International Journal of Innovation and Scientific Research
ISSN 2351-8014 Vol. 54 No. 2 May. 2021, pp. 113-124

http://www.ijisr.issr-journals.org/

Performance d'architecture d'additionneurs sur les entiers chiffrés

[Architectural performance of adders on encrypted integers]

Paulin Boale Bomolo[1], Simon Ntumba Badibanga[2], and Eugene Mbuyi Mukendi[3]

[1]Chef des Travaux, Département des Mathématiques et Informatique, Faculté des Sciences, Université de Kinshasa, RD Congo

[2]Professeur, Département des Mathématiques et Informatique, Faculté des Sciences, Université de Kinshasa, RD Congo

[3]Professeur Ordinaire, Département des Mathématiques et Informatique, Faculté des Sciences, Université de Kinshasa, RD Congo

ABSTRACT: The fully Homomorphic encryption scheme is corner stone of privacy electronic privacy in real life in more connected world. It allows to perform all kinds of computations on encrypted data. Although, time of computations are bottleneck of numerous applications of real life. In this paper, you aim to study the time of computations on multiple inputs encrypted adders with these schemes. Precisely, we present the TFHE scheme which have published by Illaria Chilloti and all. It belongs many kinds of logics gates as AND, OR, NAND. These logics gates are used to design circuit which represents four architectures of multiple encrypted inputs adders.
Finally, we determine the best architecture of adder which is Carry Look-Ahead Adder (CLA). CLA reduce to fifteen precent (15%) time of computations of dedicated architecture like Wallace.

KEYWORDS: Fully Homomorphic encryption, bootstrapping, logic gate, binary adder.

RESUME: Le schéma de chiffrement homomorphique complet est la pierre angulaire dans la protection de la vie courante dans un monde de plus en plus connecté. Il permet d'effectuer des traitements arbitraires et diverses sur les cryptogrammes ou les données chiffrées. Quoique, le temps de traitement constitue un goulot d'étranglement pour une gamme plus large d'application dans la vie courante. Dans cet article, notre objectif d'étudier la performance de ces schémas sur les architectures d'additionneurs à entrées multiples. En particulier, notre choix a été porté sur le schéma de chiffrement homomorphique complet TFHE qui offre une gamme assez des portes logiques que sont ET, OU, NONET. Ces portes ont été constituées en circuit pour construire quatre types d'architectures d'additionneurs en vue d'effectuer l'addition sur plusieurs nombres.
Enfin, une étude comparative a été réalisée pour déterminer la meilleure en terme de temps d'exécution est l'additionneur à retenue anticipée. Elle réduit de 15 % le temps d'exécution des architectures dédiées tel que celle de Wallace.

MOTS-CLEFS: Chiffrement homomorphique complet, porte logique, additionneur binaire, bootstrapping.

1 INTRODUCTION

Le chiffrement homomorphique permet d'effectuer des traitements sur les données chiffrées sans au préalable les déchiffrer. Ce concept est resté longtemps un problème ouvert jusqu'à la percée de Gentry en 2009 [4] qui a montrée dans sa thèse la possibilité d'appliquer une fonction quelconque sur les données chiffrées.

Corresponding Author: Paulin Boale Bomolo

Dans le chiffrement homomorphique, les messages sont chiffrés par un masquage d'une valeur appelé bruit et le déchiffrement consiste à enlever ledit bruit pour retrouver le message original. Ledit bruit augmente en valeur après chaque évaluation homomorphique d'une opération élémentaire. Le schéma de chiffrement peu homomorphique évalué un nombre limité d'opérations diverses jusqu'à un seuil où le déchiffrement devient invalide. Ce nombre peut-être asymptotiquement rendu illimité par la technique du bootstrapping.

Ladite technique introduite par Gentry réduit la valeur du bruit dans le message chiffré résultant permet une évaluation homomorphique des circuits arbitraires et y compris son propre circuit de déchiffrement. Elle est très coûteuse en temps et en espace. Depuis, plusieurs améliorations ont été proposées soit en termes d'efficacité [6], [SV10] soit par des nouveaux concepts alternatifs [10].

Malgré cela, de faibles évolutions ont été constatées jusqu'au [5] qui présente un bootstrapping très rapide qui s'effectue aux alentours de 0.69 secondes. Ladite technique ouvre la voie à des applications avec des circuits plus complexes par une homomorphique une universelle NAND sur un bit avec une clé d'évaluation d'environ 1GBytes. Cette performance a été améliorée par [7], [8] et [9] en réduisant le temps d'exécution à 0.1 secondes avec une clé d'évaluation d'environ 1GBytes. Elle est implémentée dans une librairie dénommée TFHE.

En se basant sur cette librairie, le présent papier évalue les performances de différents circuits d'additions les plus connus dans des additions homomorphiques sur deux ou plusieurs nombres entiers de 16 bits.

2 NOTIONS PRÉLIMINAIRES

2.1 NOTATIONS

Les symboles et notations repris ci-dessous seront utilisés dans la suite de ce document:

- $\mathcal{B}$ l'ensemble de $\{0,1\}$;
- T le torus réel $\mathbb{R}/\mathbb{Z}$: la partie fractionnaire d'un nombre réel;
- $M^{(N)}[X]$ l'ensemble des polynômes sous un groupe abélien M modulo $X^N + 1$: $M[X]/X^N + 1$;
- M^n l'ensemble des vecteurs de (dimension) n des éléments de M;
- $et\ M^{n,m}$ l'ensemble des matrices de dimension $m \times n$ des éléments de M.

LE R-MODULE

Etant donné $(R, +, \times)$ un anneau commutatif. Un ensemble M est un R-module si $(M, +)$ est un groupe abélien, et s'il existe une opération externe. Bi-distributive et homogène.

A savoir, $\forall r, s \in R\ et\ x, y \in M,\ 1_R.x = x, (r+s).x = r.x + r.s, r.(x+y) = r.x + r.y, et\ (r \times s).x = r.(s.x)$

2.2 LA VERSION HOMOGÈNE DU PROBLÈME D'APPRENTISSAGE AVEC ERREURS (LWE)

Etant donné $n \geq 1$ un entier, $\alpha \in \mathbb{R}^+$ le paramètre bruit et s un secret uniformément distribué dans une certaine limite de $S \subset \mathbb{Z}^n$. Une distribution sur $T^n \times T$ est notée $D_{s,\alpha}^{LWE}$. Elle est obtenue en tirant le couple (a, b), où le membre de gauche a est choisi uniformément et aléatoirement dans T^n et le membre de droite est une évaluation de l'expression $b = as + e$. L'erreur e est tirée d'une distribution de Gauss de paramètre α.

- Problème de recherche: étant donné des échantillons LWE, trouver $s \in S$;
- Problème de décision: distinguer entre deux distributions des échantillons LWE et des échantillons uniformes et aléatoires tirées de $T^n \times T$.

2.3 LE PROBLÈME DIFFICILE D'APPRENTISSAGE AVEC ERREURS SUR UN TORUS (TLWE)

Soient $k \geq 1$ un entier, N une puissance de *2* et $\alpha \geq 0$ un paramètre bruit. Une clé secrète TLWE $s \in B_N[X]$ est un vecteur de k polynômes $\in \mathfrak{R} = \mathbb{Z}[X]/X^N + 1$ avec des coefficients binaires.

L'espace d'échantillon des messages est $T_N[X]$. Un échantillon TLWE fraiche de message $\mu \in T_N[X]$ avec le paramètre α sous la clé s est un élément $(a, b) \in T_N[X]^k \times T_N[X], b \in T_N[X]$ possédant une distribution de Gauss $D_{\alpha, s\alpha+\mu}$ autour de

$\alpha + s\mu$. L'échantillon est aléatoire si et seulement si son membre de gauche a appelé masque est uniformément aléatoire dans $T_N[X]^k$, trivial si a est fixé à *0*, moins bruitée si $\alpha = 0$, et homogene si et seulement si $\mu = 0$.

- Problème de recherche: étant donné plusieurs échantillons TLWE, trouver leurs clés ∈ $B_N[X]^k$;
- Problème de décision: distinction entre un échantillon TLWE homogène et aléatoire d'un échantillon uniforme et aléatoire de $T_N[X]^k$.

2.4 LA PHASE D'UN ÉCHANTILLON

Soient $c = (a, b) \in T_N[X]^k \times T_N[X]$ et $s \in B_N[X]^k$, la phase d'un échantillon est définie par l'expression $\varphi_s(c) = b - as$. Une phase est linéaire sur $T_N[X]^k$ et est $(kN + 1)-$lipschitzian pour la norme l_∞ si $\forall\, x, y \in T_N[X]^{k+1}$, $\|\varphi_s(x) - \varphi_s(y)\|_\infty \leq (kN + 1)\|x - y\|_\infty$

3 LE SCHÉMA DE CHIFFREMENT HOMOMORPHIQUE TFHE

Le GSW est un schéma de chiffrement homomorphique à niveau qui a été proposé par Gentry, Sahai et Waters dans [3] et a été amélioré dans [11]. Sa sécurité est basée sur le problème d'apprentissage avec erreurs (LWE).

3.1 TGSW

Le Torus GSW est une généralisation de la version invariante à l'échelle du GSW. Il est étendu aussi la fonction de décomposition aux polynômes. Cette approximation au seuil de la précision des paramètres induit une amélioration en temps d'exécution et prérequis mémoire pour un bruit additionnel.

3.1.1 FONCTION DE DÉCOMPOSITION

Soit $h \in \mathcal{M}_{d,k+1}(T_N[X])$ comme dans (1). $Dec_{h,\beta,\epsilon}(v)$ est un algorithme de décomposition sur h, avec la qualité β et la precision ϵ si et seulement si pour tout échantillon TLWE $v \subset T_N[X]^{k+1}$, sa sortie efficiente et publique donne un petit vecteur $u \in \Re^d$ tel que $\|u\|_\infty \leq \beta$ et $\|uh - v\|_\infty \leq \epsilon$. De plus, $u.h - v$ doit être 0 quand v est uniformément distribué dans $T_N[X]^{k+1}$.

$$\begin{pmatrix} 1/B_g & \cdots & 0 \\ \cdots & \cdots & \vdots \\ 1/B_g^l & & 0 \\ \vdots & \ddots & \vdots \\ 0 & & 1/B_g \\ \vdots & \cdots & \vdots \\ 0 & & 1/B_g^l \end{pmatrix} \qquad (1)$$

3.1.2 ECHANTILLON TGSW

Soit l et $k \geq 1$ deux entiers, $\alpha \geq 0$ le paramètre bruit et h la fonction de décomposition définie en (1). Soit $s \in B_N[X]^k$ une clé $RingLWE$. $C \in \mathcal{M}_{(k+1)l,k+1}(T_N[X])$ est un échantillon TGSW fraiche de $\mu \in \Re/h^\perp$ avec un paramètre bruit α si et seulement si $C = Z + \mu.h$ où chaque ligne de $Z \in \mathcal{M}_{(k+1)l,k+1}(T_N[X])$ est échantillon TLWE homogène de 0 avec un paramètre de gauss α.

De façon réciproque, un élément $C \in \mathcal{M}_{(k+1)l,k+1}(T_N[X])$ est un échantillon TGSW valide si et seulement si il existe un unique $\mu \in \Re/h^\perp$ et une unique clé s tel que chaque ligne de $C - uh$ est un échantillon TLWE valide 0 pour une clé s. Le polynôme μ est le message C, et noté par $msg\ (C)$.

3.1.3 PHASE ET ERREUR

Soit $A \in \mathcal{M}_{(k+1)l,k+1}(T_N[X])$ un échantillon TGSW pour une clé secrète $s \in B_N[X]^k$ par le paramètre $\alpha \geq 0$.

La phase de A, noté $\varphi_s(A) \in T_N[X]^{(k+1)l}$ est définie comme une liste de $(k + 1)l$ phases TLWE de chaque ligne de A.

Dans le même ordre d'idée, l'erreur de A, noté $err\ (A)$, est définie comme la liste de $(k+1)l$ erreurs TLWE de chaque ligne de A.

3.1.4 PRODUIT EXTERNE

Le produit externe $\boxdot$ est définie comme-suit:

$$\boxdot : TGSW\ \times TLWE\ \rightarrow TLWE$$
$$(\boxdot) \rightarrow A \boxdot b = \ Dec_{h,\beta,\epsilon}(b).A$$

THÉORÈME

Soit A un échantillon TGSW valide du message μ_A et soit b un échantillon TLWE du message μ_B alors $A \boxdot b$ est un échantillon TLWE du message $\mu_A.\mu_B$ et $\|err(\mathrm{A} \boxdot \mathrm{B})\|_\infty \leq\ (k+1)lN\beta\|err(A)\|_\infty + \|\mu_A\|_1(1+kN)\epsilon + \|\mu_A\|_1\|err(B)\|_\infty$ où $\beta\ et\ \epsilon$ sont les paramètres utilisés dans la fonction de décomposition $Dec_{h,\beta,\epsilon}(b)$. Si $\|err(\mathrm{A} \boxdot \mathrm{B})\|_\infty \leq \frac{1}{4}$ alors $\mathrm{A} \boxdot \mathrm{B}$ est échantillon TLWE valide.

3.1.5 LE PRODUIT INTERNE

Soit un produit $\boxed{\times}$: TGSW $\times$ TGSW $\rightarrow$ TGSW

$$(A,B) \rightarrow \mathrm{A} \boxed{\times} \mathrm{B} = \begin{bmatrix} A \boxdot b_1 \\ \vdots \\ A \boxdot b_{(k+1)l} \end{bmatrix} = \begin{bmatrix} Dec_{h,\beta,\epsilon}(b_1).A \\ \vdots \\ Dec_{h,\beta,\epsilon}\left(b_{(k+1)l}\right).A \end{bmatrix}$$

Avec A, B deux échantillons $TGSW$ valides respectivement des messages $\mu_A\ et\ \mu_B$ et b_i correspondant à la ième ligne de B. $\mathrm{A} \boxed{\times} \mathrm{B}$ est un échantillon TGSW valide du message $\mu_A.\mu_B$ et $\|err(\mathrm{A} \boxdot \mathrm{B})\|_\infty \leq\ (k+1)lN\beta\|err(A)\|_\infty + \|\mu_A\|_1(1+kN)\epsilon + \|\mu_A\|_1\|err(B)\|_\infty$ Si $\|err(\mathrm{A} \boxdot \mathrm{B})\|_\infty \leq \frac{1}{4}$ alors $\mathrm{A} \boxdot \mathrm{B}$ est échantillon TGSW valide.

3.1.6 BOOTSTRAPPING DANS LE TFHE

Le théorème 1 est utilisé pour accélérer le bootstrapping présenté dans [DM14]. Les optimisations effectuées ont réduit la taille de la clé de bootstrapping et éliminé le surplus de bruit dans les messages chiffrés.

Pour effectuer un bootstrapping, un échantillon $LWE\ (a,b)\ \in T^{n+1}$ est remis à l'échelle comme $\left(\bar{a},\bar{b}\right) mod\ 2N$ en utilisant des chiffrés de sa clé secrète $s \in \mathcal{B}^n$, les étapes suivantes doivent être suivies:

(1) Choisir un détecteur de phase $testv\ \in\ T_N[X]$ un polynôme fixé dont les coefficients sont initialisés à des valeurs que le bootstrapping doit retourner si $\varphi_s(a,b) = \frac{i}{2N}$;
(2) Encoder $testv$ dans un échantillon TLWE trivial;
(3) Ensuite, faire une rotation des coefficients en utilisant la multiplication externe avec des chiffrés TGSW des monômes cachés $X^{-s_i\bar{a}_i}.\ testv$ tourne d'une phase caché de (a,b);
(4) Enfin, extraire les termes constants comme un échantillon LWE.

3.1.6.1 EXTRACTION DE LWE À PARTIR DE TLWE

Extraire un échantillon LWE à partir d'un échantillon TLWE consiste à réécrire les polynômes dans leurs coefficients en ignorant les N-1 derniers coefficients de b. elle fournit un chiffré LWE de termes constants du message polynôme initial ou original.

DÉFINITION 1

Soit (a'',b'') un échantillon $TLWE_{s''}(\mu)$avec une clé $s'' \in \Re^k$, $KeyExtract(s'')$ est le vecteur d'entiers $s' = \left(coefs\left(s_1''(X)\right), \ldots\ldots\ldots\ldots, coefs\left(s_k''(X)\right)\right) \in \mathbb{Z}^{kN}$ et $Sampleextract(a'',b'')$l'échantillon $LWE(a',b') \in T^{kN+1}$ où $a' = \left(coefs\left(a_1''\left(\frac{1}{X}\right)\right), \ldots\ldots\ldots\ldots, coefs\left(a_k''\left(\frac{1}{X}\right)\right)\right)$ et $b' = b_0''$ le terme constant de b''.

Alors $\varphi_{s'}(a', b')$ (resp $msg(a', b')$) est égal au terme constant de $\varphi_{s''}(a'', b'')$ $(resp\ au\ terme\ constant\ de\ \mu = msg(a'', b''))$ et $\|Err(a', b')\|_\infty \leq \|Err(a'', b'')\|_\infty$ et $Var\big(Err(a', b')\big) \leq Var\big(Err(a'', b'')\big)$.

3.1.6.2 PROCÉDURE DE COMMUTATION DES CLÉS DANS UN ÉCHANTILLON LWE

Etant donné $LWE_{s'}$ un échantillon d'un message $\mu \in T$, la procédure de commutation des clés initialement proposé dans [9,6] donne en sortie un échantillon du même message μ sans accroissement du bruit. Cette procédure tolère l'approximation dabs ce schéma contrairement à son utilisation dans d'autres schémas.

DÉFINITION 2

Soit $s' \in \{0,1\}^{n'}$, $s \in \{0,1\}^n$, un parametre $\gamma \in \mathbb{R}$ et un paramètre de précision $t \in \mathbb{N}$, la clé de commutation $KS_{s'\to s,\gamma,t}$ est une séquence d'échantillons fraiches de LWE $KS_{i,j} \in LWE_{s,\gamma}(s'_i 2^{-j})$ pour $i \in [1, n']$ et $j \in [1, t]$.

ALGORITHME 2: Procédure de commutation de clé

Entrée: un échantillon LWE $\Big(a' = \big(a'_1, \dots\dots, a'_{n'}\big)\Big) \in LWE_{s'}(\mu)$, la clé de commutation $KS_{s'\to s}$ où $s' \in \{0,1\}^{n'}$, $s \in \{0,1\}^n$ et $t \in \mathbb{N}$ un paramètre de précision.

Sortie: un échantillon LWE $LWE_s(\mu)$.

(1) Initialiser $\vec{a}'_i$ un multiple proche de $\frac{1}{2^l}$ de a'_i, ainsi $|\vec{a}'_i - a'_i| < 2^{-(t+1)}$;
(2) Décomposer en binaire chaque $\vec{a}'_i = \sum_{j=1}^{t} a'_i 2^{-j}$ où $a'_{i,j} \in \{0,1\}$;
(3) Retourner $(a, b) = (0, b') - \sum_{i=1}^{n'} \sum_{j=1}^{t} a'_{i,j} KS_{i,j}$.

3.1.6.3 LA PROCÉDURE PROPREMENT DITE DE BOOTSTRAPPING

Etant donné un échantillon LWE $LWE_s(\mu) = (a, b)$, ladite procédure construit un chiffré de μ sous la même clé s mais avec un bruit fixe et faible. Comme dans [14], un échantillon TLWE est utilisé comme un chiffré intermédiaire pour effectuer une évaluation homomorphique de la phase, mais ici le produit externe du théorème 1 est utilisé avec un chiffré TGSW de la clé s.

DÉFINITION 3

Soit $s \in \mathcal{B}^n$, $s'' \in \mathcal{B}_N[X]^k$ et α un paramètre bruit. La clé de bootstrapping $BK_{s\to s'',\alpha}$ est définie comme une séquence de n échantillons TGSW où $BK_i \in TGSW_{s'',\alpha}(s_i)$.

ALGORITHME 3: Procédure de Bootstrapping

Entrée: un échantillon $LWE(a, b) \in LWE_{s,\eta}(\mu)$, une clé de bootstrapping $BK_{s\to s'',\alpha}$, une clé de commutation $KS_{s'\to s,\gamma}$ où $s' = KeyExtract(s'')$ et deux messages $\mu_0, \mu_1 \in T$.

Sortie: un échantillon LWE $LWE_s\Big(\mu_0\ si\ \varphi_s(a, b)\ \in\ \big]\frac{-1}{4}, \frac{1}{4}\big[\ sinon\ \mu_1\Big)$.

(1) Initialiser $\bar{\mu} = \frac{\mu_1 + \mu_0}{2}$ et $\bar{\mu}' = \mu_0 - \bar{\mu}$;
(2) Initialiser $\bar{b} = \lfloor 2Nb \rceil$ et $\bar{a}_i = \lfloor 2Na_i \rceil$ pour $i \in [1, n]$;
(3) Initialiser $testv \coloneqq (1 + X + \dots\dots\dots + X^{N-1}) \times X^{-2N/4}.\bar{\mu}' \in T_N[X]$
(4) $Acc \leftarrow \Big(X^{\bar{b}}.(o, testv)\Big) \in T_N[X]^{k+1}$
(5) $pour\ i\ de\ 1$ à n
(6) $Acc \leftarrow [h + (X^{-\bar{a}_i} - 1).BK_i] \boxdot Acc$
(7) Initialiser $\mu \coloneqq (0, \bar{\mu}) + SampleExtract(Acc)$
(8) Return $KeySwitch_{KS}(\mu)$.

3.2 LA LIBRAIRIE TFHE

La TFHE est une librairie libre de droit pour le chiffrement homomorphe complet et distribuée sous les termes de la licence Apache 2.0. Elle est écrite en C/C++ par l'implémentation d'un bootstrapping très rapide basé sur le [7], [8] et [9].

Elle évalue homomorphiquement 10 portes logiques (ET, OU, NONET, NONOU, … etc) aussi bien que la négation NON et La porte MUX. Chaque porte binaire prend environ 13 millisecondes qui améliorent le [5] par un facteur de 53, et la porte MUX prend environ 26CPU-millisecondes.

Le bootstrapping dans cette librairie n'impose pas de restriction sur le nombre de portes ou même sur la composition de circuit par rapport au [5] qui ne supporte des entrées similaires.

3.2.1 FONCTIONNALITÉS DE LA LIBRAIRIE TFHE

Elle est facile d'utilisation sur les circuits fabriqués manuellement et les circuits générés automatiquement par un utilitaire matériel ou logiciel.

Du point de vue de l'utilisateur, cette librairie peut:

(1) Générer un ensemble des clés secrètes et un ensemble des clés pour le cloud. L'ensemble des clés secrètes sont privées, et fournissent respectivement la capacité de chiffrement et déchiffrement. L'ensemble des clés pour le cloud peuvent être exportés vers le Cloud, et permettre d'effectuer des opérations sur les données chiffrés;
(2) Avec l'ensemble des clés secrètes, la librairie permet de chiffrer et de déchiffrer les données. Les données chiffrées peuvent être exportées en toute sécurité vers le cloud pour y effectuer des calculs sécurisés de façon homomorphique;
(3) Avec l'ensemble des clés cloud, la librairie peut évaluer une liste des portes binaires homomorphiquement à un taux de 76 portes par seconde et par cœur sans déchiffrer les entrées.

3.2.2 LES PROCESSEURS FFT

Pour s'exécuter le TFHE a besoin d'au moins un des processeurs repris dans le tableau ci-dessous:

Tableau 1. Les processeurs FFT

Nom	Licence	Langage et portabilité	Performance	Site web
Nayuki	MIT	C et AVX	1	www.nayuki.io
spqlios	Apache 2	AVX et FMA	1	
FFTW3	GPL	C et FORTRAN	2 - 3	www.fftw.org

En termes de performance, le processeur FFT est plus performant que les deux autres. Il réduit leurs temps d'exécution par un facteur de 2 ou 3.

4 LES OPÉRATIONS D'ADDITION HOMOMORPHES AVEC LE TFHE

L'espace de messages en clair dans le TFHE est $\mathbb{Z}_2$. Les opérations d'addition et de multiplication sont définies dans ledit schéma en utilisant respectivement les portes logiques XOR et AND. Ces portes sont la fondation de l'implémentation de circuits de plus en plus complexes.

Cette section présente une implémentation de l'addition arithmétique en composant l'additionneur binaire complet avec les portes AND et XOR. Le circuit additionneur a permis de construire quatre circuits d'addition qui sont l'additionneur à propagation de retenue (Ripple Carry Adder (RCA)), l'additionneur à retenue anticipee (carry Lookhead Adder (CLA)), additionneur à sauvegarde (Carry Save Adder (CSA)) et additionneur à retenue sélective (Carry Select Adder) et additionneur à arbre de wallace.

Cette opération arithmétique d'addition sera effectuée sur des entiers d'une taille de 16 bits.

4.1 ADDITIONNEUR

L'additionneur est d'un circuit qui est réalisé à partir deux circuits de base qui sont le demi-additionneur et l'additionneur complet. Ceux- servant à la réalisation de quatre architectures d'additionneurs cités ci-haut.

4.1.1 DEMI-ADDITIONNEUR

Le demi-additionneur est un circuit qui permet le calcul de la somme S et la retenue C lors de l'addition deux bits A et B.

$$S = a \oplus b \; et \; C = AB$$

4.1.2 ADDITIONNEUR COMPLET

Un additionneur complet est un circuit qui permet le calcul de la somme S et de la retenue C_{out} lors de l'addition de deux bits A, B et d'une retenue d'entrée C_{in}. Il comprend des demi-additionneurs et des additionneurs complets. La différence est qu'un demi-additionneur n'accepte pas une retenue tandis que l'additionneur l'accepte.

L'implémentation peut varier aussi longtemps que les expressions logiques de différents implémentations sont équivalentes. Dans [YG], pour exemple, les expressions de la somme et de la retenue peuvent écrites de la manière suivante:

$$c_{i+1} = a_i . b_i \oplus c_i . (a_i \oplus b_i)$$

$$s_i = a_i \oplus b_i \oplus c_i$$

Ou a_i et b_i sont le ième bit de deux sommations, c_i est la ième retenue, et s_i est la ième somme de bits. L'expression de la retenue peut être réduite comme-suit:

$$c_{i+1} = a_i . b_i \oplus c_i . (a_i \oplus b_i) = (a_i \oplus c_i) . (b_i \oplus c_i) \oplus c_i$$

Cette expression optimisée est trouvée dans [KSS]. Elle utilise seulement pour chaque bit une porte AND, et par conséquent un additionneur complet d'un bit à une profondeur multiplicative équivalent à 1 $(L = 1)$.

4.1.2.1 L'ADDITIONNEUR À PROPAGATION DE RETENUE

Les additionneurs à propagation de retenue (appelés Ripple Carry Adder) permettent de réaliser l'addition de deux nombres binaires de n bits, $A = (A_{n-1}, A_{n-2}, \ldots\ldots\ldots\ldots, A_0)$ et $B = (B_{n-1}, B_{n-2}, \ldots\ldots\ldots\ldots, B_0)$, et d'une retenue optionnelle C_{in}, assurant la propagation de la retenue. Le résultat est un nombre de n + 1 bits, consistant en un nombre $s = (S_{n-1}, s_{n-2}, \ldots\ldots\ldots\ldots, S_0)$ et d'une retenue de débordement C_{out}. Le résultat final est obtenu par l'attente de la propagation de la retenue à travers les n cellules d'additionneurs complets. Dans cette architecture, un additionneur constitue un étage et de ce fait, la retenue se propage de l'étage le moins significatif vers l'étage le plus significatif.

L'algorithme de l'additionneur à propagation de retenue de n bits est construit par $n - 1$ additionneur complet. Cet additionneur ajoute un bit à la fois de bits moins significatifs vers des bits plus significatifs. La profondeur multiplicative est $L = n - 1$, pour chaque bit à l'exception du plus significatif du bit, une porte AND est utile et chaque bit suivant dépend du bit précèdent.

ALGORITHME:

Entrée: deux entiers chiffrées de n bits a, b

Sortie: la somme s de n bits

$c_0 = 0$

$Pour \; i = 0$ à $n - 2$

$Faire$

$c_{i+1} = (a_i \oplus c_i) . (b_i \oplus c_i) \oplus c_i$

$s_i = a_i \oplus b_i \oplus c_i$

$fin \; faire$

End Pour

$s_{n-1} = a_{n-1} \oplus b_{n-1} \oplus c_{n-1}$

Retourner s

4.1.2.2 L'ADDITIONNEUR À RETENUE ANTICIPÉE

Dans une architecture à propagation de retenue, l'addition dépend de la propagation des retenues à travers des étages de l'additionneur parallèle. Pour réduire le délai de propagation et accélérer le processus d'addition, il est possible d'anticiper la retenue de sortie de chaque étage et de produire, à partir des entrées, la retenue par génération ou par propagation. Cette technique est appelée « anticipation de retenue ».

Une génération de retenue se produit lorsqu'une retenue est générée par l'additionneur complet. Une retenue ne peut avoir lieu que lorsque les deux bits d'entrée sont de 1. La retenue générée est noté G et équivaut à $G = AB$.

Une propagation de retenue est créée lorsqu'une retenue d'entrée est répercutée vers la retenue de sortie. Dans un additionneur complet, la propagation d'une retenue d'entrée peut avoir lieu lorsqu'au moins un des bits vaut 1. La retenue propagée notée P et équivaut à $P = A + B$.

La retenue de sortie d'un additionneur complet peut s'exprimer en tant que retenue propagée P ou en tant que retenue générée G. La retenue de sortie notée C_{sor} vaut 1 si la sortie générée vaut 1 ou si la sortie propagée vaut 1 et que la retenue d'entrée (C_{en}) vaut 1.

En d'autres termes, une retenue de sortie de 1 est générée par l'additionneur complet si $A = 1\ et\ B = 1$ ou par propagation de l'additionneur de la retenue d'entrée $(A = 1\ ou\ B = 1)\ et\ (C_{en} = 1)$. L'expression ci-dessous résume tous les cas: $C_{sor} = G + PC_{en}$.

Illustrons ce concept par une application à un additionneur parallèle de quatre bits. L'étage i produit une retenue de sortie soit par la génération de celui-ci soit par la propagation de la retenue interne vers le report de sortie. Pour chaque étage i, Il génère G_i et propage P_i de la manière suivante:

- La colonne i produit une retenue de sortie si les entrées A_i et B_i sont égales à un binaire: $G_i = A_i B_i$;
- La colonne i propage la retenue interne vers le report de sortie si une des entrées est égale à 1: $Pi_i = A_i + B_i$;
- La retenue de sortie de la colonne i est donnée par l'expression suivante:

$$C_i = A_i B_i + (A_i + B_i) C_{i-1} = G_i + P_i C_{i-1}.$$

L'algorithme de l'additionneur à anticipation de retenue peut-être décrite dans les étapes ci-dessous:

- Etape 1: calculer G_i et P_i pour toutes les colonnes;
- Etape 2: calculer les G et P pour chaque bloc de k −bits;
- Etape 3: le report d'entrée Cen se propage à travers le bloc de k −bits par les fonctions de génération et de propagation des retenues.

Exemple pour un bloc de 4 bits ($P_{3:0}$ et $G_{3:0}$):

$$G_{3:0} = G_3 + P_3 (G_2 + P_2 (G_1 + P_1 G_0)$$

$$P_{3:0} = P_3 P_2 P_1 P_0$$

De façon générale,

$$G_{i:j} = G_i + P_i (G_{i-1} + P_{i-1} (G_{i-2} + P_{i-2} G_j))$$

$$P_{i:j} = P_i P_{i-1} P_{i-2} P_j$$

$$C_i = G_{i:j} + P_{i:j} C_{j-1}$$

La complexité de l'algorithme de l'additionneur à retenue anticipée respectivement en temps est $O\ (n\ log\ n)$. L'additionneur à retenue anticipée de n est plus rapide que l'additionneur à propagation de retenue qui a respectivement une complexité en temps et espace de $O\ (n)$.

4.1.2.3 ADDITIONNEUR À SAUVEGARDE DE RETENUE

Les additionneurs à sauvegarde de retenue (CSA pour Carry Save Adder) assurent la fonction d'addition en traitant la retenue intermédiaire comme une sortie, et sans la propager au travers la cellule de suivante. La retenue de chaque étage est ainsi « sauvegardée ». Le résultat est composé de deux nombres de n bits: S pour la somme et C pour la retenue. L'architecture de cet additionneur est un arrangement linéaire d'additionneurs complets. Un calcul complémentaire doit être effectué pour obtenir le résultat.

L'additionneur à sauvegarde de retenue est un ensemble des k additionneurs complets mise en parallèle sans aucune connexion horizontale. La principale fonctionnalité de ce circuit est l'addition de trois nombres A, B et C pour produire deux nombres C′ et S tel que C′ + S = A+B+C.

Etant donné A=40, B=25 et C=20, C′ et S sont calculés de la manière suivante:

A	=	40	=		1	0	1	0	0	0
B	=	25	=		0	1	1	0	0	1
C	=	20	=		0	1	0	1	0	0
S	=	37	=		1	0	0	1	0	1
C′	=	48	=	0	1	1	0	0	0	

Le i^{eme} bit de la somme S_i et (i + 1) ^{eme} bit du report C′$_{i+1}$ est calculé en utilisant les expressions reprises ci – dessous:

$$S_i = A_i \oplus B_i \oplus C_i$$

$$c_{i+1} = a_i . b_i \oplus c_i . (a_i \oplus b_i) = (a_i \oplus c_i) . (b_i \oplus c_i) \oplus c_i$$

En d'autres termes, un circuit d'additionneur à sauvegarde de retenue est une cellule d'additionneur complet avec trois entrées de données au lieu de deux entrées plus la retenue précédente.

Et pour déterminer le résultat noté R de l'addition de trois nombres, les étapes ci-dessous sont exécutées:

- Pour l'étage de l'additionneur ayant la valeur la moins significative: $R_0 = S_0$;
- Pour l'étage de l'additionneur dont la valeur précède directement la moins significative, les expressions sont utilisées $R_1 = S_1 + C'_1$ et $C_2' = S_1 C'_1$;
- Pour les autres étages, les expressions de l'additionneur complet sont utilisées;
- Pour l'étage de l'additionneur ayant la valeur la plus significative: $R_{n-1} = C_{n-1} + C'_{n-1}$.

4.1.2.4 L'ADDITIONNEUR À RETENUE SÉLECTIVE

Un additionneur à retenue sélective (Carry Select Adder) est un circuit combinatoire logique et arithmétique qui additionne deux nombres de $N - bits$ et donne en sortie leurs sommes de $N - bits$ et un bit de retenue.

Elle offre une conception différente que celle d'un additionneur à propagation de retenue. Il ne propage pas la retenue à travers les additionneurs complets. Ainsi, le temps d'addition est réduit.

Il est un circuit composé de deux additionneurs à propagation en parallèle de $N - bits$ et d'un multiplexeur pour la sélection de la somme sortante. Pour exécuter une addition entre deux nombres de $N - bits$, deux additionneurs à propagation reçoivent à tous les étages respectivement une retenue entrante à 0 et une retenue entrante à 1, une fois que la retenue effective sera générée une simple sélection active la somme sortante adéquate.

4.1.2.5 ADDITIONNEUR À ARBRE DE WALLACE

Les additionneurs à arbre de wallace sont composés d'une structure arborescente d'additionneurs à sauvegarde de retenue, et d'un additionneur à propagation de retenue. Cette configuration d'être une architecture multi-opérandes très rapides. Les expressions ci-dessous donnent l'exemple d'addition de 4 opérandes notées A, B, C, et D:

Première étage:

$$S_{et10} = A_0 \oplus B_0 \oplus C_0$$
$$C_{et11} = (A_0 \oplus C_0)(B_0 \oplus C_0) \oplus C_0$$

Deuxième étage:

$$S_{et20} = S_{et10} \oplus C_{et10} \oplus D_0$$
$$C_{et21} = (S_{et10} \oplus D_0)(C_{et10} \oplus D_0) \oplus D_0$$

Troisième étage:

$$S_0 = S_{et20} \oplus C_{et20} \oplus C_{et30}$$
$$C_{et31} = (S_{et20} \oplus C_{et30})(C_{et20} \oplus C_{et30}) \oplus C_{et30}$$

5 RÉSULTAT EXPÉRIMENTAL

Cette section reporte les résultats expérimentaux de notre implémentation décrit au-dessus.

5.1 INITIALISATION DES PARAMÈTRES

Pour réaliser cette expérience, les paramètres de sécurité par défaut fournis avec librairie ont été utilisés sans aucune modification. La librairie fournit un API qui implémente la majorité de portes logiques les portes ci-dessous ont permis de bâtir l'architecture de différents additionneurs répertoriés ci-haut:

1. Function d'affectation homomorphique:
   void bootsCONSTANT (LweSample result, int value, const TFheGateBootstrappingCloudKeySet* bk);*
2. Function de recopie d'une variable dans une autre:
   void bootsCOPY (LweSample result, const LweSample* ca, const TFheGateBootstrappingCloudKeySet* bk);*
3. Fonction logique d'inversion d'une valeur booleenne:
   void bootsNOT (LweSample result, const LweSample* ca, const TFheGateBootstrappingCloudKeySet* bk);*
4. Fonction logique de multiplication sur deux bits:
   void bootsAND (LweSample result, const LweSample* ca, const LweSample* cb, const TFheGateBootstrappingCloudKeySet* bk);*
5. Fonction logique d'addition sur deux bits:
   void bootsXOR (LweSample result, const LweSample* ca, const LweSample* cb, const TFheGateBootstrappingCloudKeySet* bk);*

5.2 PERFORMANCE ET INTERPRÉTATION

Les implémentations ont été testées sur un ordinateur portable possédant trois environnements qui possède une mémoire **RAM de 4 Giga-octets**. Le tableau 2, la colonne représente le type de processeur utilisée pendant l'expérience et la ligne quand elle le type d'additionneur. L'intersection entre la ligne et la colonne représente la durée de l'exécution d'une opération d'addition respectivement sur deux nombres et quatre nombres de 16 bits.

Tableau 2. Performance des additionneurs

Durée (s)	AMD E1-2100 APU with Radeon ™ HD Graphics 1000 Mhz	Intel® Core™ i5-3210 CPU @ 2.50 Ghz	Intel ® Xeon ® CPU 5120 @ 1.86 Ghz (2)
RCA	131 (2)	109 (2)	55 (2)
CLA	109 (2)	90 (2)	46 (2)
CSA	**247 (3)**	**205 (3)**	**105 (3)**
CSSA	393 (2)	325 (2)	167 (2)

Du tableau 2, plus la capacité du processeur augmente plus le temps d'exécution est réduite. Le temps d'exécution de l'addition de deux nombres sur une architecture d'additionneur à propagation de retenue est réduit respectivement de 16% du processeur 1 au processeur 2, de 49% du processeur 2 au processeur 3 et de 58 % du processeur 1 au processeur 3.

La meilleure architecture en termes d'exécution de l'addition de deux nombres est l'additionneur à retenue anticipée. Elle améliore le temps d'exécution de l'addition sur deux nombres respectivement de 16 % en moyenne sur tous les processeurs de la technique à propagation de retenue et de 72 % de la technique à retenue sélective.

Tableau 3. Performance des additionneurs sur la somme de quatre nombres

Durée (s)	AMD E1-2100 APU with Radeon ™ HD Graphics 1000 Mhz	Intel® Core™ i5-3210 CPU @ 2.50 Ghz	Intel ® Xeon ® CPU 5120 @ 1.86 Ghz (2)
RCA	393 (4)	333 (4)	167 (4)
CLA	326 (4)	279 (4)	139 (4)
CSA	**378 (4)**	**322 (4)**	**161 (4)**
Wallace	**377 (4)**	**320 (4)**	**161 (4)**

L'architecture d'additionneur à retenue anticipé en pipeline pour l'addition de quatre nombres est meilleure que les architectures dédiées CSA et de Wallace. En effet, une réduction de 15% est constatée entre celle-ci et les architectures dédiées quel que soit le type de processeur. L'environnement multiprocesseur donne des résultats encourageant par rapport à l'environnement monoprocesseur avec une réduction du temps d'exécution de 50%.

6 CONCLUSION

De cette étude comparative, il ressort deux faits majeurs que sont la limitation matérielle a permis de constater que la meilleure architecture pour l'addition de plusieurs nombres chiffrés est l'additionneur à retenue anticipée et la performance de l'environnement multiprocesseur. En effet, quel que soit l'environnement utilisé mono ou multiprocesseur, l'architecture d'additionneur à retenue anticipée applique un coefficient de réduction au temps d'exécution par rapport aux autres architectures. De plus, l'environnement multiprocesseur a été un atout dans la mesure de l'addition sur les données chiffrées car il réduit de moitié le temps d'exécution d'une addition sur plusieurs nombres chiffrés. Bien que le temps d'exécution de cette opération se trouve encore dans l'ordre des minutes (environ 3 minutes). Il serait utile d'explorer la piste du parallèle par unité centrale de traitement ou par unité de traitement graphique pour plus de performance.

REFERENCES

[1] Chen, Y., Gong, G.: Integer arithmetic over ciphertext and homomorphic data aggregation. In: Proceedings of 2015 IEEE Conference on Communications and Network Security, pp. 628-632. IEEE, Piscataway, NJ (2015).

[2] Kolesnikov, V., Sadeghi, A.R. Scheinder, T.: Improved garbled circuit building blocks and application to auctions and computing minima. In: Garay, J.A., Miyaji, A, Otsuka, A. (eds) CANS 2009, LNCS, vol. 5888, pp. 1-20. Springer Berlin (2009).

[3] Craig Gentry, Amit Sahai, and Brent Waters. "Homomorphic Encryption from Learning with Errors: Conceptually-Simpler, Asymptotically-Faster, Attribute-Based". In: CRYPTO 2013, Part I. Ed. by Ran Canetti and Juan A. Garay. Vol. 8042. LNCS. Springer, Heidelberg, Aug. 2013, pp. 75–92. doi: 10.1007/978-3-642-40041-4_5.

[4] Craig Gentry. "A fully homomorphic encryption scheme". http://crypto.stanford.edu/craig. PhD thesis. Stanford University, 2009.

[5] Léo Ducas and Daniele Micciancio. "FHEW: Bootstrapping Homomorphic Encryption in Less Than a Second". In: EUROCRYPT 2015, Part I. Ed. by Elisabeth Oswald and Marc Fischlin. Vol. 9056. LNCS. Springer, Heidelberg, Apr. 2015, pp. 617–640. doi: 10.1007/978-3-662-46800-5_24.

[6] Marten van Dijk, Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. "Fully Homomorphic Encryption over the Integers". In: EUROCRYPT 2010. Ed. By Henri Gilbert. Vol. 6110. LNCS. Springer, Heidelberg, May 2010, pp. 24–43.

[7] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène. TFHE: Fast Fully Homomorphic Encryption Library over the Torus. https://github. com /tfhe/tfhe. 2016.

[8] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. "Faster Fully Homomorphic Encryption: Bootstrapping in Less Than 0.1 Seconds". In: ASIACRYPT 2016, Part I. Ed. by Jung Hee Cheon and Tsuyoshi Takagi. Vol. 10031. LNCS. Springer, Heidelberg, Dec. 2016, pp. 3–33. doi: 10.1007/978-3-662-53887-6_1.

[9] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. "Faster Packed Homomorphic Operations and Efficient Circuit Bootstrapping for TFHE". In: ASIACRYPT 2017, Part I. Ed. by Tsuyoshi Takagi and Thomas Peyrin. Vol. 10624. LNCS. Springer, Heidelberg, Dec. 2017, pp. 377– 408.

[10] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. "(Leveled) fully homomorphic encryption without bootstrapping". In: ITCS 2012. Ed. by Shafi Goldwasser. ACM, Jan. 2012, pp. 309–325.

[11] Jacob Alperin-Sheriff and Chris Peikert. "Faster Bootstrapping with Polynomial Error". In: CRYPTO 2014, Part I. Ed. by Juan A. Garay and Rosario Gennaro. Vol. 8616. LNCS. Springer, Heidelberg, Aug. 2014, pp. 297–314. doi: 10.1007/978-3-662-44371-2_17.

[12] R. L. Rivest, L. Adleman, and M. L. Dertouzos. "On Data Banks and Privacy Homomorphisms". In: Foundations of Secure Computation, Academia Press (1978), pp. 169–179.

[13] Oded Regev. "On lattices, learning with errors, random linear codes, and cryptography". In: 37th ACM STOC. Ed. by Harold N. Gabow and Ronald Fagin. ACM Press, May 2005, pp. 84–93.

[14] Binary Adder Architectures for Cell-Based VLSI and their Synthesis. PhD Thesis Swiss Federal Institute of Technology Zurich 1998.