Comparaison d'algorithmes de réduction modulaire en HLS sur FPGA

Libey Djath, Timo Zijlstra, Karim Bigou, Arnaud Tisserand

Compas'2019 : Parallélisme/ Architecture / Système
LIUPPA - IUT de Bayonne, France, du 24 au 28 juin 2019

Comparaison d'algorithmes de réduction modulaire en HLS sur FPGA

Libey Djath[1], Timo Zijlstra[2], Karim Bigou[1], et Arnaud Tisserand[2]

[1,2]Lab-STICC UMR 6285, [2]CNRS, [1]Univ. Bretagne Occidentale, Univ. Bretagne Sud

---

Résumé
Dans ce travail, nous comparons différents algorithmes de réduction modulaire implantés en synthèse de haut niveau sur FPGA pour des applications de cryptographie asymétrique. Nous étudions comment effectuer les réductions modulaires en fonction des tailles et formes (particulières/quelconques) des moduli, du type et du nombre des autres opérations arithmétiques impliquées. Pour cela, nous développons une bibliothèque C, qui sera distribuée sous licence libre, d'arithmétique modulaire pour la cryptographie asymétrique.

Mots-clés : arithmétique modulaire, conception en synthèse de haut niveau, exploration d'architectures et d'algorithmes, circuit FPGA.

---

1. Introduction

Les implantations en cryptographie asymétrique nécessitent un support d'*arithmétique modulaire* de plus en plus avancé. RSA utilise des carrés et multiplications modulo un nombre de quelques milliers de bits. Les cryptosystèmes actuels nécessitent des séquences d'opérations plus complexes [1] mais sur de plus petites tailles : des centaines de bits pour les courbes elliptiques (ECC) [8] et hyper-elliptiques (HECC) [3] ; ou des dizaines de bits pour la cryptographie post-quantique (PQC) sur des réseaux euclidiens (RE).
La représentation modulaire des nombres, ou RNS pour *residue number system* [7, 11, 2], amène un plus grand *parallélisme* interne permettant d'accélérer les calculs comme dans RSA et ECC, mais il nécessite un niveau supplémentaire de *réductions modulaires*. RNS découpe les nombres en petits morceaux dans une base de moduli premiers entre eux deux à deux notés m_i. Les calculs s'effectuent sur les restes modulo les m_i dans un *canal* propre à chaque m_i. Ces « petites » réductions modulo chaque m_i s'ajoutent aux réductions modulaires à un plus haut niveau sur les nombres complets (p. ex. le modulo p pour des éléments de $\mathrm{GF}(p)$). Dans ce papier, nous nous intéressons seulement aux réductions modulo les m_i dans le cadre de l'utilisation de RNS.
Dans le cadre PQC, on utilise des *petits corps finis* (p. ex. éléments entre 13 et 23 bits) mais pour des calculs sur des *polynômes* (p. ex. degrés entre 256 et 1024) et des *petites matrices* (souvent de tailles 2×2, 3×3 , 4×4).
Dans ce papier, nous nous intéressons uniquement aux réductions modulaires pour des tailles de quelques dizaines de bits (pour les moduli de RNS et pour les petits corps utilisés dans PQC-RE). Nous ne traitons pas des réductions pour des plus grands nombres comme ceux utilisés pour RSA et ECC (ceci fera l'objet de travaux futurs).

1. Addition, soustraction, carré, multiplication, multiplication par des constantes, inversion.

Les outils actuels de synthèse de circuits n'offrent pas de support très avancé pour la réduction modulaire. Ainsi, nous développons une *bibliothèque C* dédiée à l'*arithmétique modulaire* en cryptographie asymétrique utilisable en synthèse de haut niveau (HLS pour *high level synthesis*). Elle sera distribuée sous licence libre une fois suffisamment complétée, validée et documentée.
Nous utilisons la HLS pour *explorer de nombreux compromis* entre les représentations des nombres, les algorithmes de calcul et les architectures matérielles (ce qui est très coûteux en synthèse VHDL ou Verilog). Pour un modulo quelconque (c.-à-d. avec une écriture dense et sans structure comme $5101963 = (10011011101100110001011)_2$), on utilise principalement les réductions de Montgomery [9] et de Barrett [1]. Pour RE, on utilise plutôt un modulo spécifique avec une décomposition binaire très creuse (comme $8380417 = 2^{23} - 2^{13} + 1$) car la réduction se simplifie beaucoup (voir p. ex. [10]). En RNS, on utilise souvent des moduli de la forme $m_i = 2^w \pm c_i$ où w est la taille des mots dans les canaux et les c_i des petites constantes, denses, pour des questions de performances (voir p. ex. [4]). Ici aussi, cela conduit à des réductions plus efficaces que pour des moduli quelconques. En pratique, il y a un compromis entre la forme du modulo, ou des moduli, et le nombre de telles valeurs utilisables.
Dans ce papier, nous présentons les résultats de notre bibliothèque pour la *réduction modulaire de petits nombres* (c.-à-d. d'au plus quelques dizaines de bits, pour RNS ou PQC-RE). Nous comparons expérimentalement l'impact des principaux algorithmes de réduction pour différentes formes de moduli. Les principaux motifs de calcul de nos applications, utilisés dans ce papier, sont la somme réduite et la somme réduite de produits. Nous évaluons comment se comportent les outils de HLS sur ces motifs de calcul et expérimentons différentes techniques d'exploration pour différentes contraintes arithmétiques et architecturales.

2. Définitions et notations

Pour la suite du papier, nous définissons et utilisons :

- m le modulo de taille w bits pouvant avoir plusieurs formes :
  - MQ : *modulo quelconque* avec une écriture dense et sans structure particulière ;
  - MSC : *modulo spécifique à écriture binaire très creuse* (p. ex. 3 bits non nuls sur w) ;
  - MSR : *modulo spécifique pour RNS* de forme $2^w \pm c$ (où c est petit, $c < 2^{w/2}$) ;
- la réduction modulaire $x \bmod m$ ou des opérations modulaires $(x \diamond y) \bmod m$ avec l'opération $\diamond \in \{\pm, \times\}$;
- le motif de calcul : une séquence d'opérations arithmétiques faisant intervenir des réductions modulaires sur des vecteurs de taille N, les motifs étudiés ici sont :
  - M1 : $\sum_{i=1}^{N} x_i \bmod m$;
  - M2 : $\sum_{i=1}^{N} x_i \times y_i \bmod m$;
- deux stratégies sont comparées pour la réduction de séquence d'opérations modulaires :
  - RIS : Réduction Intermédiaire Systématique, p. ex. $\left(\sum_{i=1}^{N} (x_i \times y_i \bmod m)\right) \bmod m$;
  - RSF : Réduction Seulement à la Fin, p. ex. $\left(\sum_{i=1}^{N} x_i \times y_i\right) \bmod m$;
- l'opérande de la réduction modulaire peut avoir plusieurs tailles (p. ex. w, $w + \lceil \log_2 N \rceil$, $2w$ ou $2w + \lceil \log_2 N \rceil$ bits dans nos applications) ;
- des données x et y de taille w bits pour les vecteurs des motifs ;
- TM le temps total de calcul (en ns) pour obtenir le résultat d'un motif.

3. Bibliothèque développée

Nous souhaitons aider les concepteurs d'implantations matérielles en cryptographie asymétrique à explorer différents compromis algorithmes/ représentations des nombres/architectures de calcul. Dans ce papier, nous traitons uniquement de la réduction modulo m, un entier d'au plus quelques dizaines de bits, et de son utilisation dans quelques motifs de calcul typiques. Nous travaillons d'abord sur des m « petits » pour PQC et RNS, mais nous compléterons notre bibliothèque pour des moduli plus grands dans l'avenir (p. ex. éléments de $\mathrm{GF}(p)$ de quelques centaines de bits). La forme de m influençant beaucoup les algorithmes et les performances, nous avons choisi de supporter :

— des moduli quelconques (MQ), des moduli spécifiques très creux (MSC) utilisés pour PQC et des moduli spécifiques à RNS (MSR) ;
— les algorithmes de réduction de Montgomery [9] et de Barrett [1] pour MQ, et des algorithmes spécifiques pour MSC et MSR [4, 10].

Nous comparons nos résultats avec l'algorithme « natif » employé par Vivado HLS lors de l'utilisation de l'opérateur `%` du langage C. Nos résultats montrent qu'il s'agit probablement d'une division euclidienne itérative dont on conserve le reste final [6, 5]. En tout, nous comparons 5 algorithmes de réduction modulaire différents dans le même cadre.

Toutes nos implantations sont génériques pour des opérandes de taille w bits fixée à la conception. Pour RNS et PQC, w est de quelques dizaines de bits au plus. La taille N des vecteurs dans les motifs est aussi générique. Pour Barrett et Montgomery, il faut pré-calculer à la conception des constantes internes (p. ex. un inverse modulaire qui dépend de m et de w).

Nous définissons et utilisons des types de données spécifiques pour chaque taille nécessaire et des macros de transtypage (*cast*) pour adapter correctement les tailles [2]. Par exemple pour M2 en version RSF (c.-à-d. réduction seulement à la fin), il faut pouvoir réduire l'accumulation des N produits sur $2w + \lceil \log_2 N \rceil$ bits. Pour ce motif, nous avons aussi évalué une réduction intermédiaire systématique (RIS) à chaque itération.

La figure 1 présente un extrait de code C de notre bibliothèque et un exemple de code devant être produit par l'utilisateur. La partie haute de la figure est le code de la réduction modulaire avec l'algorithme de Barrett de la bibliothèque. Les types comme `word` et `sumdword` doivent être définis par l'utilisateur selon un « *template* » fourni avec la bibliothèque. Par exemple, le type `word` est de taille de w bits (code pour $w = 13$: `typedef uint13 word;`), le type `dword` est de taille double pour les produits de deux nombres de w bits (code pour $w = 13$: `typedef uint26 word;`), le type `sumdword` est pour les accumulateurs sur $2w + \lceil \log_2 N \rceil$ bits, etc. L'utilisateur doit aussi définir des macros de transtypage (*cast*) pour chacun des types définis. Par exemple, la macro `W` est définie par `#define W(x) ((word) (x))`. Le *template* fournit la liste de l'ensemble des macros à définir selon les usages. Toutes ces définitions spécifiques à l'application doivent être codées par l'utilisateur en complétant le *template* `parameters.h` inclus en ligne 1. Ce fichier définit aussi la constante N choisie pour les vecteurs.

La partie basse de la figure 1 présente un exemple de code utilisateur pour effectuer le motif M2 avec la stratégie RSF. L'utilisateur doit :

— tout d'abord inclure le fichier `parameters.h` en ligne 1 afin de « configurer » la bibliothèque pour son application ;
— spécifier ses entrées (ici 2 tableaux de N données de taille w bits en `word`) et ses sorties (ici la somme réduite donc aussi en `word`) en ligne 4 ;

2. On rappelle que la sémantique du langage C est assez peu mathématique, p. ex. le produit de 2 mots est un mot de même taille que les opérandes.

Code (de la bibliothèque) pour la réduction modulaire avec l'algorithme de Barrett :

```
1  #include "parameters.h"
2  #include "arithmod_internal.h"
3
4  word barrett(sumdword x)
5  {
6      sumword x1 = SUM_W(x>>width);
7      sumword q = SUM_W((RSW(x1) * RSW(R_const))>>(shift - width));
8      word x0 = W(x);
9      counter c = 0;
10     if (x0 > M) c = 2;
11     else if (x0 != 0) c = 1;
12     q = q + c;
13     sumdword z = SUM_DW(q) * SUM_DW(m);
14     signword res = x - z;
15     if(res<0) res = res + M;
16     if(res<0) res = res + M;
17     return W(res);
18 }
```

Code (utilisateur) pour le motif M2 RSF avec réduction de Barrett :

```
1  #include "parameters.h"
2  #include "arithmod.h"
3
4  word m2_rsf(word A[N], word B[N])
5  {
6      sumdword res=0;
7      acc: for(counter i=0; i<N; i++)
8          res += DW(A[i]) * DW(B[i]);
9      return barrett(res);
10 }
```

FIGURE 1 – Extraits de codes C de notre bibliothèque (haut) et de son utilisation (bas).

— initialiser l'accumulateur avec la bonne taille en ligne 6 ;
— effectuer son calcul (ici la somme des produits selon la stratégie RSF, donc sans réduction intermédiaire) en lignes 7 et 8 ;
— et enfin effectuer la réduction en appelant la fonction de réduction choisie (ici `barrett(res)` en ligne 9).

L'utilisation est assez simple, puisqu'une fois les spécifications de l'application définies (préparation du fichier `parameters.h` à partir d'un *template* fourni), il suffit de faire quelques appels de fonctions, utiliser les bons types et éventuellement quelques *casts* pour s'assurer de la bonne taille des données intermédiaires. Les étiquettes comme `acc` sur la boucle d'accumulation en ligne 7 du code de la fonction `m2_rsf` sont utilisées pour spécifier les cibles des directives d'optimisations particulières de l'outil HLS (pipeline, déroulage de boucle, etc.).

Nos implantations ont été réalisées avec Vivado HLS 2017.4 [14] sur un FPGA Artix-7 (xc7a15) de Xilinx. Les résultats présentés ci-dessous ont comme contraintes : une taille de modulo de $w \in \{13, 17, 23, 30\}$ bits, une taille de vecteurs $N \in \{10, 20, 40, 100\}$, une période d'horloge cible de 3 ns et un effort d'optimisation par défaut (moyen). Nous avons aussi exploré l'impact de

directives de pipeline et de déroulage de boucle de l'outil (voir [12]). Nous obtenons ainsi plus de 2400 résultats d'implantations différentes. Nous présentons ci-dessous un sous-ensemble représentatif pour des raisons de place (d'autres résultats pour d'autres contraintes seront disponibles dans la documentation de la bibliothèque). Notre attention portera principalement sur le motif M2, car il est crucial pour RNS et PQC.

4. Résultats d'implantation

4.1. Comparaison des différents algorithmes de réduction

Nous commençons par comparer les différents algorithmes de réduction modulaire (le `%` de Vivado et nos implantations de Barrett, Montgomery, MSC et MSR) pour différentes tailles w de modulo dans le cas du motif M2 en version RSF avec $N = 20$. La figure 2 présente les résultats pour le meilleur temps de calcul obtenu pour les différentes combinaisons de directives d'optimisations testées. Pour chaque métrique (temps, surface et compromis surface $\times$ temps), les résultats sont normalisés par rapport à la plus grande valeur pour la métrique.

Nous observons que l'algorithme natif utilisé par l'outil HLS lors de l'appel de l'opérateur `%` est bien moins performant que nos implantations des algorithmes de la littérature. Il n'utilise pas de bloc DSP pour la réduction (uniquement pour l'accumulation des produits $\sum_{i=1}^{N} x_i \times y_i$), mais il nécessite de nombreux cycles de calcul. Il semble utiliser une boucle avec un nombre d'itérations dépendant de la taille de l'opérande dans nos expérimentations. Il présente donc peu d'intérêt pour nos applications cryptographiques car nos implantations sont toutes bien meilleures.

Les meilleurs résultats obtenus sont clairement pour MSR et MSC. Ceci est parfaitement logique puisque les algorithmes pour les moduli spécifiques utilisent des propriétés particulières permettant de simplifier les calculs (ce qui n'est pas possible pour des MQ). Ainsi, pour des applications RNS, l'algorithme MSR est plus performant que Barrett et Montgomery. Pour PQC, la même conclusion s'impose pour MSC.

Le nombre de cycles d'horloge de nos 4 implantations (Barrett, Montgomery, MSC et MSR) sont proches (et souvent autour de 50% de moins que pour `%`). Les fréquences obtenues sont proches de la contrainte imposée à l'outil.

Des résultats similaires sont obtenus pour les autres tailles N de vecteurs testées. Notre bibliothèque permet à l'utilisateur d'utiliser le meilleur algorithme de réduction modulaire en fonction du type de modulo utilisé dans son application. Dans le cas d'applications avec des moduli spécifiques, l'utilisation des algorithmes MSC et MSR offre de bien meilleurs résultats. Dans le cas d'applications avec des moduli quelconques, notre bibliothèque confirme que Montgomery est un peu meilleur que Barrett.

Dans la suite, nous ne donnerons plus les résultats pour l'algorithme de réduction natif avec l'opérateur `%` car il est bien trop lent pour nos applications cryptographiques.

4.2. Impact des directives d'optimisation sur la boucle d'accumulation

Nous avons testé plusieurs directives d'optimisation sur la boucle d'accumulation : `pipeline` correspond au cas où les N itérations sont pipelinées ; `unroll`k au cas où (N/k) itérations sont effectuées sur k opérateurs en parallèle.

La table 1 présente les résultats d'implantation du motif M2 RSF avec $N = 20$ pour la réduction spécifique de moduli MSR de taille $w = 23$ bits. Nous comparons différentes directives de l'outil de synthèse HLS sur la boucle d'accumulation. La première ligne du tableau présente comme référence les résultats sans aucune directive. Chaque produit de la boucle d'accumulation requiert 2 blocs DSP car ils ont 2 opérandes de $w = 23$ bits, ce qui est plus grand que la

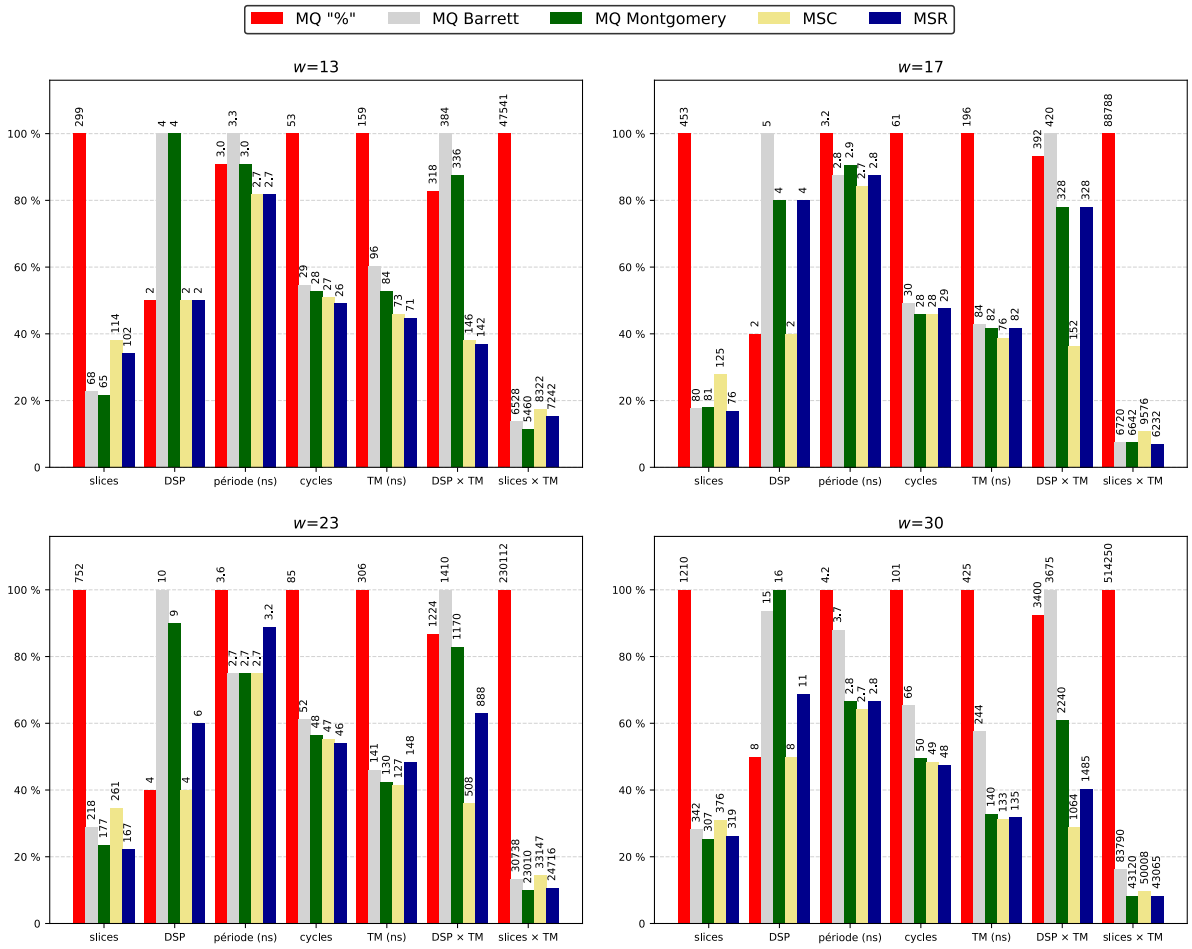

FIGURE 2 – Comparaison des différents algorithmes de réduction pour $w \in \{13, 17, 23, 30\}$ bits pour le motif M2 RSF avec $N = 20$.

directives	surface		temps (ns, cycles)			surface × temps	
	slices	DSP	période	cycles	TM	DSP × TM	slices × TM
aucune	136	4	3.1	216	670	2680	91120
`pipeline`	142	4	3.2	64	205	820	29110
`pipeline` + `unroll2`	167	6	3.2	46	148	888	24716
`pipeline` + `unroll4`	228	10	3.3	37	123	1230	28044
`pipeline` + `unroll10`	526	22	3.1	39	121	2662	63646

TABLE 1 – Impact des directives d'optimisation pour M2 RSF, MSR, $w = 23$ et $N = 20$.

taille du multiplieur disponible dans un bloc DSP du FPGA cible (voir [14, 13]). La réduction finale requiert également 2 blocs DSP.
On remarque que les différentes implantations atteignent quasiment la période cible. En pipelinant, on arrive à diviser par 3 le temps de calcul TM sans changer le nombre de DSP utilisés. On peut encore réduire de $28\,\%$ et $40\,\%$ TM en déroulant avec un facteur 2 et 4 respectivement, au prix d'une augmentation du nombre de DSP utilisés. Au-delà d'un facteur 4, on n'observe plus d'amélioration significative du temps, tout en payant le prix d'une augmentation du nombre de multiplieurs DSP utilisés. Ceci est très certainement dû au nombre croissant d'accès mémoires simultanés nécessaires à l'exploitation du parallélisme.
Enfin, avec une métrique de coût global de type surface×temps, nous observons que les directives `pipeline` et `pipeline` + `unroll2` se démarquent assez nettement des autres. Un niveau de parallélisme modéré semble donc facilement exploitable dans le contexte de notre bibliothèque. Pour pouvoir exploiter plus de parallélisme interne, il nous faudrait probablement changer l'algorithme et la structure du code.

4.3. Impact de la stratégie de réduction

Pour un motif nécessitant de nombreuses opérations modulo m, plusieurs *stratégies* de réduction sont envisageables. Il est possible d'effectuer une réduction intermédiaire systématique (RIS) à chaque itération ; pour M2 cela correspond au calcul :

$$\left(\sum_{i=1}^{N} (x_i \times y_i \bmod m)\right) \bmod m.$$

Il est possible d'effectuer une réduction seulement à la fin (RSF) ; pour M2 cela donne :

$$\left(\sum_{i=1}^{N} x_i \times y_i\right) \bmod m.$$

Dans la version RSF de motifs comme M1 et M2, l'accumulateur doit être plus large pour absorber les retenues de la somme des résultats de chaque itération (avec $\lceil \log_2 N \rceil$ bits en plus). Ceci engendre une réduction finale plus coûteuse car son opérande est plus large.
Nous avons implanté les stratégies RIS et RSF pour les motifs M1 et M2 et nos différents algorithmes de réduction. La table 2 présente une partie représentative de nos résultats.
Pour M1, nous donnons uniquement les résultats pour RSF car RIS est toujours bien moins performant (le coût d'une itération de boucle, une addition, est bien trop faible devant celui d'une réduction modulaire). Nous présentons les résultats de M1 RSF pour montrer l'impact de l'itération (par rapport à M2).

motif	algorithme et stratégie	surface		temps (ns, cycles)			surface×temps	
		slices	DSP	période	cycles	TM	DSP×TM	slices×TM
M1	Montgomery RSF	122	5	2.6	31	81	405	9882
	Barrett RSF	110	1	2.9	58	169	169	18502
	MSC RSF	62	0	2.5	17	43	0	2635
	MSR RSF	66	0	2.6	17	45	0	2970
M2	Montgomery RIS	194	12	2.6	60	156	1872	30264
	Montgomery RSF	149	7	2.6	64	167	1165	24794
	Barrett RIS	259	12	2.8	53	149	1781	38436
	Barrett RSF	218	10	2.7	52	141	1404	30608
	MSC RIS	403	4	2.7	55	149	594	59846
	MSC RSF	261	4	2.7	47	127	508	33121
	MSR RIS	146	8	2.6	31	81	645	11768
	MSR RSF	167	6	3.2	46	148	884	24583

TABLE 2 – Impact des stratégies de réduction pour $w = 23$ et $N = 20$.

Pour M2 avec Barrett et Montgomery, RSF est meilleur en compromis surface×temps (RIS peut être un tout petit plus rapide mais pour une surface plus importante).
Globalement RSF est souvent plus efficace que RIS (pour les N et w testés). Mais il nous reste à explorer ce qui se passe pour des N très grands et des w petits (comme c'est le cas pour RE avec p. ex. $w = 13$ et $N \in [256, 1024]$).
D'autres stratégies intermédiaires sont envisageables, comme réduire de temps en temps de petits accumulateurs partiels. Cela pourrait être intéressant pour des motifs où les calculs dans chaque itération sont plus complexes qu'une seule opération.

4.4. Impact de la taille N des vecteurs

Enfin, nous analysons l'impact de la taille des vecteurs des motifs sur les performances et les coûts. La figure 3 présente quelques résultats représentatifs pour l'algorithme de réduction MSR avec $w = 23$. Nous observons que pour les N testés, le compromis surface×temps est proche d'une fonction affine de N (résultat général à toutes nos implantations). Cette croissance affine avec N permet d'effectuer des estimations à haut niveau très simplement (avec une marge d'erreur raisonnable).
Cependant, la figure 3 suggère aussi un phénomène bien plus complexe à anticiper (du moins dans l'état actuel de nos connaissances). Ceci est en lien avec les deux dernières lignes de la table 2 où le motif RIS est plus efficace que RSF. On observe sur la figure 3 que le temps de calcul et le nombre de cycles du RIS croissent moins vite que ceux de RSF. Le même phénomène est observé pour les compromis surface×temps, on peut donc en déduire que le gain de RIS sur RSF se fait sur le coût de chaque itération de la boucle d'accumulation. Cependant, d'un point de vue purement arithmétique, RIS calcule $s \leftarrow (s + x_i \times y_i) \bmod m$ à chaque itération, contre $s \leftarrow (s + x_i \times y_i)$ pour RSF, donc RIS effectue plus de calculs par itération. Nous voyons que l'outil arrive à mieux pipeliner les itérations de RIS que celles de RSF dans certains cas. Il est clair qu'il nous reste du travail pour mieux cerner quand et comment (modification de la structure du code) utiliser les différentes directives d'optimisation de l'outil HLS.

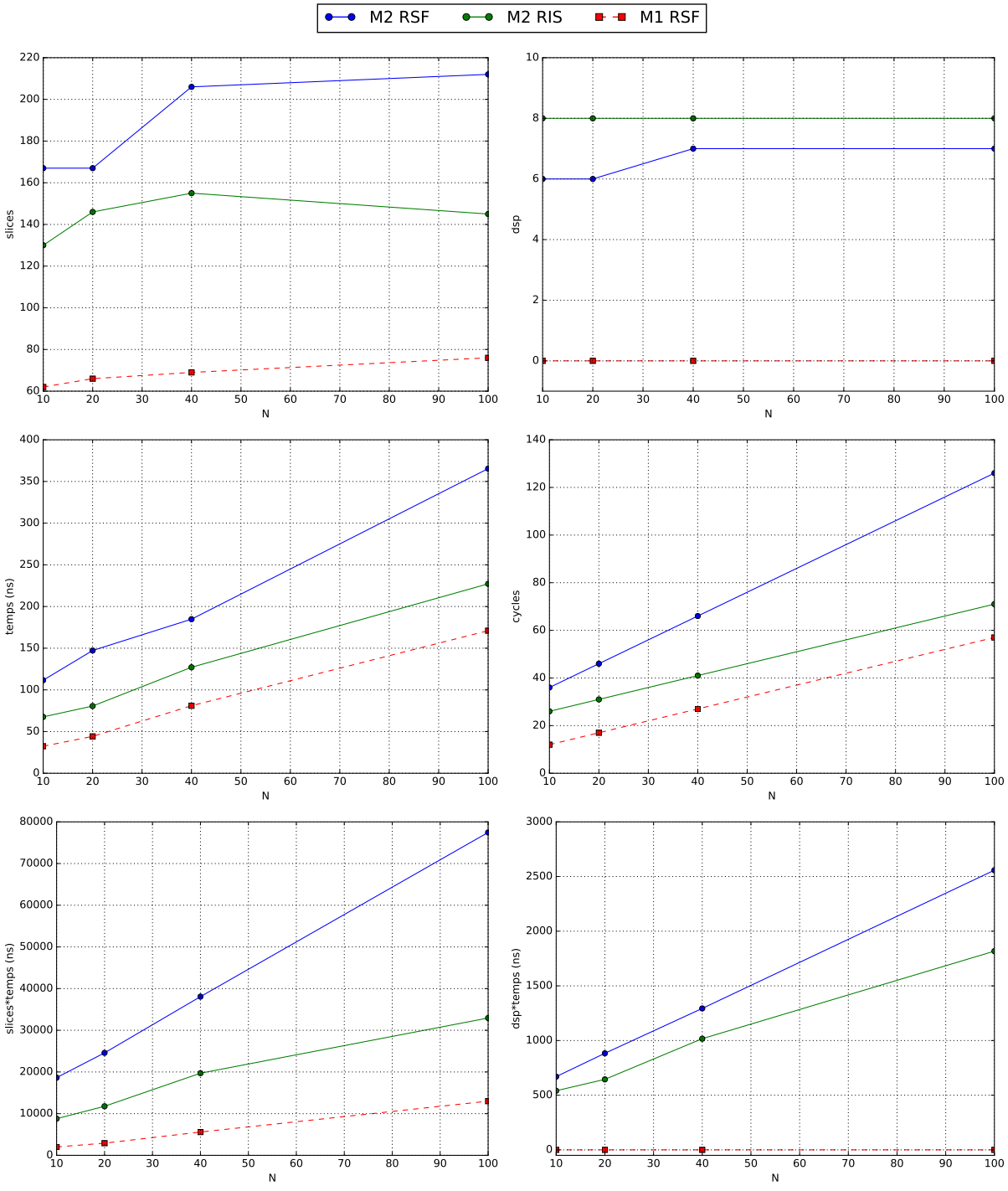

FIGURE 3 – Impact de la taille N des vecteurs pour MSR avec $w = 23$.

5. Conclusion

Notre bibliothèque offre un support, assez simple d'utilisation, d'algorithmes de réduction modulaire avancés qui ne sont pas supportés nativement par les outils de HLS actuels. Elle offre aussi la possibilité de générer des circuits optimisés en temps et en surface pour chaque type de modulo (de forme quelconque ou spécifique). Ceci est particulièrement intéressant pour des applications en cryptographie asymétrique comme ECC en RNS ou PQC.
Nous allons continuer le développement de notre bibliothèque et ajouter d'autres opérations, formes de moduli et motifs de calcul. Nous souhaitons aussi essayer d'autres outils de HLS et fabricants/familles de FPGA.

Remerciements

Ce travail a été financé en partie par le PEC, la DGA et la Région Bretagne. Nous remercions chaleureusement les relecteurs anonymes pour leurs précieuses remarques et corrections.

Bibliographie

1. Barrett (P.). – *Communications Authentication and Security Using Public Key Encryption : A Design for Implementation*. – Thèse de PhD, University of Oxford, 1984.
2. Bigou (K.). – *Étude théorique et implantation matérielle d'unités de calcul en représentation modulaire des nombres pour la cryptographie sur courbes elliptiques*. – Lannion, France, Thèse de PhD, University Rennes 1, novembre 2014.
3. Cohen (H.) et Frey (G.) (édité par). – *Handbook of Elliptic and Hyperelliptic Curve Cryptography*. – Chapman & Hall/CRC, 2005.
4. Crandall (R. E.). – Method and apparatus for public key exchange in a cryptographic system. – US Patent 5159632 A, octobre 1992.
5. Ercegovac (M. D.) et Lang (T.). – *Division and Square-Root Algorithms : Digit-Recurrence Algorithms and Implementations*. – Kluwer Academic, 1994.
6. Ercegovac (M. D.) et Lang (T.). – *Digital Arithmetic*. – Morgan Kaufmann, 2003.
7. Garner (H. L.). – The residue number system. *IRE Transactions on Electronic Computers*, vol. EC-8, n2, juin 1959, pp. 140–147.
8. Hankerson (D.), Menezes (A.) et Vanstone (S.). – *Guide to Elliptic Curve Cryptography*. – Springer, 2004.
9. Montgomery (P. L.). – Modular multiplication without trial division. *Mathematics of Computation*, vol. 44, n170, avril 1985, pp. 519–521.
10. Solinas (J. A.). – *Generalized Mersenne Numbers*. – Technical Repport nCORR-99-39, University of Waterloo, Centre for Applied Cryptographic Research, 1999.
11. Szabo (N. S.) et Tanaka (R. I.). – *Residue arithmetic and its applications to computer technology*. – McGraw-Hill, 1967.
12. Xilinx. – *Vivado HLS Optimization Methodology Guide (UG1270)*. – Rapport technique, décembre 2017.
13. Xilinx. – *7 Series DSP48E1 Slice User Guide (UG479)*. – Rapport technique, mars 2018.
14. Xilinx. – *Vivado Design Suite User Guide High-Level Synthesis (UG902)*. – Rapport technique, février 2018.