle préféré et puis le repositionner vers la position originale.

3.3.1 Représentation

Dans cette partie, nous présentons seulement l'implémentation des opérations vectorielles sur des flottants pour bien tirer parti du processeur SIMD du CELL et pour faire des optimisations en utilisant les deux pipelines du FPU. Chaque double-simple se compose de deux flottants de simple précision. Donc un vecteur de 16B du processeur SIMD peut contenir 2 double-simples. Comme le processeur SIMD ne supporte pas des vecteurs de double-simple nous implémentons ce type par une union qui contient à la fois un couple de deux double-simples, et un vecteur des flottants de simple précision.

Remarque : par la suite, un vecteur de deux double-simples est aussi utilisé comme un vecteur de quatre flottants de simple précision.

3.3.2 Les EFTs

L'EFT sur le produit est très simple avec un produit normal et un FMA. Donc ici, nous abordons seulement l'EFT sur la sommation de deux vecteurs de flottants de simple précision.

La sommation

Entrée : deux vecteurs de flottants de simple précision.

Sortie : deux vecteurs, qui sont respectivement le vecteur du résultat et le vecteur de l'erreur.

Comme on a déjà constaté dans la partie précédente, l'algorithme `Two-Sum-toward-zero` est choisi pour implémenter l'EFT sur la sommation. Cet algorithme commence par un branchement et un échange. Comme le SIMD signifie la même instruction à plusieurs données, ce branchement nous empêche de l'utiliser. Donc il faut d'abord éliminer ce branchement. La procédure d'élimination des branchements s'écrit de la manière suivante [2] :

- évaluer la condition de branchement. Le résultat est stocké dans un vecteur `comp` de type `unsigned int`, dont un entier de valeur zéro signifie que la condition n'est pas

a	a1	a2	a3	a4
b	b1>a1	b2<a2	b3=a3	b4>a4
comp=spu_cmpabsgt(b,a)				
	FFFFFFFF	00000000	00000000	FFFFFFFF
$hi = spu\_sel(a, b, comp)$				
	b1	a2	a3	b4
$lo = spu\_sel(b, a, comp)$				
	a1	b2	b3	a4

FIG. 3.8 – Exemple du résultat de l'échange deux vecteurs

satisfaite et un entier de valeur `MAX_UNSIGNED_INT` (tous les bits valent 1) signifie que la condition est satisfaite.

- calculer les valeurs des deux branches `val_1` (si la condition est satisfaite) et `val_2` (si la condition n'est pas satisfaite).
- sélectionner la bonne valeur en fonction du vecteur de condition. Le processeur SPE fournit une fonction de sélection de bit.

$$d = spu\_sel(val\_2, val\_1, comp)$$

Pour chaque bit dans le vecteur *comp* de 128-bit, le bit correspondant de val_1 ou val_2 est sélectionné. Si le bit est 0, le bit correspondant de val_2 est sélectionné, sinon le bit de val_1 est sélectionné. Le résultat est retourné dans le vecteur d.

Par exemple, dans notre cas, le branchement et l'échange peuvent être considérés comme la sélection du plus grand nombre et le plus petit nombre à partir de deux nombres donnés, donc il peut s'écrire de la manière suivante :

```
1   comp = spu_cmpabsgt(b,a)
2   hi = spu_sel(a, b, comp)
3   lo = spu_sel(b, a, comp)
```

Un exemple plus concret est donné dans la figure 3.8.

Selon le tableau 2.1 du chapitre 2 le `spu_cmpabsgt` coûte 2 cycles d'horloge et le `spu_sel` coûte seulement 2 cycles d'horloge. Les deux instructions des lignes 2 et 3 dans le code juste avant sont totalement indépendantes, donc il n'y a pas d'attente entre le lancement de ces deux instructions. Elles doivent seulement attendre le résultat de l'instruction de comparaison de la ligne 1. Au total, ces trois instructions coûtent seulement 5 cycles d'horloge ce qui est moins que le temps de calcul d'une opération flottante en simple précision.

En utilisant ce code pour faire de l'échange, et en appliquant la même procédure avec le dernier branchement de l'algorithme `Two-Sum-toward-zero`, l'algorithme `Two-Sum-toward-zero` peut être réécrit comme suit :

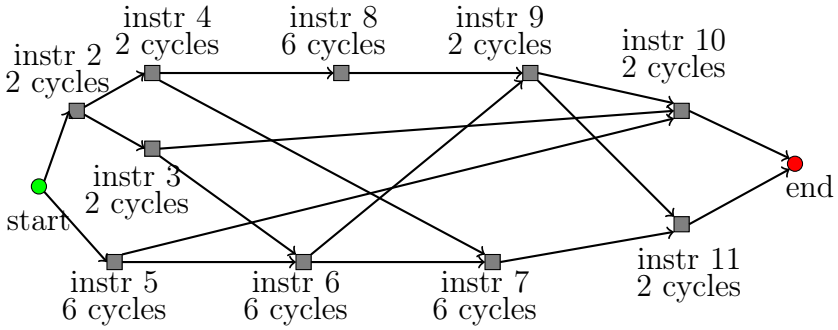

FIG. 3.9 – La dépendance entre les instructions de l'algorithme `Two-Sum-toward-zero`

```
1   Two–Sum–toward–zero (a,b)
2       comp = spu_cmpabsgt(b,a)
3       hi = spu_sel(a, b, comp)
4       lo = spu_sel(b, a, comp)
5       s = spu_add(a , b)
6       d = spu_sub(s , hi)
7       e = spu_sub(lo , d)
8       tmp = spu_mul(2 , lo)
9       comp = spu_cmpabsgt(d, tmp)
10      s = spu_sel(s, hi, comp)
11      e = spu_sel(e, lo, comp)
12  return (s,e)
```

Remarquons que les variables a et b de l'instruction de la ligne 5 ne sont pas remplacées par hi et lo. En effet le calcul de $hi+lo$ ou de $a+b$ est identique. En gardant a et b, le lien entre l'instruction de la ligne 5 et les instructions des lignes 2,3,4 est relâché. D'ailleurs, comme les instructions des lignes 2, 3, 4 ensemble coûtent 5 cycles elles peuvent être entièrement cachées par l'instruction de la ligne 5 qui coûte 6 cycles. Si on remplace les variables a et b de la ligne 4 par hi et lo, l'instruction de la ligne 4 doit attendre la fin des instructions des trois premières lignes avant d'être lancée. La dépendance des instructions de cet algorithme est décrite dans la figure 3.9. Écrit sous cette forme, le compilateur détecte l'indépendance des instructions et les exécute en parallèle.

Selon le schéma de la figure 3.9, le temps de calcul de cet algorithme est de **20 cycles d'horloge**, qui correspond à moins de 4 opérations flottantes séquentielles en simple précision. Cela signifie que l'algorithme `Two-Sum-toward-zero` est plus rapide que l'algorithme `Two-Sum` qui est implémenté par 6 sommations forcément séquentielles (les entrées d'une instruction dépendent du résultat de l'instruction précédente). Et puis, comme les trois premières instructions sont entièrement cachées par l'instruction de la ligne 5, il n'y a pas de différence par rapport au temps de calcul entre le `Two-Sum-toward-zero` et le `Quick-Sum-toward-zero`.

Comptant aussi le temps de récupérer et de stocker des arguments, un appel à cet

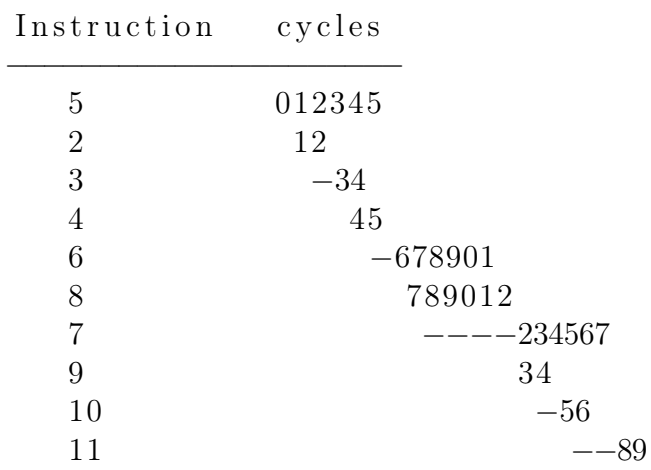

FIG. 3.10 – Le processus de calcul de l'algorithme `Two-Sum-toward-zero`

algorithme `Two-Sum-toward-zero` coûte **28 cycles d'horloge**.

Optimiser

Dans le schéma de la figure 3.10, on peut voir que dans l'algorithme `Two-Sum-toward-zero` il nous reste plusieurs cycles d'horloge où aucune instruction est lancée (marqués par un tiret). Cela veut dire que le pipeline n'est pas bien tiré parti. C'est à cause de la dépendance entre les instructions de cet algorithme.

On peut résoudre ce problème en suivant la même idée que le déroulement de boucle, effectuant plusieurs opérations au lieu d'effectuer une seule opération. Dans notre cas, avec le restreint de la mémoire locale des SPEs, nous implémentons une fonction appelée le `Two-Sum-toward-zero-2` qui effectue 8 sommations exactes en même temps. Les cycles non utilisés dans le processus de calcul d'une sommation vont être utilisés dans le processus de calcul d'une autre sommation.

Dans la pratique, la fonction `Two-Sum-toward-zero-2` coûte 32 cycles d'horloge, qui est juste un peu plus que la fonction `Two-Sum-toward-zero-2` mais effectue 8 sommations exactes en même temps. De plus la fonction `Two-Sum-toward-zero-2` ne gaspille que 2 cycles d'horloge dans le processus de calcul.

3.3.3 La renormalisation

Renormalise2-nearest

Par rapport à la transformation exacte sur la somme, il faut ajouter quelques instructions pour tester si le résultat calculé est conforme au résultat calculé avec le mode d'arrondi au plus près.

```
1  Renormalise2-nearest (a,b)
2      epsilon = 0x0.000002fp0
3      comp = spu_cmpabsgt(b,a)
4      hi = spu_sel(a, b, comp)
5      lo = spu_sel(b, a, comp)
```

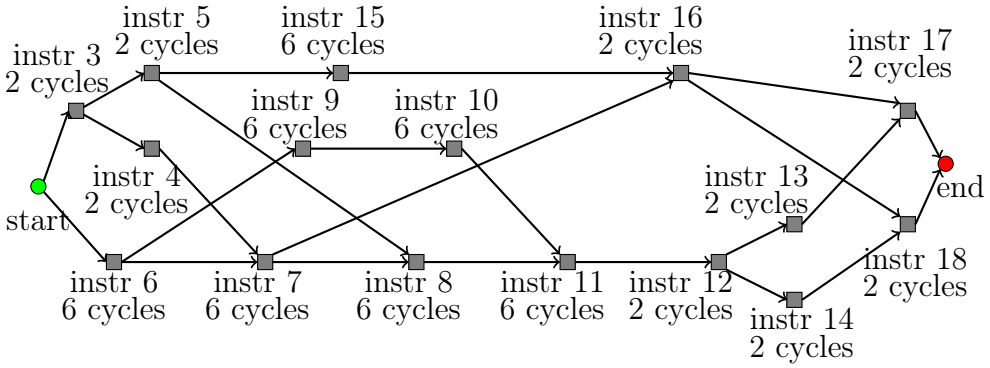

FIG. 3.11 – La dépendance des instructions de l'algorithme `Renormalise2-nearest`

```
6           s = spu_add(a , b)
7           d = spu_sub(s , hi)
8           e = spu_sub(lo , d)
9           splus = spu_madd(s , epsilon , s)
10          ulp = spu_sub(splus , s)
11          eplus = spu_sub(e , ulp)
12          comp1 = spu_cmpabsgt(e, eplus)
13          s = spu_sel(s, splus, comp1)
14          e = spu_sel(e, eplus, comp1)
15          tmp = spu_mul(2 , lo)
16          comp = spu_cmpabsgt(d, tmp)
17          s = spu_sel(s, hi, comp)
18          e = spu_sel(e, lo, comp)
19      return (s,e)
```

Les nouvelles instructions insérées sont des instructions des lignes 9, 10, 11, 12, 13, 14. C'est intéressant de noter que les instructions des lignes 9, 10 sont indépendantes des instructions des lignes 7,8 donc ils peuvent être exécutées en parallèle. Comme les fonctions `spu_cmpabsgt` et `spu_sel` coûtent chacunes 2 cycles d'horloge, l'algorithme `Renormalise2-nearest` ne coûte pas beaucoup plus cher que l'algorithme `Two-Sum-toward-zero`.

En utilisant le schéma de la figure 3.11 qui décrit la dépendance entre les instructions de l'algorithme `Renormalise2-toward-zero`, on peut estimer le temps d'exécution de cet algorithme à **32 cycles d'horloge**.

En utilisant l'outil `spu_timing` d'IBM, la fonction `Renormalise2-toward-zero` coûte **41 cycles d'horloge**.

Renormalise2-toward-zero

L'algorithme `Renormalise2-toward-zero` est beaucoup plus simple que l'algorithme `Renormalise2-nearest` avec seulement 6 opérations.

```
1  Renormalise2-toward-zero (a,b)
2     s = spu_add(a , b)
3        comp = spu_cmpabsgt(b,a)
4        hi = spu_sel(a, b, comp)
5        lo = spu_sel(b, a, comp)
6     d = spu_sub(s , hi)
7     e = spu_sub(lo , d)
8  return (s,e)
```

Comme avec `Renormalise2-nearest`, les instructions des lignes 3,4,5 de l'algorithme `Renormalise2-toward-zero` sont cachées par l'instruction de la ligne 2. Donc l'algorithme `Renormalise2-toward-zero` coûte seulement 18 cycles d'horloge.

En comptant aussi le temps de chargement et de stockage des paramètres, l'algorithme `Renormalise2-toward-zero` coûte 26 cycles d'horloge.

3.3.4 La sommation de double-simples - version 1

La version 1 de la sommation effectue la sommation de deux vecteurs de double-simples, ($vect\_a$ et $vect\_b$) soit deux sommations de double-simples puisque chaque vecteur de double-simples contient deux double-simples.

Suivant l'algorithme `add_ds_ds`, il faut d'abord calculer le résultat informatique et l'erreur du terme $a_0 + b_0$ ce qui consiste à appliquer la fonction `Two-Sum-toward-zero` présentée ci-dessus sur les deux vecteur $vect\_a$ et $vect\_b$ (coût 28 cycles). Le sucoût lié à l'appel d'une fonction, nous insérons directement le code de `Two-Sum-toward-zero` dans la fonction `add_ds_ds`. Donc on peut considérer que cette fonction coûte seulement 20 cycles. Cette fonction nous donne non seulement le résultat et l'erreur du terme $a_0 + b_0$ mais aussi du terme $a_1 + b_1$. L'erreur du terme $a_1 + b_1$ n'est pas utilisé dans cet algorithme, mais le résultat du terme $a_1 + b_1$ additionné de l'erreur du terme $a_0 + b_0$ est utilisé par la renormalisation.

Malheureusement, ces deux valeurs ne sont pas situées à la même position dans les vecteurs, parce qu'elles correspondent aux positions de différentes composantes de double-simple dans les vecteurs. Pour récupérer la sommation de ces deux valeurs, il faut d'abord les mettre dans les mêmes positions. Cela est réalisé en utilisant la fonction `spu_shuffle` du processeur SPU qui nous permet de former un vecteur à partir des octets de deux autres vecteurs et un vecteur de sélection. Cette fonction coûte 4 cycles. Pour faciliter le processus de calcul après, tous les résultats intermédiaires vont être décalés dans la première position (pour le premier double-simple de vecteur) et la troisième position (pour le deuxième double-simple de vecteur) des vecteurs. Cette procédure est décrite en détail dans le schéma de la figure 3.12.

Il est intéressant de remarquer ici que cet échange doit attendre la fin du `Two-Sum-toward-zero` avant d'être lancé, cependant il exige seulement la valeur du terme $a_1 + b_1$. Cette valeur est déjà calculée dans l'algorithme `Two-Sum-toward-zero`, car comme on implémente directement le `Two-Sum-toward-zero`, on ne doit pas attendre la fin de `Two-Sum-toward-zero` pour récupérer cette valeur. Donc cet échange peut être réalisé dans le processus de calcul du `Two-Sum-toward-zero` et va être caché par le `Two-Sum-toward-zero`.

La sommation de l'erreur du terme $a_0 + b_0$ et le résultat du terme $a_1 + b_1$ (déjà dé-

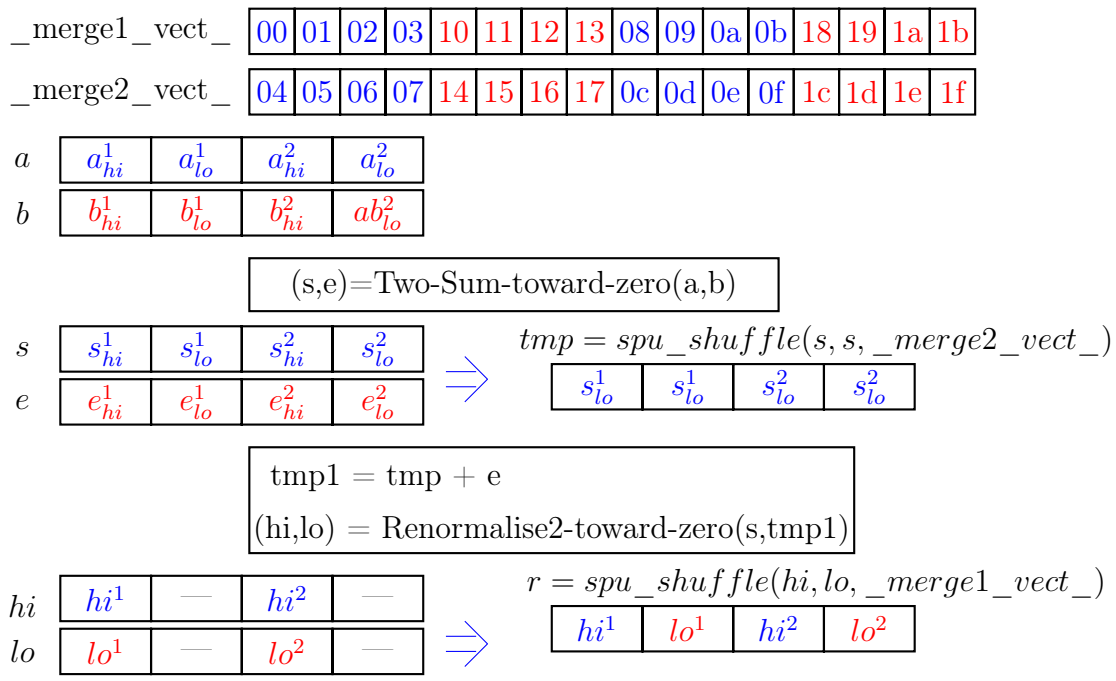

FIG. 3.12 – La sommation de deux vecteurs de double-simples version 1

calé dans les mêmes positions) avec le résultat du terme $a_0 + b_0$ sont des entrées pour la renormalisation, qui est réalisée par un `Renormalise2-toward-zero`. Comme avec le `Two-Sum-toward-zero`, ici nous implémentons directement l'algorithme `Renormalise2-toward-zero` pour éviter le temps d'invocation d'une fonction. Cela veut dire que la renormalisation coûte seulement 18 cycles. Son résultat est deux vecteurs qui représentent respectivement les premières composantes et les deuxièmes composantes des double-simples du résultat. Alors il faut utiliser encore une autre fois la fonction `spu_shuffle` pour mettre ces composantes dans un seul vecteur du résultat. Cette opération coûte 4 cycles d'horloge.

Cet algorithme s'appelle le `add_ds_ds_vect` et s'écrit comme suit :

```
1  add_ds_ds_vect (vect_a,vect_b)
2      (s , e) = Two-Sum-toward-zero(vect_a, vect_b)
3      tmp = spu_shuffle(s, s, _switch_vect_)
4      tmp1 = spu_add(tmp , e)
5      (hi , lo) = Renormalise2-toward-zero(s, tmp1)
6      res = spu_shuffle(hi, lo, _merge1_vect_)
7  return res
```

Avec un `Two-Sum-toward-zero`, un `Renormalise2-toward-zero`, une sommation normale et deux `spu_shuffle` (dont un `spu_shuffle` est caché en cours de calcul), cette fonction coûte $20+18+4+6 = 48$ cycles et s'opère sur deux vecteurs de double-simples.

La dépendance entre les instruction de cet algorithme est décrite dans la figure 3.13.

En utilisant l'outil `spu_timing` d'IBM, le temps réel de calcul de cette fonction est de **50 cycles d'horloge**.

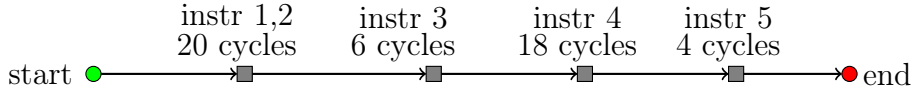

FIG. 3.13 – La dépendance entre les instruction de l'algorithme add_ds_ds_vect

3.3.5 Le produit de double-simples - version 1

Comme pour la sommation, cette version du produit effectue le produit de deux vecteurs de double-simples (*vect_a* et *vect_b*) soit deux produits de double-simples. Le schéma de calcul est décrit dans le schéma de la figure 3.14.

Suivant l'algorithme `mul_ds_ds`, il faut d'abord calculer le résultat informatique et l'erreur du terme $a_0 \times b_0$ ou appliquer la fonction `Two-Product-FMA` présentée précédemment sur les deux vecteur *vect_a* et *vect_b*, qui coûte 2 FLOPs soit 12 cycles. Cette procédure calcule aussi bien le résultat et l'erreur du terme $a_0 \times b_0$ que le résultat et l'erreur du terme $a_1 \times b_1$. Mais le résultat et l'erreur du terme $a_1 \times b_1$ ne sont pas utilisés dans cet algorithme.

Pour les termes $a_0 \times b_1$ et $a_1 \times b_0$, il nous faut seulement échanger la position des composantes b_0, b_1 (en utilisant la fonction `spu_shuffle`) et effectuer le produit sur le vecteur *vect_a* et le vecteur *vect_b* (déjà décalé). Cela nous donne un vecteur qui contient des termes $a_0 \times b_1$ aux positions 0, 2 et des termes $a_1 \times b_0$ aux positions 1, 3.

Comme on a besoin de la sommation entre les termes $a_0 \times b_1$ et les termes $a_1 \times b_0$, il faut créer un vecteur qui contient des termes $a_1 \times b_0$ aux positions 0 et 2 en utilisant la fonction `spu_shuffle`.

La sommation des trois termes $a_0 \times b_1$, $a_1 \times b_0$ et l'erreur du $a_0 \times b_0$ avec le résultat informatique du terme $a_0 \times b_0$ forment les entrées pour la renormalisation.

Comme avec le `add_ds_ds_vect`, la renormalisation renvoie deux vecteurs qui représentent respectivement les premières composantes et les deuxièmes composantes des double-simples du résultat. Donc il faut utiliser encore une autre fois la fonction `spu_shuffle` pour mettre ces composantes en ordre dans le vecteur résultat. Cette opération coûte 4 cycles d'horloge.

Cet algorithme s'appelle le `mul_ds_ds_vect` et s'écrit comme suit :

```
1   mul_ds_ds_vect (vect_a,vect_b)
2       p, e = Two-Product-FMA(vect_a, vect_b)
3       b_re = spu_shuffle(vect_b, vect_b, _switch_vect_)
4       p1 = spu_mul(vect_a , b_re)
5       p2 = spu_madd(vect_a , b_re , e)
6       p1 = spu_shuffle(p1, p1, _switch_vect_)
7       tmp1 = spu_add(p1 , p2)
8       (hi , lo) = Renormalise2-toward-zero(p , tmp1)
9       res = spu_shuffle(hi, lo, _merge1_vect_)
10  return res
```

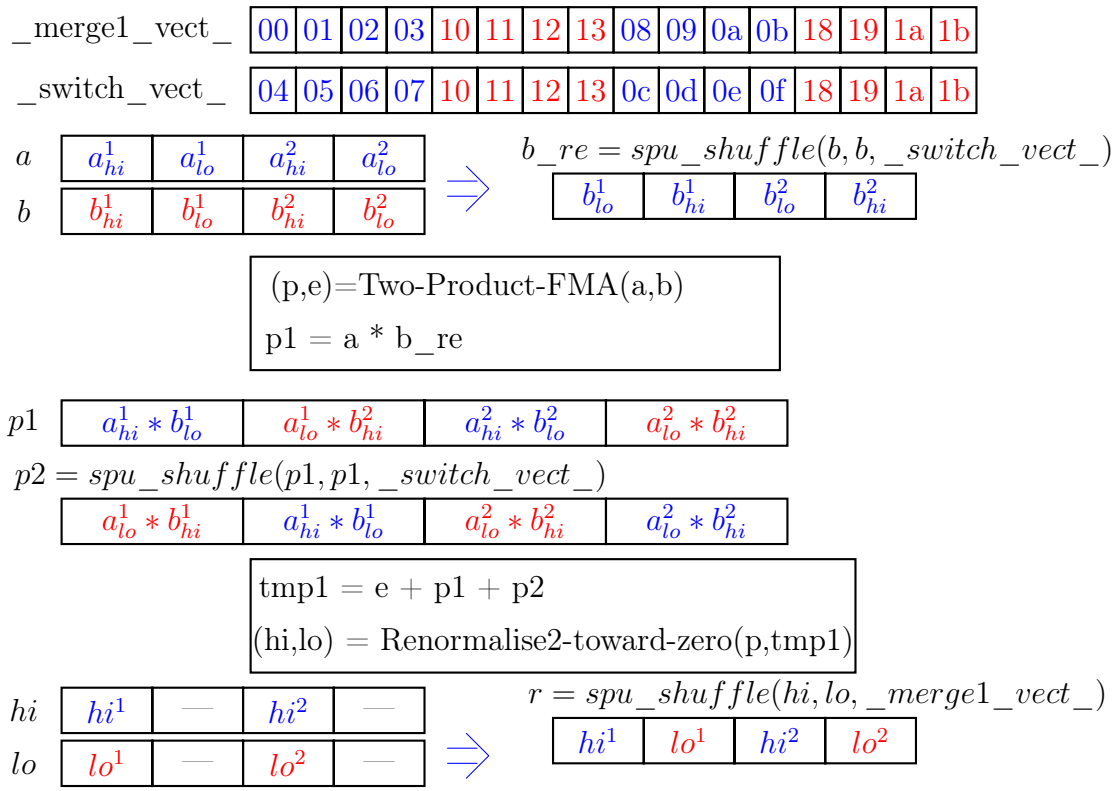

FIG. 3.14 – La multiplication de deux vecteurs de double-simples version 1

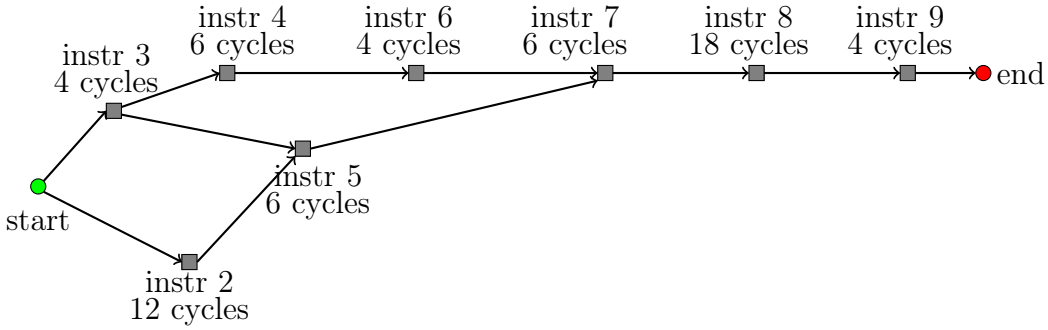

FIG. 3.15 – La dépendance entre les instructions de l'algorithme mul_ds_ds_vect

La dépendance entre les instructions de cet algorithme est décrite dans la figure 3.15. À partir de cette dépendance on peut déduire que l'algorithme `mul_ds_ds_vect` coûte **46 cycles d'horloge**. Il effectue deux produits de double-simples.

En utilisant l'outil `spu_timing` d'IBM, le temps réel de calcul de cette fonction est de **49 cycles d'horloge**.

3.3.6 Version 2 des opérations sommation et produit

Dans la section précédente, on a implémenté des opérations entre deux vecteurs de double-simples. Cela signifie que chaque opération affecte deux double-simples. Ces algorithmes tirent parti du processeur SIMD du SPE en manipulant plusieurs éléments en même temps. Mais on gaspille encore beaucoup de performance parce que les opérations effectuées sur les deux composantes de double-simple ne sont pas identiques. Dans plusieurs opérations, on n'utilise que les éléments dans les première et troisième position des vecteurs.

Pour tirer parti du SIMD, il faut utiliser tous les quatre éléments d'un vecteur. Ils doivent être identiques au sens d'opérations effectuées. Il vaut donc mieux effectuer deux opérations vectorielles en même temps, parce que deux vecteurs de double-simples contiennent quatre première composantes et quatre deuxièmes composantes de double-simple qui sont respectivement absolument identiques.

Dans cette partie, les opérations vectorielles sur les double-simples (sommation et produit) seront implémentées sur quatre vecteurs d'entrée soient $vect\_a1$, $vect\_a2$, $vect\_b1$ et $vect\_b2$ et le résultat sont deux vecteurs de double-simple $vect\_c1$, $vect\_c2$ tels que : $vect\_c1 = vect\_a1 \circ vect\_b1$ et $vect\_c2 = vect\_a2 \circ vect\_b2$ pour $\circ \in (+, \times)$

Selon le schéma de la figure 3.16, en utilisant deux fois la fonction `spu_shuffle`, deux vecteurs de double-simple peuvent être transformés dans deux vecteurs composantes, dont un vecteur contient les quatre premières composantes et un vecteur contient les quatre deuxièmes composantes (figure 3.16).

En appliquant deux fois la fusion sur deux couples de vecteurs d'entrée $(vect\_a1, vect\_a2)$ et $(vect\_b1, vect\_b2)$, on récupère respectivement deux couples de vecteurs (a_0, a_1) et (b_0, b_1) qui sont les entrées pour les algorithmes `add_ds_ds` et `mul_ds_ds`.

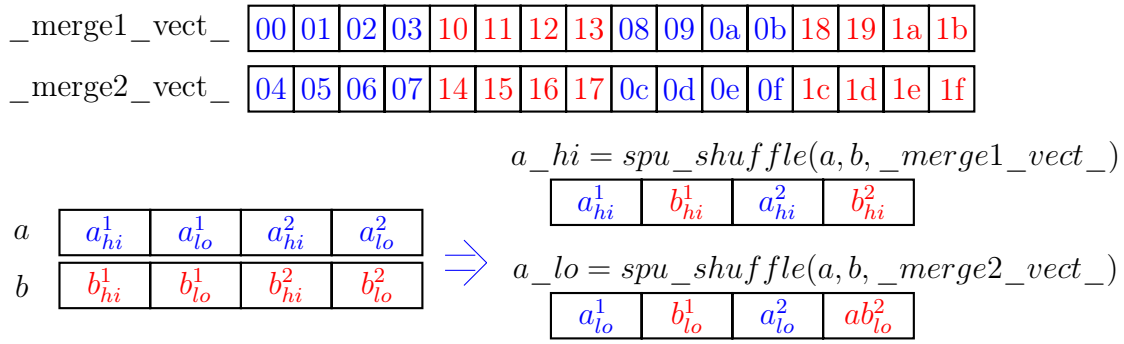

FIG. 3.16 – Fusion de deux vecteurs

Le résultat de ces algorithmes est aussi deux vecteurs qui contiennent respectivement les quatre parties hautes et les quatre parties basses des deux vecteurs résultat. Afin de reconstruire les vecteurs résultat au bon format, il est nécessaire d'appeler `spu_shuffle` deux fois.

La sommation

En suivant la procédure décrite précédemment, cela nous conduit à l'algorithme `add_ds_ds_2vect` ci-dessous qui effectue quatre sommations de double-simples en même temps. Il est décrit en détail dans le schéma de la figure 3.17.

```
1   add_ds_ds_2vect (vect_a1, vect_a2, vect_b1, vect_b2)
2       a_hi = spu_shuffle(vect_a1, vect_a2, _merge1_vect_)
3       a_lo = spu_shuffle(vect_a1, vect_a2, _merge2_vect_)
4       b_hi = spu_shuffle(vect_b1, vect_b2, _merge1_vect_)
5       b_lo = spu_shuffle(vect_b1, vect_b2, _merge2_vect_)
6       (s, e) = Two-Sum-toward-zero (a_hi, b_hi)
7       t1 = spu_add(a_lo , b_lo)
8       tmp = spu_add(t1 , e)
9       (hi, lo) = Renormalise2-toward-zero (s , tmp)
10      vect_c1 = spu_shuffle(hi, lo, _merge1_vect_)
11      vect_c2 = spu_shuffle(hi, lo, _merge2_vect_)
12  return (vect_c1, vect_c2)
```

L'instruction de la ligne 7 de l'algorithme `Two-Sum-toward-zero` est indépendante de l'instruction de la ligne 6. Donc grâce aux caractéristiques du pipeline entier, l'instruction de la ligne 7 va être cachée par le `Two-Sum-toward-zero` de la ligne 6 qui coûte 20 cycles.

La dépendance entre les instructions de cet algorithme est décrite dans la figure 3.18.

En utilisant l'outil `spu_timing` d'IBM, le temps réel de calcul de cette fonction est de **64 cycles d'horloge**.

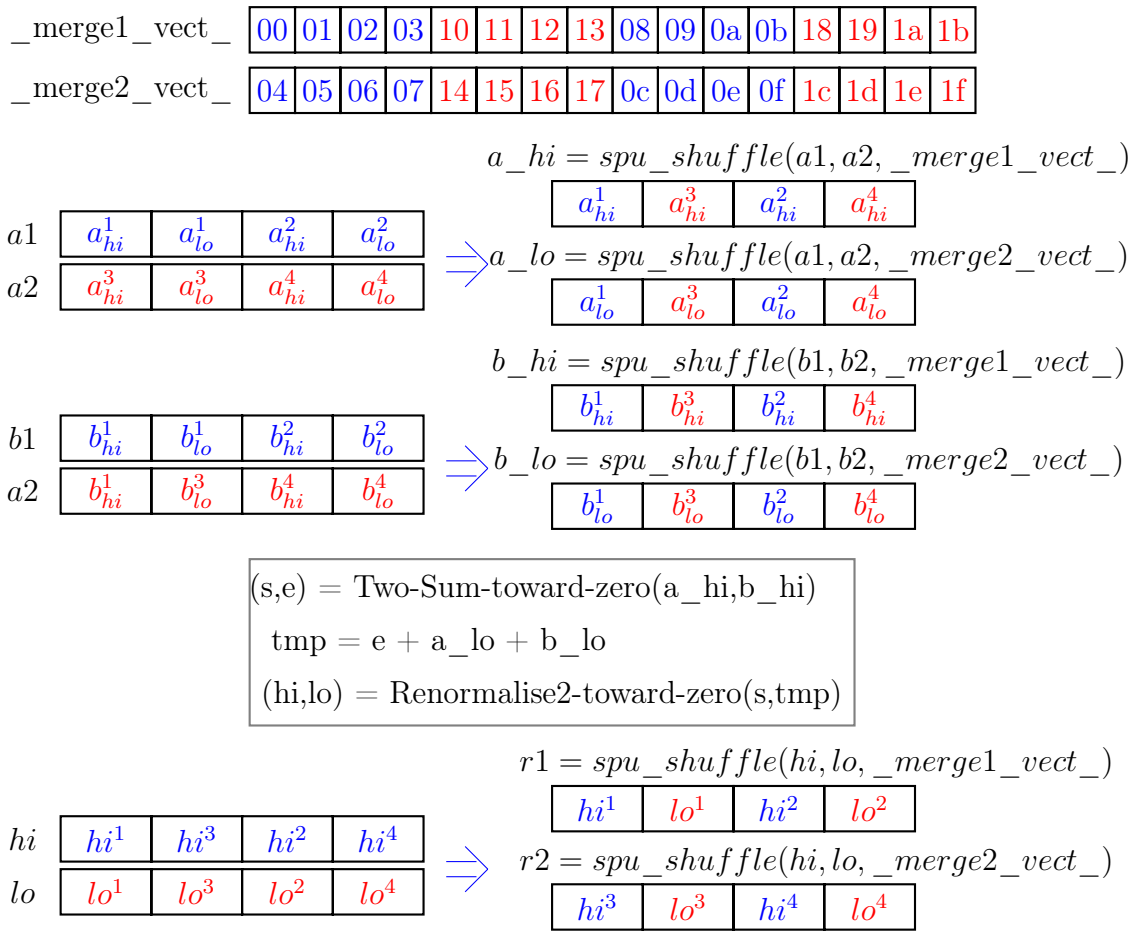

FIG. 3.17 – La sommation de deux couple de vecteurs de double-simples

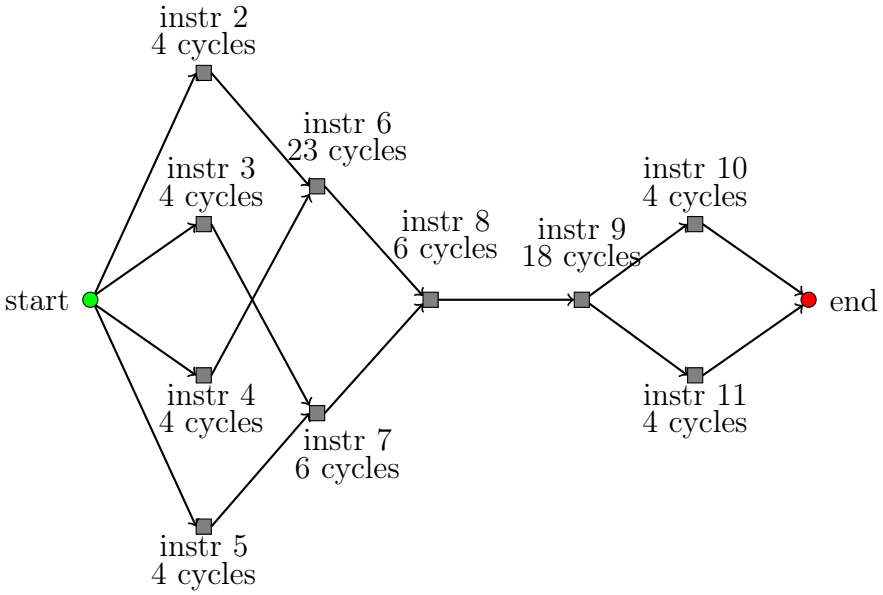

FIG. 3.18 – La dépendance entre les instruction de l'algorithme `add_ds_ds_2vect`

Le produit

Comme avec la sommation, l'algorithme `mul_ds_ds_2vect` qui effectue quatre produits de double-simples en même temps s'écrit comme suit :

```
1   mul_ds_ds_2vect (vect_a1, vect_a2, vect_b1, vect_b2)
2       a_hi = spu_shuffle(vect_a1, vect_a2, _merge1_vect_)
3       b_hi = spu_shuffle(vect_b1, vect_b2, _merge1_vect_)
4       b_lo = spu_shuffle(vect_b1, vect_b2, _merge2_vect_)
5       a_lo = spu_shuffle(vect_a1, vect_a2, _merge2_vect_)
6       p = spu_mul(a_hi , b_hi)
7       e = spu_msub(a_hi , b_hi , p)
8       t1 = spu_mul(a_hi , b_lo)
9       t2 = spu_mul(b_hi , a_lo , t1)
10      tmp = spu_add(e , t2)
11      (hi, lo) = Renormalise2-toward-zero (p, tmp)
12      vect_c1 = spu_shuffle(hi, lo, _merge1_vect_)
13      vect_c2 = spu_shuffle(hi, lo, _merge2_vect_)
14  return (vect_c1, vect_c2)
```

Il est aussi décrit dans le schéma de la figure 3.19.

Les deux instructions des lignes 8, 9 sont indépendantes des instructions des lignes 6,7. Donc elles sont cachées les unes par les autres. La dépendance entre les instructions de cet algorithme est décrite dans la figure 3.20.

En utilisant l'outil `spu_timing` d'IBM, le temps réel de calcul de cette fonction est de **60 cycles d'horloge**.

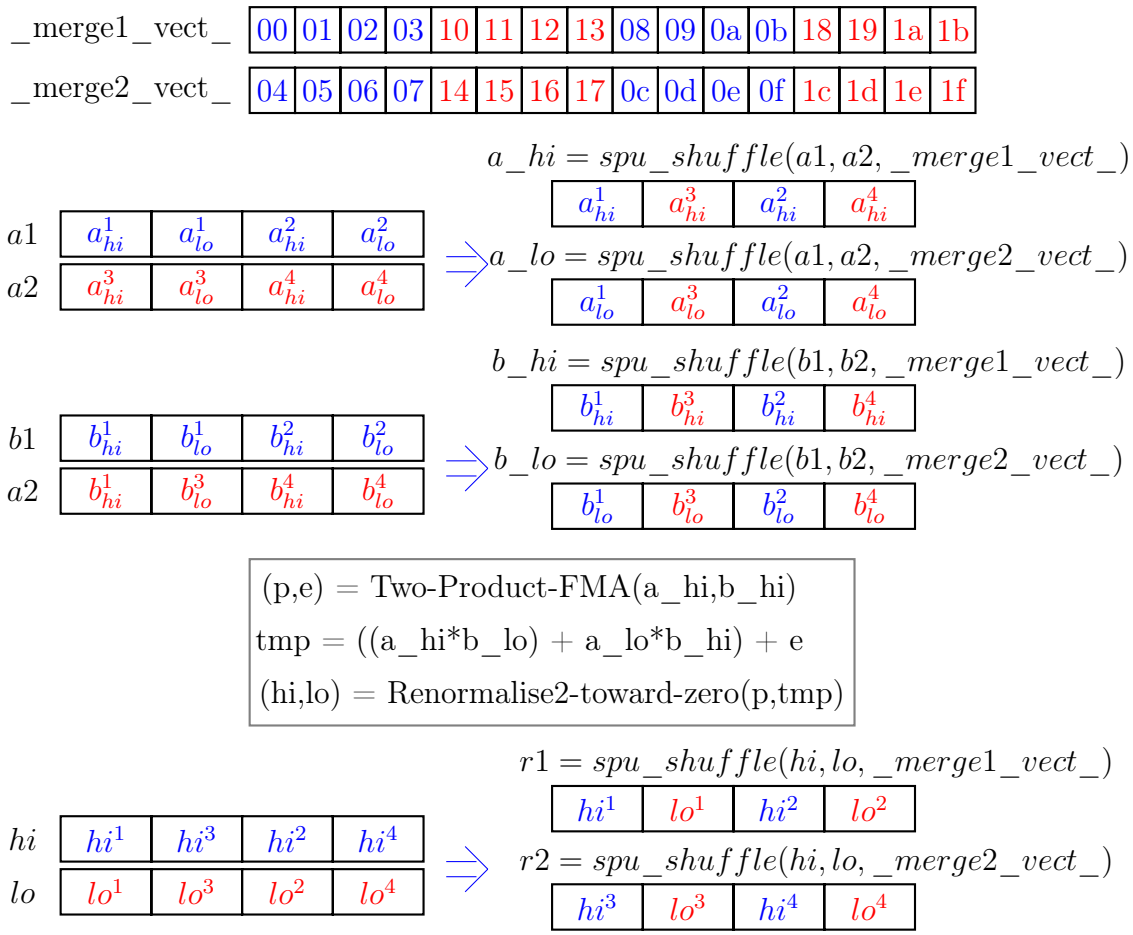

FIG. 3.19 – La multiplication de deux couples de vecteurs de double-simples

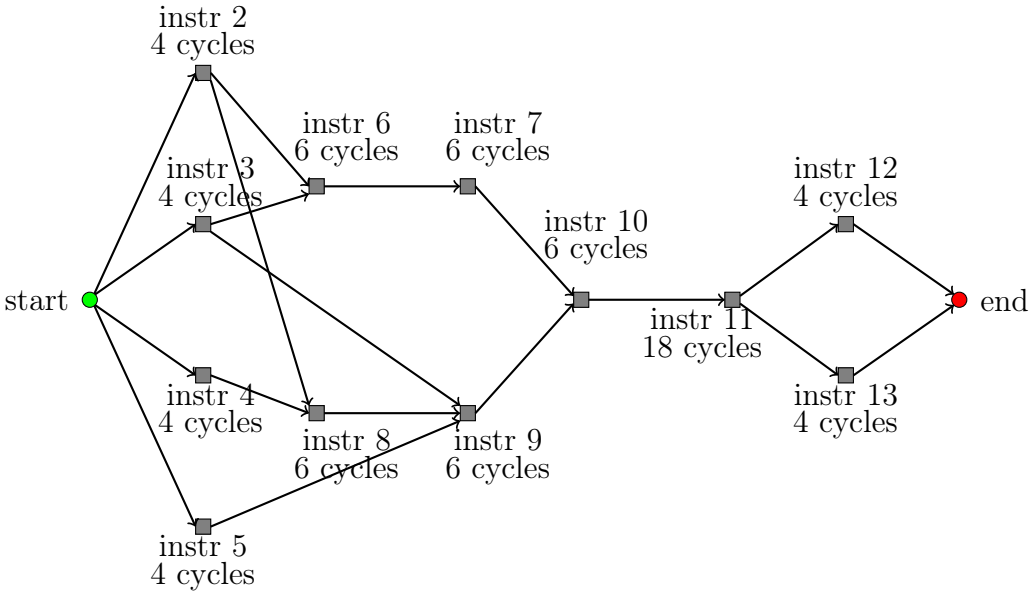

FIG. 3.20 – La dépendance entre les instructions de l'algorithme `mul_ds_ds_2vect`

3.3.7 La division

Comme expliqué dans la partie de "Opérations de base sur des double-simples", l'algorithme de la division `div_ds_ds` se base sur des opérations en simple précision, y compris le FMA et la division. Le processeur CELL ne possède pas d'opération vectorielle de division en matériel, il supporte seulement une fonction `spu_re` qui permet d'estimer l'inversion des éléments dans un vecteur de flottants de simple précision. Donc pour implémenter la division de double-simples, il faut d'abord implémenter la division en simple précision en utilisant la fonction `spu_re`, la sommation et le produit en simple précision.

La division en simple précision

La procédure pour calculer la division de deux nombres flottants en simple précision a et b est :

1. calculer l'inverse de b,
2. multiplier la valeur inverse de b avec a.

Le problème est que la fonction `spu_re` nous donne une estimation approximative de l'inversion des flottants précise à seulement 12 bits. En tirant parti de la fonction FMA et des opérations de base en simple précision qui sont respectivement la sommation et le produit, nous avons réussi à implémenter la fonction d'inversion en simple précision qui est appelée `inverse`. Cet algorithme se base sur la méthode itérative de Newton. Soit $invb[i]$ une approximation de la valeur inverse de b. Une nouvelle valeur qui est plus précise que $invb[i]$ peut être calculée par : $invb[i+1] = invb[i] + invb \times (1 - invb[i] \times b)$. A

cause de l'erreur informatique, dans la plupart de cas on ne peut pas atteindre la valeur exacte de l'opération. Donc ici, nous faisons seulement une fois d'itération.

```
1       inverse (b)
2          tmp0 = spu_re(b)
3          rerr = spu_nmsub(tmp , b , 1)
4          tmp1 = spu_madd(rerr , tmp0, tmp0)
5          rerr = spu_nmsub(tmp1 , b , 1)
6          eerr = spu_mul(rerr , tmp2)
7       return (tmp2, rerr, eerr)
```

Cet algorithme retourne trois valeurs $tmp2$, $rerr$ et $eerr$ qui sont respectivement le résultat, l'erreur relative et l'erreur de l'algorithme d'inversion. Ces trois valeurs vont être utilisées pour calculer la division en simple précision. La précision de cet algorithme est estimée par le théorème suivant.

Théorème 11. *Soit b un nombre flottant de simple précision, ε l'erreur relative en simple précision. L'erreur relative de l'algorithme* `inverse` *est bornée par :*

$$\frac{|inverse(b) - 1/b|}{|1/b|} < 2 \times \varepsilon + \mathcal{O}(\varepsilon^{3/2})$$

La démonstration de ce théorème est fournit dans l'Annexe A.

En utilisant l'algorithme `inverse`, la division en simple précision a/b peut être calculé simplement par $a \times inverse(b)$. Mais cela nous donne deux erreurs d'arrondi, une de l'inversion et une de la multiplication, qui rendent le résultat moins précis. En utilisant la méthode itérative de Newton avec $a \times inverse(b)$ pour la valeur initiale et une fois d'itération, nous pouvons récupérer un résultat plus précis.

```
1       div (a, b)
2         (invb, rerr, eerr) = inverse(b)
3         tmp = spu_mul(eerr , a)
4         q = spu_madd(a , invb , tmp)
5       return q
```

La précision de l'algorithme `div` est donné dans le théorème qui suit. Ce théorème est démontré dans l'Annexe A.

Théorème 12. *Soit a, b deux nombres flottants de simple précision, ε l'erreur relative en simple précision. L'erreur relative de l'algorithme* `div` *est bornée par :*

$$\frac{|div(a,b) - a/b|}{|a/b|} < \varepsilon + \mathcal{O}(\varepsilon^2)$$

La division de deux double-simples

En utilisant l'algorithme `div` décrit précédemment, l'algorithme `div_ds_ds` peut être implémenté simplement.

Dans l'algorithme `div_ds_ds`, sauf les deux instructions des lignes 3, 4 qui sont indépendantes, les autres instructions sont forcément couplées, il n'y a pas beaucoup de chose

pour faire de la parallélisation. Donc les deux versions de la division, `div_ds_ds_vect` pour un couple de vecteurs et `div_ds_ds_2vect` pour deux couples de vecteurs (comme avec la sommation et le produit) sont presque les mêmes. Cela signifie que la version `div_ds_ds_2vect` est deux fois plus performante que la version `div_ds_ds_vect`.

Comme avec la sommation et le produit, nous implémentons directement les algorithmes `inverse` et `div` pour éviter le temps d'invocation de fonction. Cela nous donne aussi un autre avantage. Comme toutes les deux divisions en simple précision de l'algorithme `div_ds_ds` sont par b_0, le résultat de l'algorithme `inverse` peut être utilisé pour toutes ces deux divisions.

En utilisant l'outil `spu_timing` d'IBM pour mesurer le temps, la fonction `div_ds_ds_2vect` coûtent **111 cycles d'horloge**.

3.3.8 Optimiser les algorithmes

Dans les versions 2 de la sommation et du produit des double-simple, quatre opérations sont effectuées en même temps et elles sont complètement parallélisées. Cela nous donne une grosse performance de calcul. Mais comme avec les EFTs, il nous reste encore plusieurs cycles d'horloge non utilisés. On peut encore améliorer les algorithmes précédents en augmentant le nombre d'opérations effectuées à la fois.

Avec la contrainte de la mémoire locale de chaque SPE, nous implémentons des fonctions pour effectuer 8 opérations de double-simple en même temps, qui est deux fois le nombre d'opérations effectuées dans la version 2. Pour cela, il nous suffit de doubler le bloc de code du `add_ds_ds_2vect`, du `mul_ds_ds_2vect` et du `div_ds_ds_2vect` qui nous donne les fonctions `add_ds_ds_4vect`, `mul_ds_ds_4vect` et `div_ds_ds_4vect`. Le travail d'optimisation (réorganiser les instructions) est à la charge du compilateur.

Les résultats que nous avons obenus sont très intéressants. Le `add_ds_ds_4vect` coûte **72 cycles d'horloge**, le `mul_ds_ds_4vect` coûte **63 cycles d'horloge** et le `div_ds_ds_4vect` coûte **125 cycles d'horloge**. Ces nombres nous signifient que le `add_ds_ds_4vect`, le `mul_ds_ds_4vect` et le `div_ds_ds_4vect` sont presque deux fois plus performants que le `add_ds_ds_2vect`, le `mul_ds_ds_2vect`, et le `div_ds_ds_2vect` respectivement.

3.4 Résultats théoriques

Supposons que ce bibliothèque soit testé sur un processeur CELL avec une fréquence de 3.2 GHz. Le tableau 3.1 résume les résultats théoriques pour tous les algorithmes implémentés dans notre bibliothèque. La crête théorique de performance en double précision qu'un SPE peut atteindre est : $2 \times 2 \times 3.2/7 = 1.8 GFLOPs$. Cela signifie que la performance théorique de la bibliothèque double-simple est beaucoup moins bonne que la performance théorique des opérations flottantes en double précision de la machine.

Fonction	**Nombre d'opérations**	**Temps de calcul**	**Performance**
Add_ds_ds_vect	2	50 cycles	0.128 GFLOPs
Add_ds_ds_2vect	4	64 cycles	0.2 GFLOPs
Add_ds_ds_4vect	8	72 cycles	0.355 GFLOPs
Mul_ds_ds_vect	2	49 cycles	0.130 GFLOPs
Mul_ds_ds_2vect	4	60 cycles	0.213 GFLOPs
Mul_ds_ds_4vect	8	63 cycles	0.406 GFLOPs
Div_ds_ds_2vect	4	111 cycles	0.115 GFLOPs
Div_ds_ds_4vect	8	125 cycles	0.2048 GFLOPs

TAB. 3.1 – Résultats théoriques de la bibliothèque double-simple

Chapitre 4

La bibliothèque Quad-simple

Avec la même méthodologie que la bibliothèque double-simple, on peut augmenter la précision en quad-simple [5], soit représenter un nombre flottant de précision étendue par quatre flottants de simple précision.

Ce chapitre va présenter la méthodologie et l'implémentation de la bibliothèque quad-simple. Avec quatre flottants de simple précision, toutes les opérations sur des quad-simples vont être beaucoup plus compliquées que sur des double-simples. Donc, nous n'allons pas aller trop en détail comme avec la présentation de la bibliothèque double-simple, en particulier l'implémentation de la bibliothèque.

4.1 Présentation

Un quad-simple est une somme non-évaluée de quatre nombres flottants IEEE simple précision $a = a_0 + a_1 + a_2 + a_3$, avec la contrainte suivante entre ces quatre flottants :

$$|a_{i+1}| \leq \frac{1}{2} ulp(a_i) \text{ pour } i \in \{0, 1, 2\}$$

Cette contrainte assure l'unicité de représentation pour un quad-simple.

4.2 Opérations de base

4.2.1 Renormalisation

Toutes les opérations sur les quad-simples décrites dans ce chapitre produisent un résultat de quatre flottants simple précision et un terme de correction en simple précision. Il reste encore des chevauchements entre eux. Comme pour le double-simple, il faut avoir une étape de renormalisation pour transformer ce résultat intermédiaire dans un quad-simple normalisé.

L'algorithme de renormalisation proposé par Yozo Hida [5, p. 7] nous permet de renormaliser un résultat étendu de cinq flottants dans un quad-simple. Mais cet algorithme n'est défini que pour le mode d'arrondi au plus près. De plus, il utilise une boucle qui contient un débranchement. Cette boucle nous empêche de faire de la parallélisation au niveau d'instruction. Donc nous avons réécrit cet algorithme en utilisant le

`Renormalise2-nearest` qui est une simulation de la transformation exacte sous le mode d'arrondi au plus près et en déroulant la boucle.

```
1   Renormalise5 (a0, a1, a2, a3, a4)
2       (s,t_4) = Renormalise2-nearest(a3, a4)
3       (s,t_3) = Renormalise2-nearest(a2, s)
4       (s,t_2) = Renormalise2-nearest(a1, s)
5       (t_0,t_1) = Renormalise2-nearest(a0, s)
6
7       s = t_0, e = t_1
8       if (e = 0)
9          b0 = 0
10      else
11         b0 = s, s = e
12
13      (s,e) = Renormalise2-nearest(s, t_2)
14      if (e = 0)
15         b1 = 0
16      else
17         b1 = s, s = e
18
19      (s,e) = Renormalise2-nearest(s, t_3)
20      if (e = 0)
21         b2 = 0
22      else
23         b2 = s, s = e
24
25      (b3,b4) = Renormalise2-nearest(s, t_4)
26
27      if (b2 = 0)
28         b2 = b3, b3 = b4, b4 = 0
29
30      if(b1 = 0)
31         b1 = b2, b2 = b3, b3 = b4, b4 = 0
32
33      if (b0 = 0)
34         b0 = b1, b1 = b2, b2 = b3, b3 = b4
35
36  return (b0, b1, b2, b3)
```

Dans ce cas cet algorithme fonctionne bien, les cinq éléments d'entrée a_0, a_1, a_2, a_3, a_4 sont transformées en cinq flottants b_0, b_1, b_2, b_3, b_4 qui satisfont :

$$|b_4| \leq \frac{1}{2}ulp(b_3), |b_3| \leq \frac{1}{2}ulp(b_2), |b_2| \leq \frac{1}{2}ulp(b_1), |b_1| \leq \frac{1}{2}ulp(b_0)$$

$$\text{d'où } |b_4| \leq \frac{1}{2}\varepsilon|b_3|, |b_3| \leq \frac{1}{2}\varepsilon|b_2|, |b_2| \leq \frac{1}{2}\varepsilon|b_1|, |b_1| \leq \frac{1}{2}\varepsilon|b_0|$$

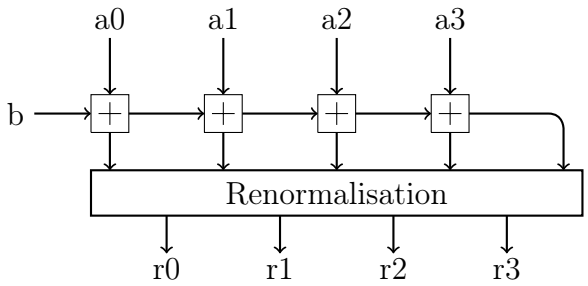

FIG. 4.1 – La sommation d'un quad-simple avec un simple

$$\text{donc } |b_4| \leq \frac{1}{16}\varepsilon^4|b_0|$$

Cela veut dire que l'erreur causée par la renormalisation satisfait la borne

$$|Renormalise5(a_0, a_1, a_2, a_3, a_4) - (a_0 + a_1 + a_2 + a_3 + a_4)| \leq \frac{1}{16}\varepsilon^4|a_0 + a_1 + a_2 + a_3 + a_4|$$

Comme avec l'algorithme de renormalisation de Yozo Hida, les conditions nécessaires pour que cet algorithme fonctionne bien ne sont pas connues [5, p. 7], mais il est applicable pour les algorithmes qui suivent.

4.2.2 La sommation

Quad-simple + simple

La sommation d'un quad-simple a avec un simple b suit le schéma de la figure 4.1.

En utilisant quatre transformations exactes sur la sommation, les cinq flottants du résultat intermédiaire avant la renormalisation représentent exactement la somme de a et b. Ces cinq flottants peuvent être considérés comme un résultat intermédiaire de quatre flottants et un terme de correction. Ils sont ensuite renormalisés pour produire le quad-simple du résultat.

Il est évident que la précision de cet algorithme dépend totalement du processus de renormalisation. Si la renormalisation fonctionne bien, cette sommation est précise à 96 bits.

Quad-simple + Quad-simple

Le schéma de la figure 4.3 décrit le processus de calcul de la somme de deux quad-simples. Il ressemble au schéma de calcul de la somme de deux double-simples. L'erreur d'une transformation exacte sur la sommation est considérée comme un terme de l'ordre plus bas. Comme la complexité de ce schéma est très grande, on risque d'ignorer plusieurs bits les moins significatifs si on garde seulement quatre termes du résultat intermédiaire. Donc il est absoluement nécessaire de calculer le terme de correction, qui est la cinquième

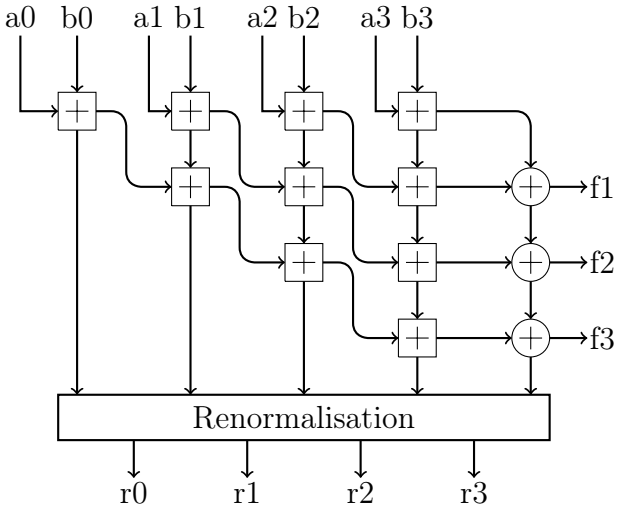

FIG. 4.2 – La sommation de deux quad-simples

composante du résultat intermédiaire. Ces cinq composantes sont ensuite normalisées pour récupérer un quad-simple du résultat.

Ce schéma de calcul nous donne aussi la possibilité de faire de la parallélisation au niveau des instructions. Toutes les transformations exactes dans une même ligne sont totalement indépendantes et donc peuvent être parallélisées.

Comme avec la sommation de deux double-simples, cet algorithme ne permet pas d'avoir un résultat qui satisfait la borne d'erreur suivante :

$$fl(a+b) = (1+\delta)(a+b) \text{ avec } |\delta| \text{ un terme de } \mathcal{O}(\varepsilon^4).$$

Au lieu de cela, il n'est possible que de donner une borne d'erreur effective de cet algorithme. Nous proposons le théorème suivant :

THÉORÈME 13. *Soient $a = (a_0, a_1, a_2, a_3)$ et $b = (b_0, b_1, b_2, b_3)$ deux quad-simples. L'erreur de la sommation de ces deux quad-simples donnée par l'algorithme* `add_qs_qs` *satisfait :*

$$|add\_qs\_qs(a,b) - (a+b)| < max(\frac{181}{8}\varepsilon^5|a_0+b_0|, \frac{31}{4}\varepsilon^4|a_1+b_1|, 4\varepsilon^3|a_2+b_2|) + \frac{1}{16}\varepsilon^4|a+b|$$

Le terme $\frac{1}{16}\varepsilon^4|a+b|$ correspond à l'erreur de la renormalisation (considérée comme correct), et le premier terme correspond à l'erreur de calcul du résultat intermédiaire avant la phase de renormalisation.

Le pire cas est obtenu quand $a_0 = -b_0$ et $a_1 = -b_1$. Dans ce cas, l'erreur effective est bornée par $4\varepsilon^3|a_2+b_2|$. Comme le terme le plus haut du résultat est a_2+b_2, l'erreur relative peut être bornée par $4\varepsilon^3$. Cela signifie qu'on a perdu la précision du dernier terme entier du quad-simple du résultat.

Dans le meilleur cas où a_0 et b_0 ont le même signe ou $min(|a_0|, |b_0|) < \frac{1}{2}max(|a_0|, |b_0|)$, l'erreur effective est bornée par

$$\frac{181}{8}\varepsilon^5|a_0 + b_0| + \frac{1}{16}\varepsilon^4|a + b|$$

et donc l'erreur relative de cet algorithme peut être bornée par

$$\frac{181}{8}\varepsilon^5 + \frac{1}{16}\varepsilon^4.$$

.

Cela veut dire que le résultat acquis est précis à 96 bits.

La démonstration du théorème 13 se trouve dans l'Annexe A.

Il y a un algorithme proposé par J. Shewchuk et S. Boldo cité dans [5] qui permet de récupérer les quatre premières composantes du résultat exact satisfaisant la borne d'erreur :

$$fl(a + b) = (1 + \delta)(a + b) \text{ avec } |\delta| \text{ un terme en } \mathcal{O}(\varepsilon^4)$$

La précision de cet algorithme enduit une augmentation au temps de calcul (fonctionne beaucoup plus lentement, 2-3.5 fois moins vite). Donc, nous avons décidé de ne pas implémenter l'algorithme de J. Shewchuk et S. Boddo.

4.2.3 Le produit

Le produit de deux quad-simples a et b est le produit de deux sommes qui représentent respectivement a et b. Chaque somme se compose de quatre éléments, donc le produit se compose de 16 composantes au total.

Avec la normalisation imposée sur ces deux quad-simples on a

$$|a_3| \leq \frac{1}{2}ulp(a_2), |a_2| \leq \frac{1}{2}ulp(a_1), |a_1| \leq \frac{1}{2}ulp(a_0)$$

$$|b_3| \leq \frac{1}{2}ulp(b_2), |b_2| \leq \frac{1}{2}ulp(b_1), |b_1| \leq \frac{1}{2}ulp(b_0)$$

d'où

$$|a_3| \leq \frac{1}{2}\varepsilon|a_2|, |a_2| \leq \frac{1}{2}\varepsilon|a_1|, |a_1| \leq \frac{1}{2}\varepsilon|a_0|$$

$$|b_3| \leq \frac{1}{2}\varepsilon|b_2|, |b_2| \leq \frac{1}{2}\varepsilon|b_1|, |b_1| \leq \frac{1}{2}\varepsilon|b_0|$$

Si on considère $a_0 \times b_0$ (le terme principal soit l'approximation du produit) comme un terme d'ordre $\mathcal{O}(1)$, les autres composantes du produit peuvent être estimées par des termes d'ordre allant de $\mathcal{O}(\varepsilon)$ à $\mathcal{O}(\varepsilon^6)$. Parce qu'une erreur d'arrondi de terme $\mathcal{O}(\varepsilon^4)$ (causé par la renormalisation) est inévitable, les composantes des termes de moins de

$\mathcal{O}(\varepsilon^4)$ ne sont pas mises en compte pour diminuer la complexité de l'algorithme.

$$\begin{aligned} a \times b \;\approx\; & \underbrace{a_0 \times b_0}_{\mathcal{O}(1)} + \underbrace{a_0 \times b_1 + a_1 \times b_0}_{\mathcal{O}(\varepsilon)} + \\ & \underbrace{a_0 \times b_2 + a_1 \times b_1 + a_2 \times b_0}_{\mathcal{O}(\varepsilon^2)} + \\ & \underbrace{a_0 \times b_3 + a_1 \times b_2 + a_2 \times b_1 + a_3 \times b_0}_{\mathcal{O}(\varepsilon^3)} + \\ & \underbrace{a_1 \times b_3 + a_2 \times b_2 + a_3 \times b_1}_{\mathcal{O}(\varepsilon^4)} \end{aligned}$$

Les composantes du produit n'étant pas représentables par des flottants, il faut d'abord utiliser des EFTs sur le produit pour décomposer chacune de ces composantes en deux flottants :

$$(p_{ij}, e_{ij}) = Two - Product(a_i, b_j)$$

avec p_{ij} un terme en $\mathcal{O}(\varepsilon^{i+j})$ et e_{ij} un terme en $\mathcal{O}(\varepsilon^{i+j+1})$.
Le produit peut s'écrire de la manière suivante :

$$\begin{aligned} a \times b \;\approx\; & \underbrace{p_{00}}_{\mathcal{O}(1)} + \underbrace{e_{00} + p_{01} + p_{10}}_{\mathcal{O}(\varepsilon)} + \\ & \underbrace{e_{01} + e_{10} + p_{02} + p_{11} + p_{20}}_{\mathcal{O}(\varepsilon^2)} + \\ & \underbrace{e_{02} + e_{11} + e_{20} + p_{30} + p_{12} + p_{21} + p_{30}}_{\mathcal{O}(\varepsilon^3)} + \\ & \underbrace{e_{03} + e_{12} + e_{21} + e_{30} + p_{13} + p_{22} + p_{31}}_{\mathcal{O}(\varepsilon^4)}. \end{aligned}$$

Donc, il y a 1 terme en $\mathcal{O}(1)$, 3 termes en $\mathcal{O}(\varepsilon)$, 5 termes en $\mathcal{O}(\varepsilon^2)$, 7 termes en $\mathcal{O}(\varepsilon^3)$ et 7 termes en $\mathcal{O}(\varepsilon^4)$. On doit accumuler ces termes pour récupérer au final 5 termes de $\mathcal{O}(1)$, $\mathcal{O}(\varepsilon)$, $\mathcal{O}(\varepsilon^2)$, $\mathcal{O}(\varepsilon^3)$ et $\mathcal{O}(\varepsilon^4)$ respectivement qui forment les entrées pour la renormalisation. Pour les composantes de terme $\mathcal{O}(\varepsilon^4)$ il nous suffit d'utiliser la sommation normale parce que les erreurs engendrées par ces opérations sont des termes en $\mathcal{O}(\varepsilon^5)$ qui ne sont pas intéressants. Pour les autres composantes de terme plus haut il faut utiliser l'EFT sur la sommation pour conserver les erreurs engendrées.

L'EFT sur la sommation transforme deux composantes de terme $\mathcal{O}(\varepsilon^k)$ dans deux composantes une de terme $\mathcal{O}(\varepsilon^k)$ et une de terme $\mathcal{O}(\varepsilon^{k+1})$. Plus généralement, en utilisant m fois l'EFT sur la sommation on peut transformer $m+1$ composantes de terme $\mathcal{O}(\varepsilon^k)$ dans une composante de terme $\mathcal{O}(\varepsilon^k)$ et m composantes de terme $\mathcal{O}(\varepsilon^{k+1})$. Et donc avec le produit de deux quad-simples, pour arriver à cinq composantes au final il faut utiliser 20 fois l'EFT sur la sommation et 18 sommations normales. L'algorithme du produit est trop compliqué pour être décrit en détail. Pour simplifier, nous utilisons des blocs de calcul qui se composent de plusieurs EFTs sur la sommation.

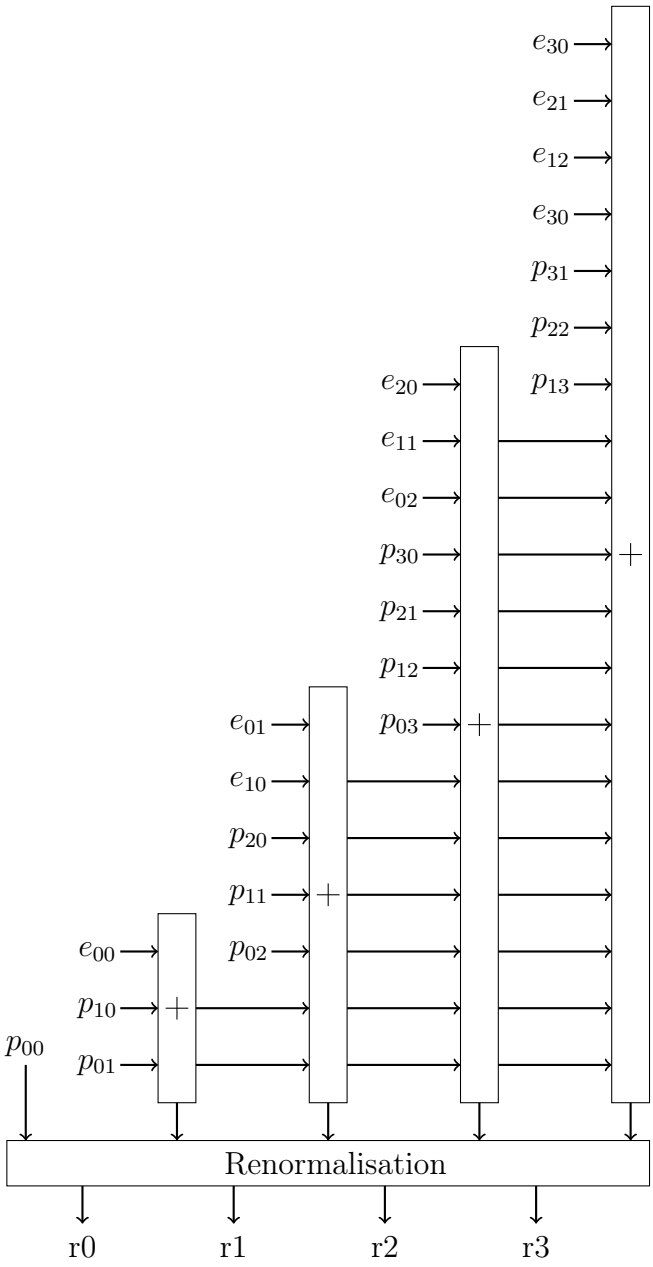

FIG. 4.3 – La multiplication de deux quad-simples

Lemme 14. *Un bloc de m EFTs sur la sommation peut être considéré comme une transformation exacte sur la sommation de $m+1$ entrées $a_0, a_1, \ldots, a_m$ qui retourne une estimation de la somme totale p avec m erreurs e_i pour $i \in 1, 2, \ldots, m$ qui satisfont :*

$$\begin{aligned} |p| &\leq \sum_{k=0}^{m} |a_i| \\ |e_i| &\leq \varepsilon \sum_{k=0}^{m} |a_k| \text{ pour } i \in 1, 2, \ldots, m \\ \text{d'où } \sum_{i=1}^{m} |e_i| &\leq m\varepsilon \sum_{k=0}^{m} |a_k|. \end{aligned}$$

Le lemme 14 nous permet d'avoir une borne d'erreur relative de notre algorithme de calcul du produit de deux quad-simples.

Théorème 15. *Soient a et b deux quad-simples. Le résultat retourné par l'algorithme* `mul_qs_qs` *sur ces deux quad-simples satisfait la borne d'erreur :*

$$|mul\_qs\_qs(a,b) - a \times b| < 12500\varepsilon^5|a \times b| + \frac{1}{16}\varepsilon^4|a \times b|.$$

La première composante du côté droit est beaucoup plus petite que la deuxième composante qui est l'erreur de la renormalisation. Donc dans le cas où la renormalisation fonctionne bien, l'algorithme `mul_qs` nous permet de récupérer un résultat précis jusqu'à 96 bits.

4.2.4 La division

L'algorithme de la division de deux quad-simples est basé sur le meme algorithme que la double-simple. Mais pour économiser du temps de calcul, au lieu de calculer les éléments de la division un par un, nous tirons parti de la division de double-simples. Un quad-simple est représenté par quatre flottants donc on peut considérer qu'il peut être représenté par deux double-simples. Ce point de vue nous conduit à un algorithme qui ressemble à la division de deux double-simples. Pour augmenter la précision de cet algorithme, une cinquième composante de correction va être calculée pour compenser le résultat.

L'algorithme de la division de $a = (a_0, a_1, a_2, a_3)$ par $b = (b_0, b_1, b_2, b_3)$ est appelé `div_qs_qs` et est décrit de la manière suivante :

```
1  div_qs_qs (a,b)
2    (q0,q1) = div_ds_ds((a0,a1) , (b0,b1))
3    (s0,s1,s2,s3) = sub_qs_qs(a , mul_qs_qs((q0,q1,0,0) , b))
4    (q2,q3) = div_ds_ds((s0,s1) , (b0,b1))
5    (t0,t1,t2,t3) = sub_qs_qs(a , mul_qs_qs((q0,q1,q2,q3) , b))
6    q4 = fl(t0 / b0)
7    (q0,q1,q2,q3) = renormalisation(q0,q1,q2,q3,q4)
8  return (q0,q1,q2,q3)
```

Supposons que nous disposons d'une opération de division en simple précision avec la borne d'erreur relative de ε, d'après le chapitre **bibliothèque double-simple**, l'algorithme `div_ds_ds` a une borne d'erreur relative de $15.5 \times \varepsilon^2 + \mathcal{O}(\varepsilon^3)$. Nous proposons le théorème suivant qui donne une borne de l'erreur relative de l'algorithme `div_qs_qs`.

Théorème 16. *Soient a et b deux quad-simples. Le résultat retourné par l'algorithme* `mul_qs` *sur ces deux quad-simples satisfait la borne d'erreur :*

$$|div\_qs\_qs(a,b) - a/b| < 14000\varepsilon^5|a/b| + \frac{1}{8}\varepsilon^4|a/b|$$

Ce théorème signifie que l'algorithme `div_qs_qs` donne un résultat précis jusqu'à 94 bits. Nous n'avons pas eu le temps d'insérer la démonstration dans le manuscript.

4.3 Implémentation

Comme toutes les opérations sur des quad-simples sont vraiment compliquées, avec la contrainte de la mémoire locale des programmes SPEs, on est forcé d'utiliser des appels de fonctions au lieu de les implémenter directement. Ces appels de fonctions nécessitent le changement de contexte et donc l'augmentation du temps d'exécution du programme. De plus, les appels de fonctions diminuent les possibilités de faire de la parallélisation au niveau des instructions.

Dans l'implémentation de la bibliothèque quad-simple nous allons utiliser la fonction `Two-Sum-toward-zero-2` de la bibliothèque double-simple qui tire parti des pipelines pour calculer la transformation exacte sur deux couples de vecteurs de simples presque en parallèle.

Nous implémentons d'abord la version dépaquetée des algorithmes présentés précédemment. Ces fonctions ne traitent pas directement des quad-simples mais elles opèrent sur des vecteurs de flottants en simple précision qui contiennent chacun des composantes de quatre quad-simples en même temps. Ces vecteurs peuvent être considérés comme des quad-simples dépaquetés.

En utilisant des fonctions dépaquetées, les algorithmes d'opérations sur des quad-simples seront implémentés comme suit : dépaqueter le(s) quad-simple(s) d'entrées dans quatre vecteurs de flottants, puis appliquer la fonction correspondante sur des vecteurs dépaquetés, et finalement paqueter des vecteurs du résultat dans un (des) quad-simple(s) du résultat.

Chaque nombre de couples de quad-simples d'entrée nous donne une version des algorithmes d'opérations de base sur des quad-simples. Le meilleur cas est d'avoir 4 couples de quad-simples d'entrée, ils seront dépaquetés complètement dans 4 couples de vecteurs de flottants en simple précision. La fonction dépaquetée correspondante nous donne 4 vecteurs de flottants du résultat. Ils seront paquetés pour récupérer 4 quad-simples du résultat.

Dépaqueter et paqueter des quad-simples

Le processus pour dépaqueter quatre quad-simples dans quatre vecteurs de simples est décrit dans le schéma de la figure 4.4. Pour repaqueter des vecteurs de simples dans des quad-simples il faut proceder à l'inverse.

Chaque appel de la fonction `spu_shuffle` coûte 4 cycles d'horloge. Avec la dépendance entre les instructions, le processus pour dépaqueter ou de paqueter des quad-simples coûte au total 14 cycles d'horloge.

4.4 Résultats théoriques

En utilisant l'outil `spu_timing` d'IBM avec la façon simplifiée de calculer le temps de calcul d'une fonction :

Temps de calcul d'une fonction = temps de calcul de cette fonction retourné par `spu_timing` + le temps de calcul de toutes les fonctions appelées dans cette fonction,

on peut mesurer le temps de calcul des fonctions implémentées dans la bibliothèque quad-simple.

Les résultats obtenus avec la bibliothèque quad-simple sont résumés dans le tableau 4.1. La performance est mesurée sur un SPE avec une fréquence de 3.2 GHz.

Fonction	**Nombre d'opérations**	**Temps de calcul**	**Performance**
Add_qs_qs_4vect	4	449 cycles	28.5 MFLOPs
Mul_qs_qs_4vect	4	583 cycles	21.9 MFLOPs
Div_qs_qs_4vect	4	2667 cycles	4.79 MFLOPs

TAB. 4.1 – Résultats théoriques de la bibliothèque quad-simple

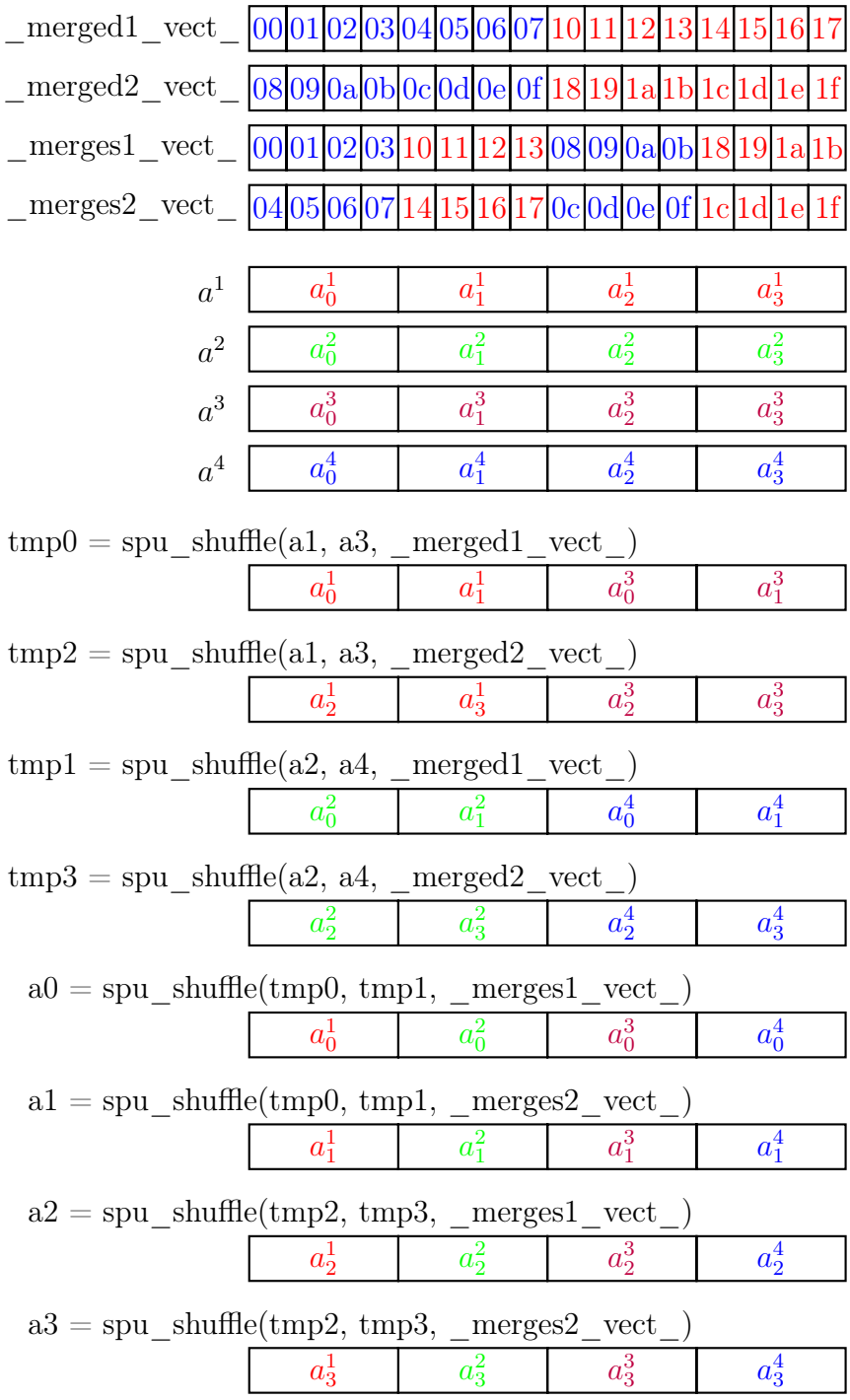

FIG. 4.4 – Algorithme pour dépaqueter 4 quad-simples dans 4 vecteurs de simples

Chapitre 5

Résultats et simulations numériques

5.1 La libraire double-simple

5.1.1 Résultats expérimentaux

Pour comparer la performance de la bibliothèque quasi-double et le vrai double de la machine, on réalise un grand nombre de calculs sur les deux types et on compare le temps de calcul entre eux.

Le programme de test implémenté est chargé d'effectuer une opération (soit la sommation, soit le produit) sur des éléments de deux grand vecteurs de double-simples avec un nombre donné de répétition. Ce programme utilise des fonctions `add_ds_ds_4vect`, `mul_ds_ds_4vect`, et `div_ds_ds_4vect` qui nous permettent d'atteindre la performance de crête de la bibliothèque double-simple. Pour chaque opération il faut avoir deux entrées et une sortie. Cela veut dire que pour effectuer une opération, il faut effectuer deux opérations de récupération des entrées à partir de la mémoire principale, et une opération de stockage du résultat vers la mémoire principale. Pour cacher le temps de communication entre le LS et la mémoire principale on utilise le `double-buffering`.

Le programme de test est lancé sur un vrai processeur CELL situé au CINES à Montpellier. Il dispose de 8 SPEs, chacun a une fréquence de 3.2GHz.

En lançant le programme de test plusieurs fois avec des tailles différentes des vecteurs et des tampons, avec des nombres différents d'SPEs participant(1, 2, 4 et 8 SPEs), nous récupérons les résultats dans l'annexe B.

Si on fait 100 fois la sommation de deux vecteurs de $2^{23} = 8,388,608$ éléments, la taille du message de transfert est de 16 KB, les résultats sont résumés dans la figure 5.1. Le tableau 5.1 nous donne la comparaison entre la performance théorique et la performance expérimentale des algorithmes.

Fonctions	**Performance théorique**	**Performance expérimentale**
Add_ds_ds_4vect	355 MFLOPs	250.4 MFLOPs
Mul_ds_ds_4vect	406 MFLOPs	287.2 MFLOPs
Div_ds_ds_4vect	204 MFLOPs	166.4 MFLOPs

TAB. 5.1 – Performance des fonctions de la bibliothèques double-simple

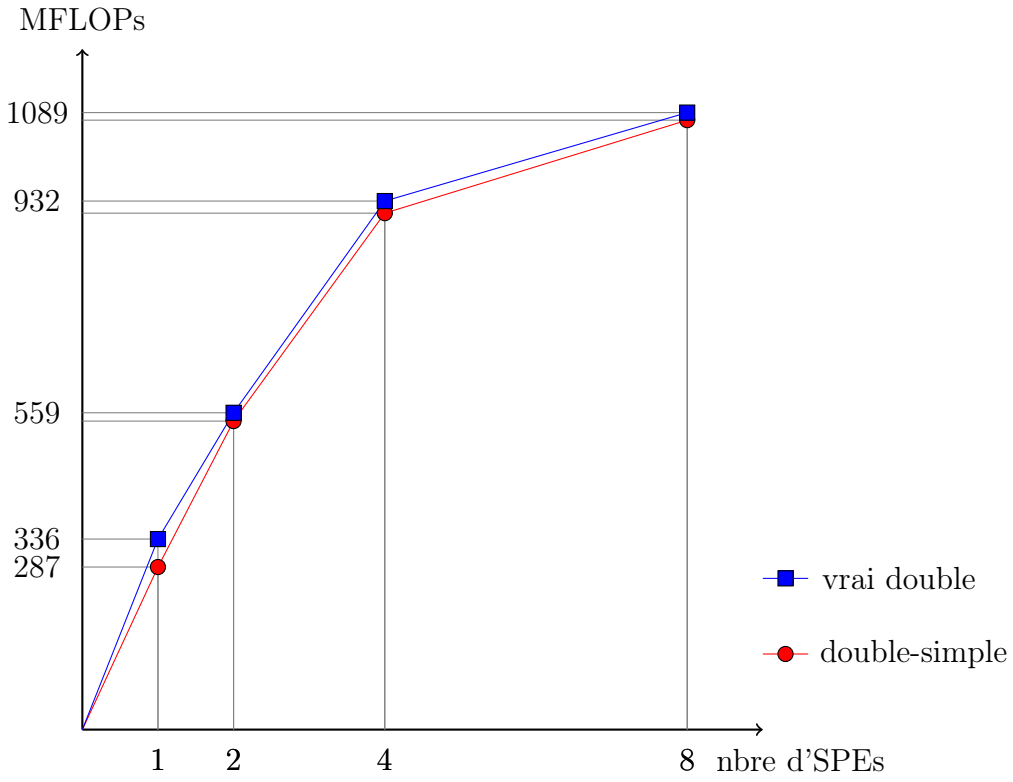

FIG. 5.1 – La performance de la bibliothèque double-simple : la sommation

5.1.2 Analyse des résultats

D'un point de vue performance la bibliothèque double-simple est dans la pratique bien moins performante. Pendant que la performance théorique de la sommation de la bibliothèque est 350 MFLOPs pour chaque SPE, la performance réelle de chaque SPE n'atteint qu'enivrons 287 MFLOPs soit 90/8 cycles pour chaque addition de double-simples.

D'après la partie **La bibliothèque double-simple** précédente, chaque sommation de deux double-simples coûte 72/8 cycles (avec l'algorithme `add_ds_ds_4vect`) et est implémentée sous la forme d'une fonction. Chaque appel à cette fonction coûte encore du temps d'invocation, par exemple le temps du chargement de paramètres. Chaque entrée coûte 6 cycles de chargement, la fonction `add_ds_ds_2vect` a huit entrées donc le temps du chargement d'entrées est 14 cycles.

Et puis, l'addition de deux grands vecteurs est réalisée par une boucle `for` qui coûte encore des cycles supplémentaires. Cela explique pourquoi on ne peut pas atteindre la performance de crête théorique de la libraire.

Quant au vrai-double la performance réelle est beaucoup plus petite que la crête de performance théorique. D'ailleurs, la performance de chaque SPE diminue quand le nombre de SPEs participant augmente. Dans le pire cas avec 8 SPEs participants, la performance avec des vrais doubles est presque égale à la performance de la bibliothèque double-simple.

Comme le programme de test effectue une seule opération sur de nombreuses données, le temps de calcul est plus petit que le temps de transfert. Le système de transfert du CELL ne peut pas fournir assez de données pour le calcul. C'est le cas de communication intensive, un nombre négligeable d'opérations sur un nombre important de données qui nécessitent trop de communications avec la mémoire principale. Dans ce cas on ne peut pas tirer parti de la puissance des SPEs.

Et puis, quand le nombre de SPEs participants augmente, le système de transfert atteint la saturation, c'est le cas où la vitesse du transfert ne change pas quand le nombre d'SPEs augmente. Donc la performance totale du programme ne change pas, ce qui tend à diminuer les performances de chaque SPE.

Pour mieux tester la performance de la bibliothèque double-simple, nous avons créé un autre programme qui effectue sur des fixées données d'entrée pour éviter le temps de transportations. Plus exactement, les données sont générées dans les mémoires locales des SPEs, qui sont les vecteurs de petite taille pour ne pas excéder la taille de LS. Chaque SPE va effectuer un grand nombre de fois des opérations sur ces données. Il n'y a pas de transportation entre le PPE et les SPEs. Donc le temps d'exécution du programme est exactement le temps de calcul sur des SPEs. Le tableau suivant résume les performances réelles obtenues par ce programme :

Les résultats dans le tableau 5.2 signifient que nous avons atteint la performance de crête avec le vrai double. Comme nous utilisons des données fixées, pour chaque opération, qui sont effectuée plusieurs fois, il ne faut charger qu'une seule fois les données d'entrée. Pour les prochaines fois les données sont déjà disponibles. Cela nous permet d'atteindre la performance de crête du vrai double. C'est pas le cas avec le double-simple. Chaque opération de double-simple est implémenté par une fonction. Pour effectuer des opérations de double-simple il faut utiliser des appels de fonction qui nécessitent à chaque fois le

Fonctions	Performance théorique (1SPE)	Performance expérimentale (1 SPE)	Performance expérimentale (8 SPEs)
Add_ds_ds_4vect	355 MFLOPs	266 MFLOPs	2133 MFLOPs
Mul_ds_ds_4vect	406 MFLOPs	320 MFLOPs	2560 MFLOPs
Div_ds_ds_4vect	204 MFLOPs	172 MFLOPs	1383 MFLOPs
addition en double précision	914 MFLOPs	914 MFLOPs	7314 MFLOPs
produit en double précision	914 MFLOPs	914 MFLOPs	7314 MFLOPs
division en double précision	(non supporté)	86 MFLOPs	691 MFLOPs

TAB. 5.2 – Performance des fonctions de la bibliothèques double-simple de du vrai double, sans transfert de données

chargement des données et le stockage de résultat. Donc on ne peut pas atteindre la performance de crête théorique de la bibliothèque double-simple.

5.1.3 Exactitude

Nous avons testé l'exactitude de la bibliothèque double-simple de la manière suivante :

1. créer par hasard deux vecteurs de 2^{24} double-simples,
2. effectuer des opérations sur ces deux vecteurs de 2^{24} double-simples,
3. calculer les valeurs estimées en double précision de ces deux vecteurs double-simple,
4. effectuer les même opérations sur deux vecteurs double précision,
5. calculer la différence entre le vecteur double-simple du résultat et le vecteur double du résultat.

Les résultats sont résumé dans le tableau 5.3.

Opération	Différence Maximale	Différence moyenne
Addition	0.0e+00	0.00e+00
Multiplication	2.964e-14	1.425e-16
Division	2.373e-14	1.758e-15

TAB. 5.3 – L'exactitude de la bibliothèque double-simple

5.2 La bibliothèque quad-simple

5.2.1 Résultats expérimentaux

Pour tester la bibliothèque quad-simple, nous implémentons la bibliothèque quad-simple en même temps que la bibliothèque double-double. La bibliothèque double-double nous donne aussi une précision étendue de quad-simple, plus exactement une précision de $53 \times 2 = 106$ bits. La bibliothèque double-double utilise la même méthodologie que la bibliothèque double-simple sauf qu'elle se base sur le mode d'arrondi au plus près. Le mode d'arrondi au plus près simplifie beaucoup le processus d'implémentation. Par

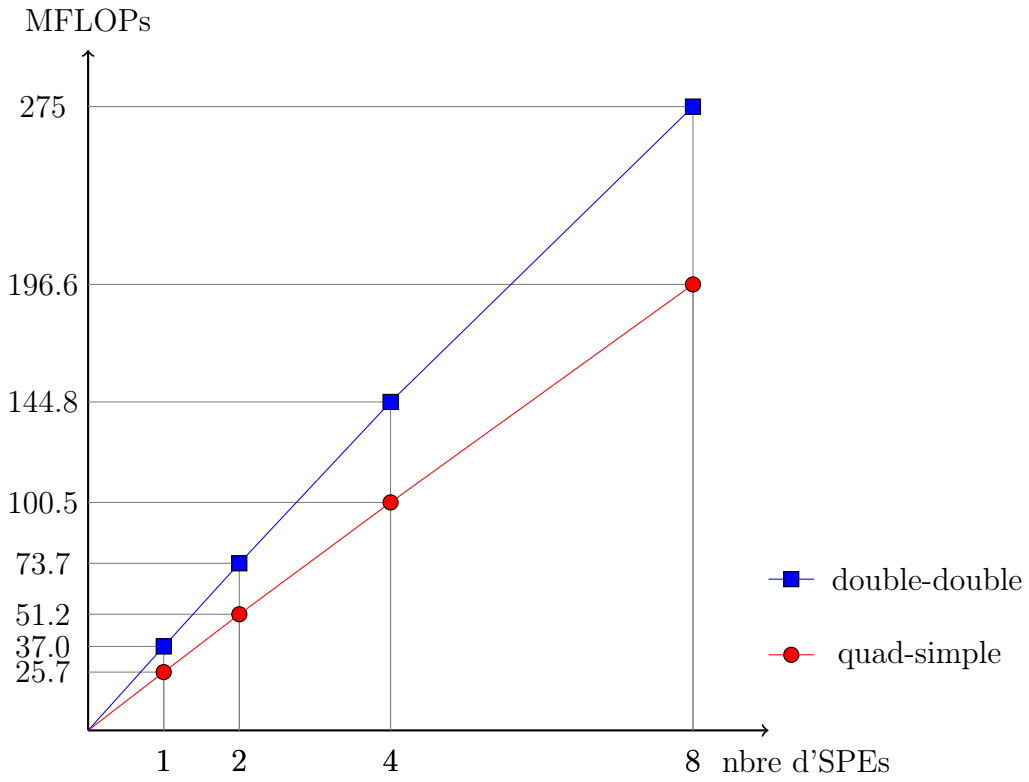

FIG. 5.2 – La performance de la bibliothèque quad-simple : la sommation

contre par manque de temps nous n'avons pas fait pas beaucoup d'optimisation sur la bibliothèque double-double.

En appliquant aussi des opérations sur deux grand vecteurs de quad-simples on peut mesurer le temps réel de calcul de la bibliothèque quad-simple. Ces résultats sont présentés dans la partie Annexe B.

Par exemple, si on fait 100 fois la sommation de deux vecteurs de $2^{23} = 8,388,608$ quad-simples, les résultats sont résumés dans la figure 5.2.

Fonctions	**Performance théorique**	**Performance expérimentale**
Add_qs_qs_4vect	28.5 MFLOPs	25.77 MFLOPs
Mul_qs_qs_4vect	21.9 MFLOPs	20.20 MFLOPs
Div_qs_qs_4vect	4.79 MFLOPs	4.61 MFLOPs

TAB. 5.4 – Performance des fonctions de la bibliothèque quad-simple

Il est clair que la bibliothèque quad-simple fonctionne beaucoup plus lentement que la bibliothèque double-simple parce qu'elle est beaucoup plus complexe. Mais elle n'est pas beaucoup moins performante que la bibliothèque double-double. Et puis, la bibliothèque double-double se base sur le mode d'arrondi au plus près, donc les transformation exactes sur la sommation et la renormalisation sont plus simples. Mais les opérations en double

précision du SPU ne sont pas complètement pipelinées. Chaque opération en double précision coûte 13 cycles d'horloge avec un temps d'attente de six cycles d'horloge. Cela veut dire que ce n'est possible de lancer une opération en double précision que tous les sept cycles d'horloge.

Pour la bibliothèque quad-simple, il n'y a pas de grande différence entre les performances théoriques et la performances expérimentales (voir tableau 5.4) parce que le temps d'exécution des fonctions de la bibliothèque est très grand par rapport au temps de changement de contexte. Et puis, le temps d'exécution est beaucoup plus grand que le temps de communication donc il n'y a pas de phénomène de saturation du système de communication.

5.2.2 Exactitude

Avec la même procédure qu'avec la bibliothèque double-simple, nous avons obtenus la précision réelle de la bibliothèque quad-simple. Les résultats sont résumés dans le tableau 5.5 suivant.

Opération	**Différence maximale**	**Différence moyenne**
Addition	7.675e-30	1.241e-36
Multiplication	3.259e-29	1.576e-30
Division	4.742e-29	2.835e-30

TAB. 5.5 – L'exactitude de la bibliothèque quad-simple

Conclusion et perspectives

Mon stage a commencé le 1er mars. Il se compose de 2 parties :
- la partie bibliographique (2 mois)
- la partie expérimentale (4 mois)

J'ai fait la partie bibliographique pendant les 2 premiers mois en étudiant
- les algorithmes flottants et la méthodologie d'augmentation de précision,
- le processeur CELL et sa programmation parallèle.

La deuxième partie de mon stage consiste à implémenter les précisions étendues sur le processeur CELL. Plus exactement, j'ai implémenté une bibliothèques double-simple avec trois versions d'algorithme et quad-simple qui permettent les programmes de faire des calculs précis jusqu'à 42 bits et 94 bits respectivement. Ces deux bibliothèques se basent seulement sur des opérations en simple précision sous le mode d'arrondi vers zéro tels que la sommation, le produit, le FMA et l'inversion. J'ai implémenté aussi une bibliothèque double-double qui se base sur des opérations en double précision sous le mode d'arrondi au plus près. Dans le cadre de mon stage, je n'ai implémenté et optimisé que pour les opérations de base sur des nombres en précision étendue :
- la sommation/soustraction,
- le produit,
- la division.

La preuve de la précision de ces opérations est aussi fournie dans ce rapport, qui me permet de constater que la bibliothèque double-simple est précise jusqu'à 42 bits et la bibliothèque quad-simple est précise jusqu'à 94 bits.

Nous avons fait au total 11 théorèmes avec 9 démonstrations. Quant aux trois bibliothèque, nous avons fait plus de 2000 lignes de code.

La performance des ces deux bibliothèques est testée par un programme qui effectue la sommation, le produit et la division sur deux grand vecteurs de double-simples et de quad-simples respectivement.

Ces deux bibliothèques ne traitent pas des exceptions. Dans le futur, on peut ajouter des mécanismes pour détecter des anomalies (par exemple l'infini ou une division par zéro). Ces deux bibliothèques peuvent être complétées par des opérations binaires, des opérations algébriques (la puissance N, la racine carrée, et la racine de l'ordre N), des opérations transcendantes (exposant, logarithme, fonctions hyperboliques, ...). Les algorithmes pour ces opérations utilisent la méthode d'itération de Newton et se base sur des opérations déjà implémentées.

Notre travail atteindra son objectif lorsque le consortium IBM, Toshiba, Sony proposeront un processeur CELL avec un SIMD 64 bits. On pourra alors atteindre la précision étendue à 256 bits.

Annexe A

Démonstration des algorithmes

A.1 La renormalisation pour le double-simple

Algorithme

```
1     Renormalise2 (a,b)
2         epsilon = 2^(-23)
3         if (|a| < |b|)
4             swap(a,b)
5         s = fl(a + b)
6         d = fl(s - a)
7         e = fl(b - d)
8         splus = fl(s + s * epssilon)
9         u = splus - s
10        eplus = e - u
11        if (|e| > |eplus|)
12            s = splus, e = eplus
13        if (|2 * b| < |d|)
14            s = a, e = b
15    return (s,e)
```

Théorème

Soit a et b deux nombre flottants de simple précision. Le résultat de l'algorithme `Renormalise2` sur ces deux flottants est un couple de deux flottants de simple précision (s, e) qui satisfait :

$$s + e = a + b \text{ et } |e| \leq \frac{1}{2} ulp(s)$$

L'égalité de la deuxième relation se trouve quand e et s ont la même signe.

Démonstration

Après l'instruction de la ligne 4 on a $|a| \geq |b|$. Dans cette partie nous ne traitons que le cas $a > 0$. Le cas $a < 0$ est symétrique et donc la preuve est identique.

D'après la démonstration de `Two-Sum-toward-zero2` (1.2.2), après l'instruction de la ligne 7 on a toujours :

$$d = fl(s-a) = s-a \text{ et } e = fl(b-d) = fl(err(a+b)).$$

$|a| \geq |b|$, $a > 0$ donc $a+b > 0$. Avec le mode d'arrondi vers zero on a $ulp(s) > err(a+b) \geq 0$.

Il y a deux cas $|2*b| < |d|$ et $|2*b| \geq |d|$ chacun nous donne un résultat différent. Nous allons traiter ces deux cas.

Cas $|2*b| < |d|$

Dans ce cas, le résultat retourné de cet algorithme est exactement les deux entrées (a, b).

On a

$$\begin{aligned} d &= s-a \\ &= (a+b-err(a+b))-a \\ &= b-err(a+b) \end{aligned}$$

Si $b \geq 0$ d'où $a+b \geq a$ et $s = fl(a+b) \geq a$.

Cela veut dire $d = s-a \geq 0$ d'où

$$\begin{aligned} |2*b| &< |d| \\ \Leftrightarrow 2*b &< b-err(a+b) \\ \Leftrightarrow b &< -err(a+b) \leq 0 \\ \rightarrow b &< 0. \end{aligned}$$

Donc il faut que $b < 0$. D'où

$$\begin{aligned} |2*b| &< |d| \\ \Leftrightarrow -2*b &< err(a+b)-b \\ \Leftrightarrow -b &< err(a+b) \\ \rightarrow -b &< err(a+b) < ulp(s). \end{aligned}$$

On a

$$\begin{aligned} a &= (a+b)-b \\ &= (s+err(a+b))+(-b) \\ &< s+2*ulp(s) \end{aligned}$$

On a aussi $a > a+b \geq fl(a+b) = s$.

Donc $a = s+ulp(s)$.

De plus

$$\begin{aligned} a &= (a+b)-b \\ &= (s+err(a+b))+(-b) \\ &> s+2*|b| \end{aligned}$$

d'où $|b| < \frac{1}{2}ulp(s) \leq \frac{1}{2}ulp(a)$.

Cas $|2 * b| \geq |d|$

Dans ce cas, d'après la démonstration de `Two-Sum-toward-zero2` (1.2.2) $err(a+b)$ est représentable et que $e = fl(err(a+b)) = err(a+b)$.

D'après la partie 1.1

$$\frac{\varepsilon}{2}|x| < ulp(x) \leq \varepsilon|x|$$

d'où $ulp(s) \leq epsilon * s < 2 * ulp(s)$.
Donc $splus = fl(s + epsilon * s) = s + ulp(s)$.
D'où $u = fl(splus - s) = ulp(s)$
On a $0 \leq err(a+b) < ulp(s)$ d'où

$$\begin{aligned} 0 &\leq e < u \\ \rightarrow 0 &\geq fl(e-s) \geq e - s \\ \Leftrightarrow 0 &\geq eplus \geq e - u \\ \rightarrow |e| + |eplus| &\leq ulp(s). \end{aligned}$$

- Si $|e| \leq |eplus|$ on déduit que $|e| \leq \frac{1}{2}ulp(s)$. D'apès l'instruction de la ligne 11, les valeurs de s et e ne changent pas. Donc le résultat retourné est (s, e) qui satisfait

$$\begin{aligned} s + e &= fl(a+b) + err(a+b) = a + b \\ \text{et } |e| &\leq \frac{1}{2}ulp(s). \end{aligned}$$

- Si $|e| > |eplus|$, cela veut dire que $|e| > |fl(e-u)|$. D'après le mode d'arrondi vers zéro, $fl(e-u)$ est un nombre flottant de simple précision qui est le plus proche de $(e-u)$ et satisfait $|fl(e-u)| < |e-u|$. Donc $|e| \geq |e-u|$, d'où $e \geq (u-e)$. Cela veut dire $2 * e \geq u > e$. D'où $e - u$ est exact et représentable par un flottant de simple précision.

$$eplus = fl(e-u) = e - u = e - ulp(s).$$

$$\begin{aligned} \rightarrow splus + eplus &= (s + ulp(s)) + (e - ulp(s)) \\ &= s + e \\ &= fl(a+b) + err(a+b) \\ &= a + b. \end{aligned}$$

De plus $|eplus| < |e|$ d'où $|eplus| < \frac{1}{2}(|eplus| + |e|) = \frac{1}{2}ulp(s) \leq \frac{1}{2}ulp(splus)$.
Donc le résultat retourné est $(splus, eplus)$ qui satisfait :

$$splus + eplus = a + b$$

$$\text{et } |eplus| < \frac{1}{2}ulp(splus)$$

A.2 La sommation de deux double-simples

Algorithme

```
1       add_ds_ds (a0, a1, b0, b1)
2           (t0, t1) = Two–Sum–toward–zero (a0, b0)
3           t2 = fl(a1 + b1)
4           t3 = fl(t1 + t2)
5           (b0, b1) = Renormalise2 (t0, t3)
6       return (b0, b1)
```

Théorème

Soit $a_h + a_l$ et $b_h + b_l$ deux double-simples. Soit $r_h + r_l$ le résultat retourné par l'algorithme `add_ds_ds` sur $a_h + a_l$ et $b_h + b_l$. L'erreur de cet algorithme δ satisfait :

$$r_h + r_l = (a_h + a_l) + (b_h + b_l) + \delta$$

et est bornée par :

$$|\delta| < max(2^{-23} \times |a_l + b_l|, 2^{-43} \times |a_h + a_l + b_h + b_l|) + 2^{-45}|a_h + a_l o + b_h i + b_l o|.$$

Démonstration

Comme `Two-Sum-toward-zero` est la transformation exacte sur la somme, il ne produit pas d'erreur de calcul. Donc l'erreur de cet algorithme est causé par les deux instructions des lignes 3,4 :

$$err = err(a1 + b1) + err(t1 + t2).$$

L'erreur de la sommation de la ligne 3 satisfait :

$$\begin{aligned} |err(a1 + b1)| &< \varepsilon \times |a1 + b1| \\ &< 2^{-23} \times |a1 + b1|. \end{aligned}$$

Il existe deux cas : $t1 = 0$ et $t1 \neq 0$. Nous allons traiter ces deux cas.

Cas $t1 = 0$

D'où la sommation de la ligne 4 est exacte et ne donne pas d'erreur d'arrondi. Donc l'erreur de cet algorithme est :

$$\begin{aligned} err &= err(a1 + b1) \\ \rightarrow |err| &< 2^{-23} \times |a1 + b1|. \end{aligned}$$

Cas $t1 \neq 0$

Si $a0$, $b0$ ne disposent pas le même signe et que

$$min(|a0|, |b0|) \geq \frac{1}{2} max(|a0|, |b0|),$$

la sommation $a0 + b0$ sera exacte et il n'y a pas d'erreur d'arrondi, soit $t1 = 0$. Donc il faut que

– $a0, b0$ ont le même signe ou
– $min(|a0|, |b0|) \geq \frac{1}{2} max(|a0|, |b0|)$.

Cela nous permet d'avoir :

$$\begin{aligned} |a0 + b0| &> \frac{1}{2} max(|a0|, |b0|) \\ \rightarrow |a0 + b0| &\geq \frac{1}{2} max(|a0|, |b0|) \\ \rightarrow |a0 + b0| &\geq \frac{1}{2} max(|a0|, |b0|) \\ \rightarrow |a0 + b0| &> \frac{1}{3} (|a0| + |b0|), \end{aligned}$$

et puis

$$\begin{aligned} |a1 + b1| &\leq |a1| + |b1| \\ &< \varepsilon \times |a0| + \varepsilon \times |b0| \\ &< \varepsilon \times (|a0| + |b0|) \\ &< 3 \times \varepsilon \times |a0 + b0|. \end{aligned}$$

D'où

$$\begin{aligned} |a0 + a1 + b0 + b1| &\leq |a0 + b0| - |a1 + b1| \\ &> |a0 + b0| - 3 \times \varepsilon \times |a0 + b0| \\ &> (1 - 3 \times \varepsilon) \times |a0 + b0|. \end{aligned}$$

La transformation exacte sur la somme nous donne la relation :

$$|t1| < \varepsilon \times |a0 + b0|.$$

Le résultat et l'erreur de la sommation de la ligne 3 satisfont :

$$\begin{aligned} |t2| &\leq |a1 + b1| \\ &< 3 \times \varepsilon \times |a0 + b0|, \\ \text{et } |err(a1 + b1)| &< \varepsilon \times |t2| \\ &< 3 \times \varepsilon^2 \times |a0 + b0|. \end{aligned}$$

Le résultat et l'erreur de la sommation de la ligne 4 satisfont :

$$\begin{aligned} |t3| &\leq |t1+t2| \\ &\leq |t1|+|t2| \\ &< \varepsilon \times |a0+b0| + 3 \times \varepsilon \times |a0+b0| \\ &< 4 \times \varepsilon \times |a0+b0|, \\ \text{et } |err(t1+t2)| &< \varepsilon \times |t3| \\ &< 4 \times \varepsilon^2 \times |a0+b0|. \end{aligned}$$

Donc l'erreur de cet algorthme satisfait :

$$\begin{aligned} |err| &= |err(a1+b1) + err(t1+t2)| \\ &\leq |err(a1+b1)| + |err(t1+t2)| \\ &< 3 \times \varepsilon^2 \times |a0+b0| + 4 \times \varepsilon^2 \times |a0+b0| \\ &< 7 \times \varepsilon^2 \times |a0+b0| \\ &< 7 \times \varepsilon^2 \times \frac{|a0+a1+b0+b1|}{(1-3\times\varepsilon)} \\ &< 7 \times (2^{-23})^2 \times \frac{|a0+a1+b0+b1|}{(1-3\times 2^{-23})} \\ &< 2^{-43} \times |a0+a1+b0+b1|). \end{aligned}$$

Avec l'erreur de la renormalisation on a :

$$|err| < max(2^{-23} times |a1+b1|, 2^{-43} \times |a0+a1+b0+b1|) + 2{-}45|a0+a1+b0+b1|.$$

A.3 Le produit de double-simples

Algorithme

```
1  mul_ds_ds (a0, a1, b0, b1)
2      (t0, t1) = Two-Product-FMA (a0, b0)
3      t2 = fl(a1 * b0)
4      t3 = fl(a0 * b1 + t2)
5      t4 = fl(t3 + t2)
6      (p0, p1) = Renormalise2 (t0, t4)
7  return (p0, p1)
```

Théorème

Soit $a_h + a_l$ et $b_h + b_l$ deux double-simples. Soit $r_h + r_l$ le résultat retourné par l'algorithme `mul_ds_ds` sur $a_h + a_l$ et $b_h + b_l$. L'erreur de cet algorithme δ satisfait :

$$|(r_h+r_l)-(a_h+a_l)\times(b_h+b_l)| < 3\times 2^{-45} \times |(a_h+a_l)\times(b_h+b_l)| + 9\times 2^{-68} \times |(a_h+a_l)\times(b_h+b_l)|$$

Démonstration

Comme `Two-Product-FMA` et `Renormalise2` donne le résultat exact, l'erreur de cet algorithme est causée par les trois instructions des lignes 3, 4, 5 :

$$err = err(a1 * b0) + err(a0 * b1 + t2) + err(t1 + t2).$$

On sait que $|fl(a \circ b)| \leq |a \circ b|$ et $err(a \circ b) < \varepsilon |fl(a \circ b)|$. Donc :

$$\begin{aligned}
|t2| &\leq |a1 \times b0| \\
&< \varepsilon \times |a0 \times b0|, \\
|err(a1 \times b0)| &< \varepsilon \times |t2| \\
&< \varepsilon^2 \times |a0 \times b0|, \\
|t3| &\leq |a0 \times b1 + t2| \\
&\leq |a0 \times b1| + |t2| \\
&< \varepsilon \times |a0 \times b0| + \varepsilon \times |a0 \times b0| \\
&< 2 \times \varepsilon \times |a0 \times b0|, \\
|err(a0 \times b1 + t2)| &< \varepsilon \times |t3| \\
&< 2 \times \varepsilon^2 \times |a0 \times b0|, \\
|t4| &\leq |t3 + t2| \\
&\leq |t3| + |t2| \\
&< \varepsilon \times |a0 \times b0| + 2 \times \varepsilon \times |a0 \times b0| \\
&< 3 \times \varepsilon \times |a0 \times b0|, \\
|err(t3 + t2)| &< \varepsilon \times |t4| \\
&< 3 \times \varepsilon^2 \times |a0 \times b0|.
\end{aligned}$$

D'où

$$\begin{aligned}
|err| &= |err(a1 \times b0) + err(a0 \times b1 + t2) + err(t1 + t2)| \\
&\leq |err(a1 \times b0)| + |err(a0 \times b1 + t2)| + |err(t1 + t2)| \\
&< \varepsilon^2 \times |a0 \times b0| + 2 \times \varepsilon \times |a0 \times b0| + 3 \times \varepsilon^2 \times |a0 \times b0| \\
&< 6 \times \varepsilon^2 \times |a0 \times b0|.
\end{aligned}$$

De plus

$$\begin{aligned}
|a0| &< (1 + \varepsilon)|a0 + a1| \\
\text{et } |b0| &< (1 + \varepsilon)|b0 + b1|.
\end{aligned}$$

Donc

$$\begin{aligned}
|err| &< 6 \times \varepsilon^2 \times |a0 \times b0| \\
&< 6 \times \varepsilon^2 \times (1 + \varepsilon)^2 \times |(a0 + a1) \times (b0 + b1)|
\end{aligned}$$

Comme $\varepsilon = 2^{-23}$, on a

$$\begin{aligned}
|err| &< 6 \times \varepsilon^2 \times |a0 \times b0| \\
&< 6 \times 2^{-46} \times (1 + 2^{-23})^2 \times |(a0 + a1) \times (b0 + b1)| \\
&< 3 \times 2^{-45} \times |(a0 + a1) \times (b0 + b1)| + 9 \times 2^{-68} \times |(a0 + a1) \times (b0 + b1)|
\end{aligned}$$

Avec l'erreur de la renormalisation qui est bornée par $2^{-45}|p0+p1|$, on a

$$|err| < 2^{-43} \times |(a0+a1) \times (b0+b1)| + 9 \times 2^{-68} \times |(a0+a1) \times (b0+b1)|$$

A.4 L'inversion d'un nombre flottant

Rappelons l'algorithme `inverse` et le théorème d'erreur de calcul.

Algorithme

```
1       inverse b
2          tmp0 = spu_re(b)
3          rerr0 = fl(1 - tmp0 * b)
4          tmp1 = fl(tmp0 + rerr0 * tmp0)
5          rerr = fl(1 - tmp1 * b)
6          eerr = fl(rerr * tmp1)
7       return (tmp2, rerr, eerr)
```

Théorème

Soit b un nombre flottant de simple précision, ε l'erreur relative en simple précision. L'erreur relative de l'algorithme `inverse` est bornée par :

$$\frac{|inverse(b) - 1/b|}{|1/b|} < 2 \times \varepsilon + \mathcal{O}(\varepsilon^{3/2})$$

Démonstration

La fonction `spu_re` fournit par le FPU nous permet d'avoir une approximation précis à 12 bits de la valeur inverse d'un flottant. Cela nous donne :

$$\begin{aligned} tmp0 &= (1+\eta_0) \times (1/b) \\ \text{avec } |\eta_0| &< 2^{-12} \\ \text{d'où } |\eta_0| &< \varepsilon^{1/2} \\ \rightarrow |\eta_0^2| &< \varepsilon \end{aligned}$$

En utilisant la fonction FMA, l'instruction de la ligne 3 nous donne :

$$\begin{aligned} rerr0 &= (1+\delta_0) \times (1 - tmp0 \times b) \text{avec } |\delta_0| < \varepsilon \\ &= (1+\delta_0) \times (1 - (1+\eta_0) \times (1/b) \times b) \\ &= (1+\delta_0) \times (-\eta_0) \end{aligned}$$

D'après l'instruction de la ligne 4 :

$$\begin{aligned} tmp1 &= (1+\delta_1)\times(tmp0 + rerr0 \times tmp0) \text{avec } |\delta_1| < \varepsilon \\ &= (1+\delta_1)\times tmp0 \times (1+rerr0) \\ &= (1+\delta_1)\times(1+\eta_0)\times(1/b)\times(1+(1+\delta_0)\times(-\eta_0)) \\ &= (1+\delta_1)\times(1+\eta_0)\times(1-\eta_0-\eta_0\times\delta_0)\times(1/b) \\ &= (1+\delta_1)\times(1-\eta_0^2-\eta_0\times\delta_0\times(1+\eta_0))\times(1/b) \\ &= (1-\eta_0^2+\delta_1-\delta_1\times\eta_0^2-\eta_0\times\delta_0\times(1+\eta_0)\times(1+\delta_1))\times(1/b) \end{aligned}$$

Posons

$$\begin{aligned} \eta_1 &= -\eta_0^2+\delta_1-\delta_1\times\eta_0^2-\eta_0\times\delta_0\times(1+\eta_0)\times(1+\delta_1) \\ \text{d'où } |\eta_1| &< |\eta_0^2| + |\delta_1| + |\delta_1|\times|\eta_0^2| + |\eta_0|\times|\delta_0|\times(1+|\eta_0|)\times(1+|\delta_1|) \\ &< \varepsilon+\varepsilon+\varepsilon\times\varepsilon+\varepsilon^{1/2}\times\varepsilon\times(1+\varepsilon^{1/2})\times(1+\varepsilon) \\ &< 2\times\varepsilon+\mathcal{O}(\varepsilon^{3/2}) \\ \rightarrow |\eta_1^2| &< 5\times\varepsilon^2 \end{aligned}$$

Donc

$$tmp1 = (1+\eta_1)/b \text{ avec } |\eta_1| < 2\times\varepsilon+\mathcal{O}(\varepsilon^{3/2})$$

L'instruction de la ligne 5 nous donne une estimation de l'erreur relative du résultat, soit η_1 :

$$\begin{aligned} rerr &= fl(1-tmp2\times b) \\ &= (1+\delta_2)\times(1-((1+\eta_1)/b)\times b) \text{ avec } |\delta_2|<\varepsilon \\ &= (1+\delta_2)\times(-\eta_1) \end{aligned}$$

Et donc l'instruction de la ligne 6 calcule l'erreur exacte du résultat :

$$\begin{aligned} eerr &= fl(rerr\times tmp1) \\ &= (1+\delta_3)\times(rerr\times tmp1) \text{ avec } |\delta_3|<\varepsilon \\ &= (1+\delta_3)\times(1+\delta_2)\times(-\eta_1)\times(1+\eta_1)/b \end{aligned}$$

Comme $|\delta_2| < \varepsilon$, $|\delta_3| < \varepsilon$ et $|\eta_1| < 2\times\varepsilon+\mathcal{O}(\varepsilon^{3/2})$, $eerr$ peut s'écrire de la manière suivante :

$$\begin{aligned} eerr &= (1+\delta_3)\times(1+\delta_2)\times(-\eta_1)\times(1+\eta_1)/b \\ &= (1+\delta_4)\times(-\eta_1)/b \\ &= (1+\delta_4)\times(b-tmp1) \\ \text{avec } |\delta_4| &< 5\times\varepsilon \end{aligned}$$

A.5 La division en simple précision

Algorithme

```
1       div (a, b)
2         (invb, rerr, eerr) = inverse(b)
3         tmp = fl(eerr * a)
4         q = fl(tmp + a * invb)
5       return q
```

Théorème

Soit a, b deux nombres flottants de simple précision, ε l'erreur relative en simple précision. L'erreur relative de l'algorithme `div` est bornée par :

$$\frac{|div(a,b) - a/b|}{|a/b|} < \varepsilon + \mathcal{O}(\varepsilon^2)$$

Démonstration

D'après la démonstration précédente de l'algorithme d'inversion, le résultat retourné par l'instruction de la ligne 2 satisfait :

$$\begin{aligned} invb &= (1+\eta)/b \text{ avec } |\eta| < 2 \times \varepsilon + \mathcal{O}(\varepsilon^{3/2}) \\ rerr &= (1+\delta_1) \times (-\eta) \text{ avec } |\delta_1| < \varepsilon \\ eerr &= (1+\delta_2) \times (-\eta)/b \text{ avec } |\delta_2| < 5 \times \varepsilon \end{aligned}$$

L'instruction de la ligne 3 nous donne :

$$\begin{aligned} tmp &= fl(eerr \times a) \\ &= (1+\delta_3)(eerr \times a) \text{ avec } |\delta_3| < \varepsilon \\ &= (1+\delta_3) \times (1+\delta_2) \times (-\eta) \times (a/b) \end{aligned}$$

La division de a par b est calculée par l'instruction de la ligne 4 :

$$\begin{aligned} q &= fl(tmp + a \times invb) \\ &= (1+\delta_4) \times (tmp + a \times invb) \text{ avec } |\delta_3| < \varepsilon \\ &= (1+\delta_4) \times ((1+\delta_3) \times (1+\delta_2) \times (-\eta) \times (a/b) + a \times (1+\eta)/b) \\ &= (1+\delta_4) \times (1 + \eta \times (1 - (1+\delta_3) \times (1+\delta_2))) \times (a/b) \\ &= (1+\delta_4) \times (1 + \eta \times (-\delta_3 - \delta_2 - \delta_3 \times \delta_2)) \times (a/b) \end{aligned}$$

Posons

$$\begin{aligned} \delta &= (1+\delta_4) \times (1 + \eta \times (-\delta_3 - \delta_2 - \delta_3 \times \delta_2)) - 1 \\ &= \delta_4 + \eta \times (-\delta_3 - \delta_2 - \delta_3 \times \delta_2) \\ \text{d'où } |\delta| &< |\delta_4| + |\eta| \times (|\delta_3| + |\delta_2| + |\delta_3| \times |\delta_2|) \\ &< \varepsilon + (2 \times \varepsilon + \mathcal{O}(\varepsilon^{3/2})) \times (\varepsilon + 5 \times \varepsilon + \varepsilon \times 5 \times \varepsilon) \\ &< \varepsilon + 12 \times \varepsilon^2 + \mathcal{O}(\varepsilon^{5/2}) \\ &< \varepsilon + \mathcal{O}(\varepsilon^2) \end{aligned}$$

q peut être écrit de la manière suivante :

$$\begin{aligned} q &= (1+\delta) \times (b/a) \text{ avec } |\delta| < \varepsilon + \mathcal{O}(\varepsilon^2) \\ \text{d'où } \frac{|q - b/a|}{|b/a|} &< \varepsilon + \mathcal{O}(\varepsilon^2) \end{aligned}$$

A.6 La division de deux double-simples

Algorithme

```
1  div_ds_ds(a,b)
2      p0 = fl(a0 / b0)
3      tmp1 = fl(a0 - q0 * b0)
4      tmp2 = fl(a1 - q0 * b1)
5      r = fl(tmp1 + tmp2)
6      p1 = fl(r / b_0)
7      (q0,q1) = Quick-Sum-toward-zero(p0, p1)
8      return (q0,q1)
```

Théorème

Soit $a = (a_0, a_1)$ et $b = (b_0, b_1)$ deux double-simples, ε l'erreur relative en simple précision, et ε_1 la borne d'erreur relative de la division en simple précision avec $\mathcal{O}(\varepsilon_1) = \mathcal{O}\varepsilon$. L'erreur de l'algorithme `div_ds_ds` est bornée par :

$$\frac{|div\_ds\_ds(a,b)) - a/b|}{|a/b|} < \varepsilon^2 \times (6 + 7 \times \varepsilon_1/\varepsilon + 2 \times (\varepsilon_1/\varepsilon)^2) + \mathcal{O}(\varepsilon^3))$$

Démonstration

Comme la borne d'erreur relative de la division en simple précision est ε, la valeur de p_0 peut s'écrire de la manière suivante :

$$\begin{aligned} p_0 &= (1+\delta_0)a_0/b_0 \text{ avec } \delta_0 < \varepsilon_1 \\ \text{d'où } p_0 \times b_0 &= (1+\delta_0) \times a_0 \\ \text{et } p_0 \times b_0 - a_0 &= \delta_0 \times a_0 \\ \rightarrow |p_0 \times b_0| &< (1+\varepsilon_1) \times |a_0| \\ \text{et } |p_0 \times b_0 - a_0| &< \varepsilon_1 \times |a_0| \end{aligned}$$

En utilisant la fonction FMA, l'instruction de la ligne 3 nous donne :

$$\begin{aligned}
tmp1 - (a_0 - p_0 \times b_0) &= \delta_1 \times (a_0 - p_0 \times b_0) \text{ avec } |\delta_1| < \varepsilon \\
\text{et } tmp1 &= (1+\delta_1)(a_0 - p_0 \times b_0)) \\
\text{d'où } |tmp1 - (a_0 - p_0 \times b_0)| &< \varepsilon \times |a_0 - p_0 \times b_0| \\
&< \varepsilon \times \varepsilon_1 \times |a_0| \\
\text{et } |tmp1| &< (1+\varepsilon) \times |a_0 - p_0 \times b_0| \\
&< (1+\varepsilon) \times \varepsilon_1 \times |a_0|
\end{aligned}$$

Comme $|a_1| < \varepsilon \times |a_0|$ et $|b_1| < \varepsilon \times |b_0|$, on a

$$\begin{aligned}
|a_1 - p_0 \times b_1| &< |a_1| + |p_0| \times |b_1| \\
&< \varepsilon \times |a_0| + |p_0| \times \varepsilon \times |b_0| \\
&< \varepsilon \times |a_0| + \varepsilon \times |p_0 \times b_0| \\
&< \varepsilon \times |a_0| + \varepsilon \times (1+\varepsilon_1) \times |a_0| \\
&< \varepsilon \times (2+\varepsilon_1) \times |a_0|
\end{aligned}$$

L'instruction de la ligne 4 nous donne :

$$\begin{aligned}
tmp2 - (a_1 - p_0 \times b_1) &= \delta_2 \times (a_1 - p_0 \times b_1) \text{ avec } |\delta_2| < \varepsilon \\
\text{et } tmp2 &= (1+\delta_2)(a_1 - p_0 \times b_1) \\
\text{d'où } |tmp2 - (a_1 - p_0 \times b_1)| &< \varepsilon \times |a_1 - p_0 \times b_1| \\
&< \varepsilon \times \varepsilon \times (2+\varepsilon_1) \times |a_0| \\
&< \varepsilon^2 \times (2+\varepsilon_1) \times |a_0| \\
\text{et } |tmp2| &< (1+\varepsilon) \times |a_1 - p_0 \times b_1| \\
&< (1+\varepsilon) \times \varepsilon \times (2+\varepsilon_1) \times |a_0|
\end{aligned}$$

On peut déduire que

$$\begin{aligned}
|tmp1 + tmp2| &\leq |tmp1| + |tmp2| \\
&< (1+\varepsilon) \times \varepsilon_1 \times |a_0| + (1+\varepsilon) \times \varepsilon \times (2+\varepsilon_1) \times |a_0| \\
&< \varepsilon \times (1+\varepsilon) \times (2 + \varepsilon_1/\varepsilon + \varepsilon_1) \times |a_0|
\end{aligned}$$

D'après les instructions des lignes 5, 6

$$\begin{aligned}
p_1 &= fl(r/b_0) \\
&= (1+\delta_3) \times (r/b_0) \text{ avec } |\delta_3| < \varepsilon_1 \\
&= (1+\delta_3) \times (fl(tmp1 + tmp2)/b_0) \\
&= (1+\delta_3) \times (1+\delta_4) \times (tmp1 + tmp2)/b_0 \text{ avec } |\delta_4| < \varepsilon \\
\text{d'où } p_1 \times b_0 &= (1+\delta_3) \times (1+\delta_4) \times (tmp1 + tmp2) \\
\rightarrow |p_1 \times b_0| &< (1+\varepsilon_1) \times (1+\varepsilon) \times |tmp1 + tmp2| \\
&< (1+\varepsilon_1) \times (1+\varepsilon) \times \varepsilon \times (1+\varepsilon) \times (2 + \varepsilon_1/\varepsilon + \varepsilon_1) \times |a_0| \\
&< \varepsilon \times (1+\varepsilon)^2 \times (1+\varepsilon_1) \times (2 + \varepsilon_1/\varepsilon + \varepsilon_1) \times |a_0|
\end{aligned}$$

L'instruction de la ligne 7 peut être considéré comme le processus de renormalisation qui nous permet d'avoir un double-simple qui représente exactement la somme $(p_0 + p_1)$. Cela veut dire

$$q_0 + q_1 = p_0 + p_1$$

Pour estimer l'erreur de cet algorithme, on calcule la différence entre $q \times b$ et a :

$$\begin{aligned}
q \times b - a &= (q_0 + q_1) \times (b_0 + b_1) - (a_0 + a_1) \\
&= (p_0 + p_1) \times (b_0 + b_1) - (a_0 + a_1) \\
&= (p_0 \times b_0 - a_0) + (p_0 \times b_1 - a_1) + p_1 \times b_0 + p_1 \times b_1 \\
&= (p_0 \times b_0 - a_0) + (p_0 \times b_1 - a_1) + \\
&\quad (1 + \delta_3) \times (1 + \delta_4) \times (tmp1 + tmp2) + \\
&\quad p_1 \times b_1 \\
&= (tmp1 - (a_0 - p_0 \times b_0)) + (tmp2 - (a_1 - p_0 \times b_1)) + \\
&\quad (\delta_3 + \delta_4 + \delta_3 \times \delta_4) \times (tmp1 + tmp2) + \\
&\quad p_1 \times b_1 \\
\text{d'où } |q \times b - a| &< |tmp1 - (a_0 - p_0 \times b_0)| + |tmp2 - (a_1 - p_0 \times b_1)| + \\
&\quad (|\delta_3| + |\delta_4| + |\delta_3| \times |\delta_4|) \times |tmp1 + tmp2| + \\
&\quad |p_1| \times |b_1| \\
&< \varepsilon \times \varepsilon_1 \times |a_0| + \varepsilon^2 \times (2 + \varepsilon_1) \times |a_0| + \\
&\quad (\varepsilon + \varepsilon_1 + \varepsilon \times \varepsilon_1) \times \varepsilon \times (1 + \varepsilon) \times (2 + \varepsilon_1/\varepsilon + \varepsilon_1) \times |a_0| + \\
&\quad |p_1| \times |b_0| \times \varepsilon \\
&< \varepsilon^2 \times (2 + \varepsilon_1/\varepsilon) \times |a_0| + \\
&\quad \varepsilon^2 \times (1 + \varepsilon_1/\varepsilon + \varepsilon) \times (1 + \varepsilon) \times (2 + \varepsilon_1/\varepsilon + \varepsilon_1) \times |a_0| + \\
&\quad \varepsilon \times \varepsilon \times (1 + \varepsilon)^2 \times (1 + \varepsilon_1) \times (2 + \varepsilon_1/\varepsilon + \varepsilon_1) \times |a_0| \\
\text{Comme } \mathcal{O}\varepsilon_1 &= \mathcal{O}\varepsilon \\
|q \times b - a| &< \varepsilon^2 \times (2 + \varepsilon_1/\varepsilon + 2 \times (1 + \varepsilon_1/\varepsilon) \times (2 + \varepsilon_1/\varepsilon)) \times |a_0| + \\
&\quad \mathcal{O}(\varepsilon^3) \times |a_0| \\
&< \varepsilon^2 \times (6 + 7 \times \varepsilon_1/\varepsilon + 2 \times (\varepsilon_1/\varepsilon)^2) \times |a_0| + \mathcal{O}(\varepsilon^3) \times |a_0|
\end{aligned}$$

D'après la normalisation imposée sur deux composantes d'un double-simple

$$\begin{aligned}
|a_1| &< \varepsilon \times |a_0| \\
\text{d'où } |a_0 + a_1| &> |a_0| - \varepsilon \times |a_0| \\
&> (1 - \varepsilon) \times |a_0| \\
\text{ou bien } |a_0| &< |a|/(1 - \varepsilon)
\end{aligned}$$

Cela nous donne

$$\begin{aligned}
|q \times b - a| &< \varepsilon^2 \times (6 + 7 \times \varepsilon_1/\varepsilon + 2 \times (\varepsilon_1/\varepsilon)^2) \times |a_0| + \mathcal{O}(\varepsilon^3) \times |a_0| \\
&< \varepsilon^2 \times (6 + 7 \times \varepsilon_1/\varepsilon + 2 \times (\varepsilon_1/\varepsilon)^2) \times |a_0| + \mathcal{O}(\varepsilon^3) \times |a|/(1 - \varepsilon) \\
&< \varepsilon^2 \times (6 + 7 \times \varepsilon_1/\varepsilon + 2 \times (\varepsilon_1/\varepsilon)^2) \times |a| + \mathcal{O}(\varepsilon^3) \times |a| \\
\text{d'où } \frac{|q - a/b|}{|a/b|} &< \varepsilon^2 \times (6 + 7 \times \varepsilon_1/\varepsilon + 2 \times (\varepsilon_1/\varepsilon)^2) + \mathcal{O}(\varepsilon^3)
\end{aligned}$$

Avec l'erreur de la renormalisation on a :

$$\frac{|q - a/b|}{|a/b|} \quad < \quad \varepsilon^2 \times (6.5 + 7 \times \varepsilon_1/\varepsilon + 2 \times (\varepsilon_1/\varepsilon)^2) + \mathcal{O}(\varepsilon^3)$$

Annexe B

Résultat des tests

La performance des bibliothèque mesurée en MFLOP en effectuant des opérations sur deux vecteurs de $2^{23} = 8,388,608$ éléments avec la taille des tampons de 16K.

Fonctions	1 SPE	2 SPEs	4 SPEs	8 SPEs
Add_ds_ds_4vect	250.4	482.1	735.8	1075
Mul_ds_ds_4vect	287.2	544.7	735.8	1075
Div_ds_ds_4vect	166.4	326.4	626.0	1089
Add_qs_qs_4vect	25.77	51.21	100.58	196.45
Mul_qs_qs_4vect	20.20	40.25	80.04	150.33
Div_qs_qs_4vect	4.61	9.23	18.44	36.79

TAB. B.1 – Performance des fonctions des bibliothèques double-simple et quad-simple

Bibliographie

[1] http://fr.wikipedia.org/wiki/Cell_(processeur).

[2] IBM STI Design Center Daniel A. Brokenshire (brokensh@us.ibm.com), Senior Technical Staff Member, *Maximizing the power of the cell broadband engine processor : 25 tips to optimal application performance*, http://www-128.ibm.com/developerworks/power/library/pa-celltips1/, Year = 2006.

[3] T.J. Dekker, *A floating-point technique for extending the available precision*, Numer. Math. **18** (1971), 224–242.

[4] Michael Gschwind (IBM) et al, *Synergistic processing in cell's multicore architecture*, IEEE Micro **26** (2006), 10–24.

[5] Yozo Hida, Xiaoye S.Li, and David H.Baily, *Quad-double arithmetic : Algorithms, implementation, and application*, 2000.

[6] Donald Knuth, *The art of computer programming : Seminumerical algorithms*, vol. 2, Reading, Massachusetts : Addison-Wesley, 1969.

[7] Christoph Quirin Lauter, *Basic building blocks for a triple-double intermediate format*, Tech. report, INRIA, 9 2005.

[8] Douglas M.Priest, *On properties of floating point arithmetics : numerical stability and the cost of accurate computations*, University of California at Berkeley, 1992.