

Thèse

présentée au Laboratoire d'Informatique de Robotique
et de Microélectronique de Montpellier pour
obtenir le diplôme de doctorat

Spécialité : Informatique
Formation Doctorale : Informatique
École Doctorale : Information, Structures, Systèmes

Opérateurs arithmétiques parallèles pour la cryptographie asymétrique

par

Thomas Izard

Soutenue le 19 Décembre 2011, devant le jury composé de :

Directeur de thèse

LAURENT IMBERT, Chargé de recherche

CNRS, LIRMM, Montpellier

Co-Directeur de thèse

PASCAL GIORGI, Maître de conférences

Université Montpellier 2

Rapporteurs

JEAN-GUILLAUME DUMAS, Maître de conférences

Université Joseph Fourier, Grenoble

PIERRICK GAUDRY, Directeur de recherche

CNRS, LORIA, Nancy

Président du jury

JEAN-CLAUDE BAJARD, Professeur

Université Pierre et Marie Curie, Paris

Examineurs

PHILIPPE LANGLOIS, Professeur

Université de Perpignan Via Domitia

BRUNO ROUZEYRE, Professeur

Université Montpellier 2

Résumé

Les protocoles de cryptographie asymétrique nécessitent des calculs arithmétiques dans différentes structures mathématiques. Un grand nombre de ces protocoles requièrent en particulier des calculs dans des structures finies, rendant indispensable une arithmétique modulaire efficace. Ces opérations modulaires sont composées d'opérations multiprécision entre opérandes de tailles suffisamment grandes pour garantir le niveau de sécurité requis (de plusieurs centaines à plusieurs milliers de bits). Enfin, certains protocoles nécessitent des opérations arithmétiques dans le groupe des points d'une courbe elliptique, opérations elles-mêmes composées d'opérations dans le corps de définition de la courbe. Les tailles de clés utilisées par les protocoles rendent ainsi les opérations arithmétiques coûteuses en temps de calcul. Par ailleurs, les architectures grand public actuelles embarquent plusieurs unités de calcul, réparties sur les processeurs et éventuellement sur les cartes graphiques. Ces ressources sont aujourd'hui facilement exploitables grâce à des interfaces de programmation parallèle comme OpenMP ou CUDA. Cette thèse s'articule autour de la définition d'opérateurs arithmétiques parallèles permettant de tirer parti de l'ensemble des ressources de calcul, en particulier sur des architectures multicœur à mémoire partagée. La parallélisation au niveau arithmétique le plus bas permet des gains modérés en termes temps de calcul, car les tailles des opérandes ne sont pas suffisamment importantes pour que l'intensité arithmétique des calculs masque les latences dues au parallélisme. Nous proposons donc des algorithmes permettant une parallélisation aux niveaux arithmétiques supérieurs : algorithmes parallèles pour la multiplication modulaire et pour la multiplication scalaire sur les courbes elliptiques. Pour la multiplication modulaire, nous étudions en particulier plusieurs ordonnancements des calculs au niveau de l'arithmétique modulaire et proposons également une parallélisation à deux niveaux : modulaire et multiprécision. Ce parallélisme à plus gros grain permet en pratique des gains plus conséquents en temps de calcul. Nous proposons également une parallélisation sur processeur graphique d'opérations modulaires et d'opérations dans le groupe des points d'une courbe elliptique. Enfin, nous présentons une méthode pour optimiser la multiplication scalaire sur les courbes elliptiques pour de petits scalaires.

Mots clefs : *Arithmétique multiprécision - Arithmétique modulaire - Arithmétique des courbes elliptiques - Parallélisme - Calcul haute-performance*

Abstract

Protocols for asymmetric cryptography require arithmetic computations in several mathematical structures. In particular, many of them need computations in finite structures, imposing an efficient modular arithmetic. These modular calculations are composed of multiprecision operations between operands of sizes large enough to insure the required level of security (between several hundred and several thousand of bits). Finally, some protocols need arithmetic operations in the group of points of an elliptic curve, which are themselves composed of modular computations. The sizes of the keys used by the protocols make the arithmetic computations expansive in terms of execution time. Nowadays, current architectures have several computing units, which are distributed over the processors and GPU. These resources are now easily programmable using dedicated languages such as OpenMP or CUDA. This thesis focuses on the definition of parallel algorithms to take advantage of all the computing resources of multi-core shared-memory architectures. Parallelism at the lowest arithmetic level gives a moderate speedup since the sizes of the operands are not large enough so that the arithmetic intensity hides the latencies induced by the parallelism. We propose algorithms for parallelization at higher arithmetic: parallel algorithms for modular multiplication and scalar multiplication on elliptic curves. For modular multiplication, we study in particular several schedulings of modular computations. We also propose a two-level parallelization, at modular and multiprecision levels. This "coarse-grained" parallelism allows in practice a more substantial speedup. We also present a parallelization of modular and elliptic curves operations on GPU. Finally, we introduce a method to optimize scalar multiplication on elliptic curves for small scalars.

Keywords: *Multiprecision arithmetic - Modular arithmetic - Elliptic curves arithmetic - Parallelism - High-performance Computing*

REMERCIEMENTS

Je tiens avant tout à remercier profondément mes deux directeurs de thèse, Pascal Giorgi et Laurent Imbert, pour leur présence, leurs conseils, leur enthousiasme, leur implication et leur soutien tout au long de ces trois années de thèse.

Je remercie vivement Jean-Guillaume Dumas et Pierrick Gaudry qui ont accepté de relire ce manuscrit. Je tiens également à remercier Jean-Claude Bajard, Philippe Langlois et Bruno Rouzeyre, qui ont accepté de faire parti de mon jury en tant qu'examineurs.

Je remercie les membres présents et passés de l'équipe ARITH du LIRMM pour leur accueil. Je n'oublie pas les personnels administratifs du LIRMM, de l'université Montpellier 2 et de l'école doctorale I2S.

Je remercie sincèrement ma famille et mes amis, en particulier mes parents, mon frère et ma sœur pour m'avoir soutenu et encouragé durant ces trois années.

Introduction	1
1 Introduction aux architectures parallèles	5
1.1 Introduction au parallélisme	5
1.2 Modèles de parallélisme	6
1.3 Architecture SMP à mémoire partagée	7
1.3.1 Ordonnancement	8
1.3.2 Implémentation de codes parallèles dans le modèle synchrone	8
1.4 Processeurs graphiques	10
1.4.1 Les cartes Tesla	12
1.4.1.1 Architecture	12
1.4.1.2 CUDA	12
1.4.1.3 Exécution d'une grille	13
1.4.1.4 Mémoires	13
1.4.1.5 Optimisations et bonnes pratiques	14
1.4.2 Un exemple concret : les requêtes par fredonnement sur GPU	15
1.4.2.1 Requête par fredonnement	15
1.4.2.2 Alignement de séquences	16
1.4.2.3 Implémentation sur GPU	16
1.4.2.4 Résultats expérimentaux	18
2 Arithmétique pour la cryptographie	21
2.1 La cryptographie à clé publique	21
2.2 Arithmétique multiprécision	24
2.2.1 Représentations des entiers multiprécision	24
2.2.2 Addition et soustraction	24
2.2.3 Multiplication	24
2.2.3.1 Version scolaire	25
2.2.3.2 Karatsuba	25
2.2.3.3 Toom-3	26
2.2.3.4 Toom-Cook r	27
2.2.3.5 Algorithmes à complexité quasi-linéaire	28
2.2.4 Division	28
2.2.5 Résumé des complexités	29
2.2.6 L'arithmétique multiprécision en pratique	29
2.3 Arithmétique modulaire	30
2.3.1 Addition et soustraction	31
2.3.2 Multiplication	31
2.3.2.1 L'algorithme classique	31

2.3.2.2	Multiplication de Blakely	31
2.3.2.3	Multiplication de Montgomery	32
2.3.2.4	Méthode FIOS	33
2.3.2.5	Multiplication modulaire de Barrett	33
2.3.2.6	Multiplication modulaire bipartite	35
2.4	Arithmétique des courbes elliptiques	35
2.4.1	La cryptographie basée sur les courbes elliptiques	35
2.4.2	Définition des courbes elliptiques	36
2.4.3	Projections et équations de courbes	37
2.4.4	La multiplication scalaire	38
2.4.4.1	Algorithmes à fenêtre : représentation en base 2^w	39
2.4.4.2	Non-adjacent form	40
2.4.4.3	Double-Base number system	41
3	Arithmétiques parallèles	43
3.1	Parallélisme sur architecture SMP à mémoire partagée	44
3.1.1	Arithmétique entière	44
3.1.1.1	Addition et soustraction entières	44
3.1.1.2	Ensemble d'additions	44
3.1.1.3	Multiplication entière	45
3.1.1.4	Résultats expérimentaux	45
3.1.2	Arithmétique modulaire	46
3.1.3	Courbes elliptiques	47
3.1.3.1	Approche naïve	48
3.1.3.2	Répartition homogène des calculs	49
3.1.3.3	Résultats expérimentaux	50
3.2	Parallélisme sur les processeurs graphiques	53
3.2.1	Représentation des entiers multiprécision	53
3.2.2	Gestion de la mémoire	54
3.2.3	Arithmétique modulaire	56
3.2.3.1	Addition et soustraction	56
3.2.3.2	Multiplication	56
3.2.3.3	Résultat expérimentaux	56
3.2.4	Arithmétique des courbes elliptiques	57
3.2.4.1	Courbes elliptiques sur GPU	57
3.2.4.2	Addition et doublement dans $E(\mathbb{F}_p)$	58
3.2.4.3	Multiplication scalaire	59
3.3	Conclusion et perspectives	61
3.3.1	Architecture SMP	61
3.3.2	Processeurs graphiques	62
4	La multiplication modulaire multipartite	63
4.1	Réductions partielles	64
4.1.1	Réduction partielle de Montgomery	64
4.1.2	Réduction partielle de Barrett	65
4.2	Définition de la multiplication multipartite	67
4.2.1	La multiplication bipartite	67
4.2.2	La multiplication quadripartite	67
4.2.3	Généralisation	68
4.2.4	Analyse de complexité	70
4.3	Complexité parallèle de la multiplication multipartite	72
4.4	Ordonnancement des calculs pour l'approche modulaire	72
4.4.1	Regroupement des produits médians	73
4.4.2	Diminuer le nombre de réductions modulaires	73
4.4.3	Méthodes heuristiques pour l'ordonnancement des calculs	76
4.4.3.1	Ordonnancement dans le modèle quadratique	76

4.4.3.2	Ordonnancement dans le modèle linéaire	77
4.4.4	Conclusion	78
4.5	Modèle réaliste : un parallélisme à deux niveaux	79
4.6	Résumé des complexités parallèles de la multiplication modulaire	80
4.7	Résultats expérimentaux	81
4.7.1	Multiplication modulaire parallèle	81
4.7.2	Exponentiation parallèle modulaire	82
4.8	Conclusion et perspectives	88
4.8.1	Un carré multipartite	88
4.8.2	Une réduction multipartite	89
5	Optimisation des formules pour la multiplication scalaire	91
5.1	Formules	92
5.1.1	Définition	92
5.1.2	Représentation des formules	94
5.1.3	Le problème de l'optimisation	94
5.2	Une approche pour l'optimisation automatique	96
5.2.1	Élimination des sous-expressions communes	96
5.2.2	Transformations arithmétiques	98
5.2.3	Straight-Line programs	98
5.3	Une fonction Hash heuristique	99
5.3.1	Le problème <i>Ensemble Computation</i>	102
5.4	Implémentation et résultats	102
5.4.1	Implémentation	102
5.4.2	Formule de quintuplement	102
5.4.3	Cas spéciaux	105
5.4.4	Production de code	105
5.5	Conclusion et perspectives	107
	Conclusion	109
A	PACE : Prototyping Arithmetic for Cryptography Easily	111
A.1	Entiers multiprécision	111
A.2	Arithmétique modulaire	111
A.3	Courbes elliptiques	113
A.4	Algorithmes de multiplication scalaire ou d'exponentiation modulaire	113
B	Exemple de code spécialisé : multipartite $k = 4$ sur 4 processeurs	115
	Bibliographie	119

TABLE DES FIGURES

1	Hiérarchie d'arithmétiques pour les protocoles classiques de cryptographie asymétrique	2
2	Parallélisations à différents niveaux arithmétiques	2
3	Parallélisations des protocoles utilisant les courbes elliptiques	3
1.1	Deux modèles de parallélisme	7
1.2	Schéma simplifié d'une architecture SMP à deux processeurs et huit cœurs	7
1.3	Architecture d'un nœud de la plate-forme HPC@LR (deux processeurs Intel Xeon X5650)	9
1.4	Comparaison CPU-GPU	11
1.5	Architecture des GPU Tesla (Nvidia)	12
1.6	Schéma d'exécution du code CUDA	13
1.7	Accès coalescents à la mémoire globale	15
1.8	Calcul de la case $M_{i,j}$ de l'algorithme Smith-Waterman	17
2.1	Accélération de la multiplication GMP par rapport à la multiplication quadratique	30
2.2	$E(\mathbb{R}) : y^2 = x^3 - 18x + 25$	37
2.3	$E(\mathbb{F}_{23}) : y^2 = x^3 + 5x + 2$	37
2.4	Addition de deux points sur \mathbb{R}	37
2.5	Doublement de point sur \mathbb{R}	37
3.1	Comparaison des performances entre différentes implémentations de la multiplication entière utilisant un schéma de parallélisation quadratique et la multiplication séquentielle quadratique	46
3.2	Comparaison des performances entre les opérateurs de multiplications utilisant un schéma de parallélisation quadratique basés sur les algorithmes de multiplication optimisés de GMP	47
3.3	Découpage non homogène de k	49
3.4	Multiplication scalaire parallèle avec pré-calcul	51
3.5	Multiplication scalaire parallèle sans pré-calcul	52
3.6	Organisation des données sur le CPU et sur le GPU	54
3.7	Structure pour la représentation des entiers multiprécision en registre avec un accès indexable	55
3.8	Opérateur de comparaison pour les entiers stockés en registres	55
3.9	Temps de calculs pour l'addition et la multiplication en fonction des mémoires utilisées	57
3.10	Accélération de la version GPU par rapport à la version CPU	58
3.11	Temps de calcul pour l'addition et le doublement dans $E(\mathbb{F}_p)$ en utilisant LM , SM et R .	59
3.12	Accélération de SM par rapport à la version CPU PACE et mpFq	60
3.13	Comparaison des débits en $[k]P/s$ des versions CPU et GPU	60
4.1	Réductions de Montgomery	65
4.2	Réductions de Barrett	66
4.3	Multiplication multipartite avec $k = 5$	69

4.4	Diminution du nombre de réductions par regroupement des calculs de même poids pour $k = 3$	74
4.5	Comparaison des performances des multiplications modulaires de Barrett, de Montgomery, bipartite et multipartite	83
4.5	Comparaison des performances des multiplications modulaires de Barrett, de Montgomery, bipartite et multipartite	84
4.6	Comparaison des performances d'une exponentiation modulaire <i>Square-and-multiply</i> basée sur nos implémentations parallèles des multiplications de Barrett, de Montgomery, bipartite et multipartite	86
4.6	Comparaison des performances d'une exponentiation modulaire <i>Square-and-multiply</i> basée sur nos implémentations parallèles des multiplications de Barrett, de Montgomery, bipartite et multipartite	87
4.7	Réduction multipartite avec $k = 3$: trois réductions pour les $n/2$ mots de poids fort de c , trois réductions pour les $n/2$ mots de poids faible	89
5.1	Arbre de la formule de doublement en coordonnées jacobiennes	95
5.2	DAG de l'arbre 5.1	97
A.1	Hierarchie de la bibliothèque PACE	112

Autrefois réservée à des applications militaires ou diplomatiques, la cryptographie est de nos jours tellement courante que nous n'y prêtons guère attention. Acheter en ligne, payer par carte bleue, téléphoner, échanger des courriers électroniques, voire ouvrir ou démarrer sa voiture... derrière tous ces gestes quotidiens se cachent des protocoles de sécurité, parfois simples à déjouer, souvent invulnérables.

C'est cette invulnérabilité qui est au cœur des enjeux de la cryptologie. Avec la multiplication des échanges, la cryptographie historique qui consistait à chiffrer un message avec un secret connu uniquement des deux parties communicantes a trouvé sa limite : comment échanger les secrets de manière fiable, en particulier à l'heure actuelle où des milliards d'humains cherchent à communiquer ?

La résolution de ce problème par Diffie et Hellman en 1976 a ouvert la voie aux protocoles de cryptographie asymétrique. En utilisant des fonctions mathématiques bien choisies, les deux parties souhaitant communiquer fabriquent un secret commun connu d'eux seuls.

L'invulnérabilité de ces protocoles repose sur une ou plusieurs opérations dans des structures mathématiques particulières, qui sont généralement elles-mêmes constituées d'opérations dans d'autres structures arithmétiques. Cette hiérarchie de niveaux arithmétiques pour les protocoles classiques est représentée par la figure 1.

Des cryptosystèmes comme RSA ou El Gamal sont composés d'opérations arithmétiques modulaires, elles mêmes composées d'opérations d'arithmétique entière. Pour garantir un niveau de sécurité suffisant, les opérands utilisés doivent être de grande taille et ces opérations ne peuvent pas être traitées directement par le processeur. Elles sont donc composées d'appels aux instructions atomiques de l'unité de calcul.

Les protocoles de cryptographie basés sur les courbes elliptiques sont aujourd'hui de plus en plus utilisés, car ils permettent des niveaux de sécurité identiques à RSA ou El Gamal mais avec des paramètres de taille réduite. Ces cryptosystèmes nécessitent une nouvelle couche arithmétique : celle des opérations entre points d'une courbe elliptique.

Les performances en termes de temps de calcul de ces protocoles dépendent de l'efficacité des algorithmes utilisés à chaque niveau arithmétique. On dispose généralement de plusieurs algorithmes pour réaliser une même opération. L'efficacité de ces algorithmes varie en fonction d'un certain nombre de paramètres, comme la taille des opérands, l'utilisation de pré-calcul, etc. Mais elle dépend également de la puissance des processeurs employés pour les opérations atomiques.

Les architectures grand public actuelles disposent d'un certain nombre d'unités de calcul réparties sur un ou plusieurs processeurs et sur d'éventuelles cartes graphiques. Les capacités de calcul des plates-formes sont décuplées et peuvent traiter simultanément un certain nombre de processus.

Le parallélisme est ainsi devenu un enjeu majeur pour l'accélération des calculs scientifiques. La cryptanalyse en particulier bénéficie d'architectures contenant de grandes ressources de calcul : cela va de la grille de calcul pour la factorisation d'un module RSA de 768 bits [54, 55] à la carte graphique pour la méthode de factorisation basée sur les courbes elliptiques [7].

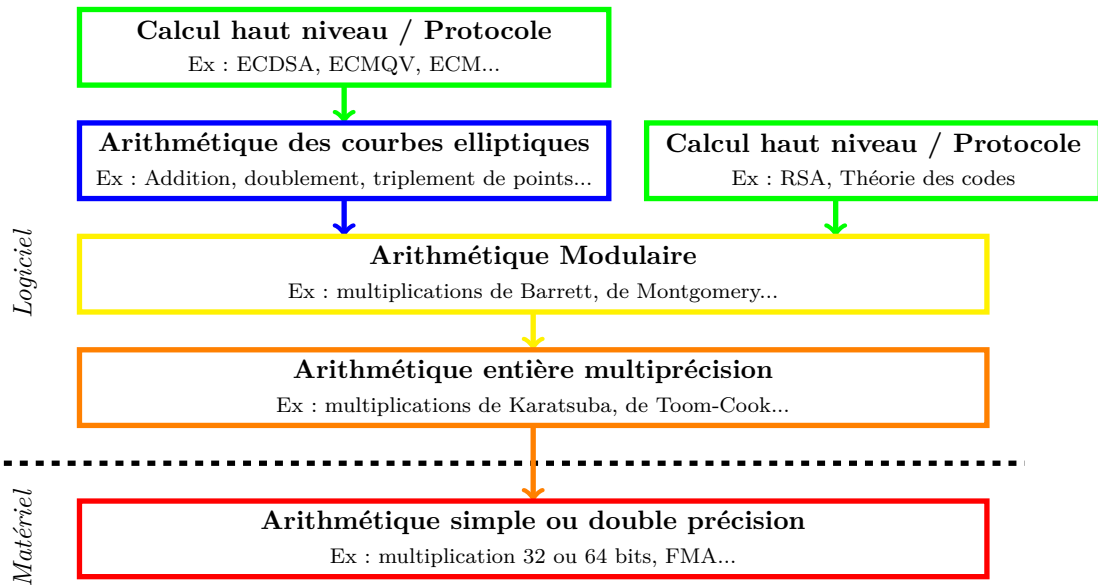


FIGURE 1 – Hiérarchie d'arithmétiques pour les protocoles classiques de cryptographie asymétrique

Se posent alors un certain nombre de questions naturelles, auxquelles nous avons tenté de répondre durant cette thèse :

Les opérations arithmétiques nécessaires aux protocoles cryptographiques peuvent-elles tirer parti de cette multiplicité de ressources de calcul ?

Avec comme question sous-jacente comment paralléliser des calculs arithmétiques déjà performants séquentiellement. Certains opérateurs arithmétiques se prêtent facilement à la parallélisation, comme par exemple la version scolaire de la multiplication entière multiprécision. Dans cette opération, on réalise un certain nombre de produits indépendants, qui peuvent donc être traités en parallèle. En revanche, les algorithmes d'exponentiation modulaire sont séquentiels, car ils opèrent bit-à-bit sur la représentation binaire de l'exposant, ce qui implique une forte dépendance dans les calculs.

À quel(s) niveau(x) arithmétique(s) doit-on paralléliser ?

Les figures 2 et 3 présentent différentes parallélisations possibles des arithmétiques pour les protocoles cryptographiques. On peut paralléliser des instances de l'application (figure 3b) ou au contraire paralléliser les niveaux les plus bas (figure 2a).

Quelles ressources de calcul donneront les meilleures performances ?

Les processeurs disposent d'un nombre limité d'unités de calcul, généralement cadencées à plusieurs Giga Hertz. À l'inverse, les cartes graphiques disposent d'un grand nombre d'unités de calcul, mais de fréquence plus faible. Quels seront les meilleures architectures pour les applications visées ?

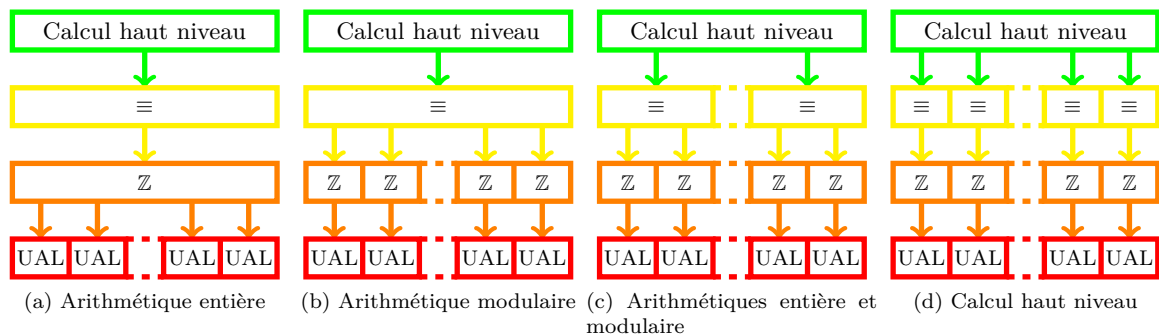


FIGURE 2 – Parallélisations à différents niveaux arithmétiques

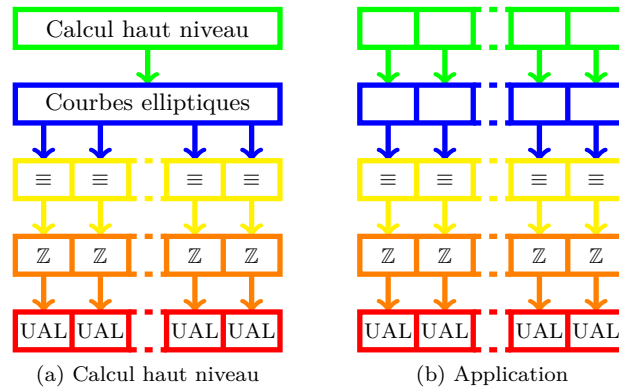


FIGURE 3 – Parallélisations des protocoles utilisant les courbes elliptiques

Pour répondre à ces questions, nous avons défini des opérateurs parallèles pour l’arithmétique entière, l’arithmétique modulaire et l’arithmétique des courbes elliptiques sur architecture multicœur à mémoire partagée et sur processeur graphique.

Dans le premier chapitre, nous introduisons le parallélisme et les deux types d’architecture que nous utilisons. Nous présentons également un travail annexe à cette thèse qui a consisté à utiliser les cartes graphiques pour accélérer une application de recherche de similarités dans des bases de données.

Dans le second chapitre, nous détaillons les différentes arithmétiques présentées précédemment : arithmétique des courbes, qui permet de définir le problème sur lequel repose la sécurité du protocole, arithmétique modulaire, car nos courbes reposent sur des structures mathématiques finies et enfin arithmétique multiprécision, essentielle car pour des tailles de données communément traitées par les processeurs (de 32 à 128 bits), les algorithmes de cryptanalyse sont capables de casser les protocoles en temps humain.

Dans le troisième chapitre, nous présentons nos implémentations et nos premiers résultats pour la parallélisation d’opérateurs arithmétiques sur architecture multicœur ainsi que le lancement d’opérations modulaires parallèle sur les processeurs graphiques. Dans le premier cas, nous nous intéressons plus particulièrement à la parallélisation de l’arithmétique entière (figure 2a) ainsi qu’à la parallélisation de la multiplication scalaire sur les courbes elliptiques (figure 3a). Dans le second cas, nous nous intéressons à la parallélisation d’instances d’opérations modulaires (figure 2d) et d’instances d’opérations sur les courbes elliptiques (figure 3b).

Nous présentons dans le quatrième chapitre une multiplication modulaire totalement parallèle qui permet de découper le calcul en un nombre arbitraire de tâches (figure 2b). En utilisant les opérateurs parallèles d’arithmétique entière, nous pourrions alors proposer une multiplication modulaire utilisant un double niveau de parallélisme (figure 2c).

Le cinquième chapitre est consacré à la multiplication scalaire sur les courbes elliptiques, l’opération de base de tous les protocoles reposant sur ces objets mathématiques. Nous verrons comment définir les meilleures multiplications possibles pour de petits scalaires, et en quoi ces optimisations permettront d’accélérer les temps de calcul.

CHAPITRE 1

INTRODUCTION AUX ARCHITECTURES PARALLÈLES

Sommaire

1.1	Introduction au parallélisme	5
1.2	Modèles de parallélisme	6
1.3	Architecture SMP à mémoire partagée	7
1.3.1	Ordonnancement	8
1.3.2	Implémentation de codes parallèles dans le modèle synchrone	8
1.4	Processeurs graphiques	10
1.4.1	Les cartes Tesla	12
1.4.2	Un exemple concret : les requêtes par fredonnement sur GPU	15

L'objectif de cette thèse est de proposer des opérateurs arithmétiques parallèles et efficaces pour la cryptographie asymétrique. Dans ce premier chapitre, nous introduisons la notion de parallélisme et les deux architectures sur lesquelles nous travaillerons : les architectures SMP à mémoire partagée et les processeurs graphiques.

1.1 Introduction au parallélisme

Si le parallélisme existe depuis un demi-siècle, il apparaît dans les architectures grand public en 2005 avec les premiers processeurs multicœurs, qui bénéficient ainsi de 50 ans de recherches dans ce domaine.

Le parallélisme a pour but d'augmenter la vitesse d'applications en multipliant les ressources de calcul d'un système. Pour en bénéficier, l'application doit être pensée en termes de tâches indépendantes qui pourront être exécutées de manière concurrente par les différentes unités de calcul.

Définition 1 (Tâche). Une tâche est une partie d'une application exécutée de manière indépendante par une unité de calcul. Elle a son propre espace de données et a accès à l'ensemble des mémoires du système.

On peut supposer en théorie qu'en multipliant le nombre d'unités de calcul par M les temps de calcul sont divisés par M . Cette vision néglige deux points essentiels : le passage à l'échelle (*scalabilité*) de l'application sur l'architecture parallèle et le surcoût (*overhead*) lié au parallélisme, c'est-à-dire le coût pratique lié aux seules instructions permettant de lancer, synchroniser, stopper, etc. des processus concurrents.

La loi d’Amdahl [3] permet d’exprimer le gain potentiel d’une application lancée sur un système multiprocesseur à M cœurs en considérant une proportion d’activité parallélisable s (sans considérer l’*overhead*) par rapport à l’application séquentielle.

Théorème 1 (Loi d’Amdahl). $Gain(s, M) = \frac{1}{(1-s) + \frac{s}{M}}$

La partie $(1 - s)$ correspond à la proportion de l’application qui ne peut pas être parallélisée. Il est clair que cette proportion doit être la plus petite possible pour offrir les meilleurs gains de performances. Si cette loi a été corrigée ou critiquée au fil des ans, elle n’en illustre pas moins les limites du parallélisme par rapport au séquentiel. Ainsi, le gain attendu en termes d’efficacité pour une application parallèle ne dépend pas uniquement du nombre d’unités de calcul disponibles. On trouvera dans [44] une première approche de transposition de cette loi aux architectures parallèles modernes.

Le surcoût dû au parallélisme est lui lié à la fois à l’architecture et au système d’exploitation. Si ses causes peuvent être établies, elles sont difficiles à contourner sans maîtrise totale de la plateforme. Une proportion non négligeable provient des instructions propres au parallélisme : lancement des tâches parallèles (instructions système `clone()` ou `fork()` sur un système Unix) mais surtout des synchronisations nécessaires entre ces différentes tâches (instruction `wait()`). Dans la conception d’une application parallèle, il est donc primordial de minimiser les dépendances entre tâches. Une mauvaise utilisation de la mémoire peut également causer des surcoûts importants. Nous définissons par la suite les bonnes pratiques à adopter sur les deux architectures considérées.

En 1966, Flynn propose une classification des architectures selon quatre grands types [32] :

- SISD (*Single instruction simple data*) : une instruction agit sur une donnée. Ce type d’architecture correspond à une machine de von Neumann classique.
- SIMD (*Single instruction multiple data*) : la même instruction agit sur différentes données. C’est une architecture qui exploite un parallélisme de données, à l’instar des cartes graphiques.
- MISD (*Multiple instructions single data*) : plusieurs instructions agissent sur une même donnée. C’est le cas des architectures de type *pipeline*.
- MIMD (*Multiple instructions multiple data*) : plusieurs instructions agissent sur plusieurs données. Ce type d’architecture se divise en deux catégories : les architectures à mémoire distribuée et les architectures à mémoire partagée. Dans le premier cas chaque processeur a sa propre zone mémoire et aucune connaissance sur les mémoires des autres processeurs. Les unités de calcul communiquent *via* des systèmes de messagerie comme MPI (Message Passing Interface). Dans le second cas, que nous détaillons plus bas, les processeurs partagent la mémoire centrale *via* le bus système.

1.2 Modèles de parallélisme

Le parallélisme de données, réalisé par les architectures SIMD, consiste à faire appliquer par un ensemble de tâches la même série d’instructions sur des données différentes. Sur architecture MIMD, on peut distinguer deux grands types de parallélisme *de tâches*. Le premier modèle, qui a été utilisé dans nos travaux sur architecture SMP à mémoire partagée, est un modèle de parallélisme synchrone [100] (figure 1.1a) : toutes les tâches sont globalement séquentielisées : les lancements, synchronisations et fusions sont communes.

Le second modèle de parallélisme consiste à lancer, synchroniser ou fusionner les tâches localement et dynamiquement (figure 1.1b). Les tâches sont représentées par une structure de DAG (Directed Acyclic Graph) qui permet de hiérarchiser les différentes tâches en fonction des tâches antérieures qu’elles nécessitent. Ce parallélisme asynchrone peut être réalisé par différents langages, comme Cilk [14], NESL [13], KAAPI [38] ou encore JADE [79]. Ces langages se différencient en fonction des algorithmes d’ordonnancement ou d’accès à la mémoire qu’ils utilisent [75]. Par exemple, Cilk et KAAPI utilisent des techniques de *vol de travail* (work-stealing [15]) et NESL un parallélisme imbriqué : les tâches lancent les sous-tâches qu’elles peuvent paralléliser.

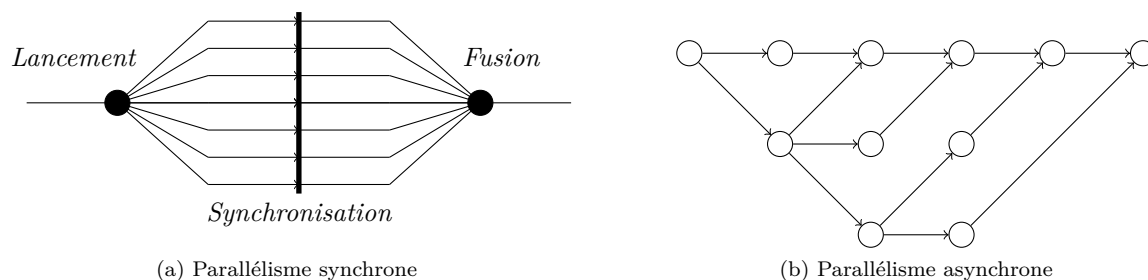


FIGURE 1.1 – Deux modèles de parallélisme

Dans la suite de cet état de l'art, nous considérons uniquement le modèle de parallélisation synchrone. L'utilisation du parallélisme asynchrone est explicité dans les différentes perspectives de nos travaux.

1.3 Architecture SMP à mémoire partagée

L'architecture SMP (pour Symmetric MultiProcessing) consiste à multiplier les unités de calcul au sein d'une même plate-forme. Tous les processeurs se partagent un unique BUS système, et donc l'accès à la mémoire centrale et aux différents périphériques.

Dans les systèmes multicœurs qui équipent aujourd'hui la plupart des machines grand public, les processeurs regroupent plusieurs unités de calcul, nommées *cœurs*. Généralement, l'architecture SMP considère chacun de ces cœurs comme un processeur à part entière. Ce double niveau de parallélisme (plusieurs processeurs possédant chacun plusieurs cœurs) doit cependant être pris en compte pour développer des applications performantes.

En effet, comme nous l'illustrons par la figure 1.2, la communication entre les différents processeurs passe par la mémoire centrale et le BUS système, ce qui peut causer des latences importantes. À l'inverse, la communication entre les cœurs d'un même processeur est plus efficace car la plupart des processeurs modernes embarquent des *cache* mémoire partagés par les différentes unités de calcul.

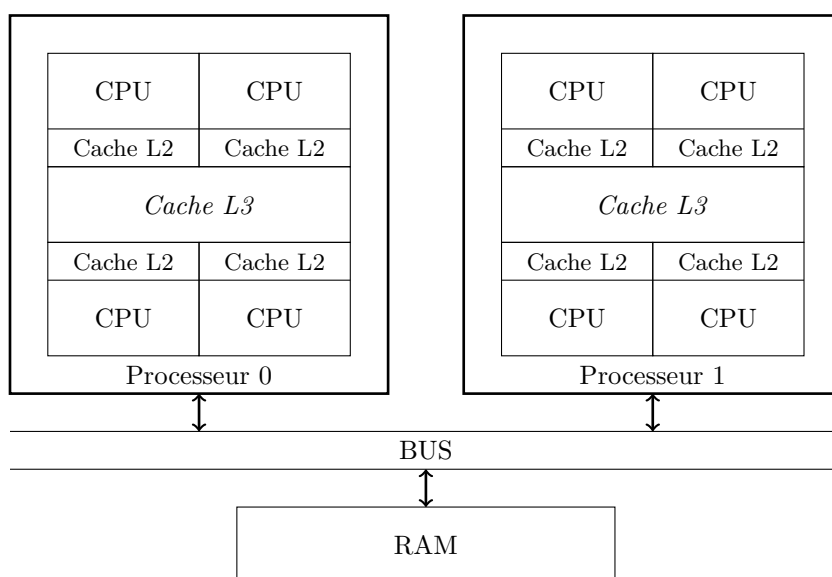


FIGURE 1.2 – Schéma simplifié d'une architecture SMP à deux processeurs et huit cœurs

1.3.1 Ordonnancement

Toutes les unités de calcul d'une architecture SMP ont le même statut et se partagent les différentes tâches. La répartition des processus est à la discrétion de l'ordonnanceur du système d'exploitation. Ainsi, si le débit de l'ordonnanceur est augmenté proportionnellement au nombre d'unités de calcul présentes sur l'architecture, chaque processus sera exécuté séquentiellement par une seule unité de calcul et son temps d'exécution sera au moins aussi important que sur un système monocœur.

Il est cependant possible dans le cas d'applications parallèles d'affecter les tâches à une unité de calcul en particulier. En OpenMP que nous présentons plus bas, cette affectation est réalisée *via* la variable d'environnement `GOMP_CPU_AFFINITY` qui prend comme valeur la liste des cœurs ordonnée selon les tâches : le premier cœur de la liste correspondra à la première tâche, le second à la seconde tâche et ainsi de suite. Sur une architecture à M cœurs, ces derniers sont représentés par un entier entre 0 et $M - 1$.

Ces identifiants peuvent être obtenus par des programmes dédiés comme `hwloc`¹. La figure 1.3 illustre le résultat du programme sur un nœud de la plate-forme HPC@LR² que nous avons utilisé pour une partie de nos tests. Ces nœuds sont composés de deux processeurs Intel Xeon X5650 (architecture Westmere) cadencés à 2.67 GHz. Les *processing units* (*PU*) correspondent aux processeurs logiques et sont numérotés dans l'ordre naturel (de 0 à 5 sur un processeur, de 6 à 11 sur le second). Pour nos applications arithmétiques, il sera donc fondamental de grouper les tâches qui communiquent le plus sur un même processeur car les caches de type L3 présents sur les processeurs sont de taille suffisamment grande (12 Mo) pour stocker l'ensemble des données nécessaires aux calculs.

1.3.2 Implémentation de codes parallèles dans le modèle synchrone

Si l'ordonnanceur permet un parallélisme de haut niveau entre les différents processus, pour bénéficier de l'ensemble des ressources de l'architecture une application doit être explicitement parallélisée dans son code source.

La gestion du parallélisme existe dans de nombreux langages de programmation. Dans ce manuscrit, nous présentons uniquement des applications sur architecture SMP écrites en langage C++. Pour les plates-formes UNIX, la première interface C a été standardisée en 1995 par la norme IEEE POSIX 1003.1c. Nommée PThreads, elle permet la gestion des tâches concurrentes (création, interruption, fusion) et la synchronisation à l'aide de sémaphores et d'exclusions mutuelles.

En 1997, les principaux constructeurs ont introduit une interface de programmation permettant le développement d'applications parallèles à l'aide de directives de compilation. Nommée OpenMP, cette interface que nous présentons plus bas permet une gestion simplifiée du parallélisme.

Il existe d'autres interfaces de programmation parallèle en C++. On peut noter en particulier HMPP et OpenCL, toutes deux particulièrement intéressantes et qui pourront constituer des perspectives attractives pour nos applications. En effet, ces interfaces ont vocation à s'abstraire totalement de l'architecture considérée et permettent d'écrire du code unifié qui sera exécutable sur des architectures hétérogènes (CPU, GPU, Cell). Elles donnent ainsi la possibilité de tirer parti de l'ensemble des ressources d'un système sans travail préalable d'implémentation spécifique pour l'architecture considérée.

Les applications parallèles sur architecture SMP que nous détaillons dans ce manuscrit utilisent l'interface OpenMP ainsi qu'un parallélisme synchrone. Le choix d'OpenMP est motivé par ses performances et son efficacité dans le modèle de parallélisme choisi. Cette interface nous permet en particulier d'utiliser un parallélisme haut niveau, par exemple la parallélisation *simple* de boucles, ou bien un parallélisme à grain plus fin où chaque tâche a son propre code, ce qui sera préféré en pratique pour nos applications. Enfin, dans sa dernière version, elle embarque une structure permettant un parallélisme asynchrone.

1. Portable Hardware Locality (<http://www.open-mpi.org/projects/hwloc/>)

2. Centre de compétences Calcul Haute Performance du Languedoc-Roussillon (<http://www.hpc-lr.univ-montp2.fr/>)



FIGURE 1.3 – Architecture d'un nœud de la plate-forme HPC@LR (deux processeurs Intel Xeon X5650)

Créée et maintenue par un consortium regroupant les constructeurs de processeurs (AMD, Intel, IBM, Nvidia...), OpenMP consiste en une série de directives de compilation placées autour des régions de code à paralléliser.

Exemple 1. Parallélisation d'une boucle pour en OpenMP

```
void somme_vecteurs(int *C, const int *A, const int *B, int n) {
#pragma omp parallel for
  for (int i = 0; i < n; i++) {
    C[i] = A[i]+B[i];
  }
}
```

Nous présentons ici une liste non exhaustive des commandes OpenMP utilisées dans ces travaux. Les spécifications du langage se trouvent sur le site officiel³, un descriptif complet du langage est disponible sur le site du laboratoire national Lawrence Livermore⁴. Enfin, une compilation des bonnes pratiques OpenMP est donnée dans [96].

OpenMP consiste en une série d'instructions divisées en plusieurs catégories (constructions de zones parallèles, types de régions parallèles, attributs de données, synchronisation), et d'une bibliothèque de fonctions permettant de modifier le comportement de l'application ou de récupérer des informations durant l'exécution.

Le code séquentiel est porté par une tâche maîtresse jusqu'à la rencontre d'une région parallèle identifiée par la directive `#pragma omp parallel`. Le code suivant cette directive est lancé sur un certain nombre de tâches défini explicitement (`num_threads`) ou implicitement (par le compilateur ou lors de l'exécution).

Remarque 1. Différents types de régions parallèles peuvent être définis : itérations d'une boucle (directive `for` ou `do`), sections de codes différentes exécutées concurremment (directive `section`), section de code exécutée par une et une seule tâche (directive `single` ou `master`).

La protection des variables est également gérée par OpenMP. Nous utilisons principalement deux directives selon qu'une variable est partagée par toutes les tâches (`shared`) ou privée (`private`). Dans ce dernier cas, chaque tâche possède une copie locale de la variable.

Pour éviter les collisions lors de l'écriture dans des variables partagées ou pour garantir que toutes les tâches ont fini une portion de code avant de passer à la suivante, OpenMP dispose de directives de synchronisation. Celle que nous utilisons par la suite est la barrière de synchronisation (`#pragma omp barrier`) qui bloque chaque tâche jusqu'à ce que toutes aient atteint le même point. Si cette directive est vitale pour garantir le bon déroulement de l'application et la validité du résultat final, elle doit néanmoins être utilisée avec précaution car la synchronisation est la première cause de l'*overhead* lié au parallélisme.

1.4 Processeurs graphiques

Les processeurs graphiques sont aujourd'hui incontournables dans le calcul haute performance. Trois raisons à cela : l'arrivée de langages dédiés facilement utilisables comme CUDA (Nvidia) ou ATI Stream (AMD) ; leur grande puissance de calcul et leur faible coût. Ces GPU (Graphics Processing Units) ne sont plus réservés aux seuls calculs graphiques, mais utilisés comme coprocesseur pour des calculs scientifiques. À l'heure actuelle, ce *General-Purpose computing on Graphics Processing Units* (GPGPU) [98, 76] est utilisé dans de nombreux domaines : finance, bio-informatique, mécanique des fluides, chimie, etc.

3. <http://openmp.org/wp/>

4. <https://computing.llnl.gov/tutorials/openMP/>

Les processeurs graphiques sont de type SIMD : chaque unité de calcul du processeur effectue le même traitement sur des données différentes. En général, les unités de calculs sont regroupées sur un ou plusieurs *multiprocesseurs*. Ces derniers embarquent les unités de contrôle et un certain nombre de mémoires (ou de caches). Les tâches qui s'exécutent sur un même multiprocesseur peuvent donc communiquer entre elles. Enfin les cartes graphiques disposent d'une mémoire globale accessible à toutes les tâches qui sert également d'interface avec la mémoire centrale de la machine hôte.

La figure 1.4 (éditée par Nvidia) présente une comparaison assez fidèle entre un processeur multicœur et un processeur graphique. L'échelle de cette figure n'est pas anodine. En effet, si chaque multiprocesseur du GPU embarque un nombre plus important d'unités de calcul qu'un processeur classique, elles sont généralement moins puissantes. De plus, la taille de la mémoire cache permettant la communication entre les différents cœurs d'un processeur est plus importante que celle de la mémoire partagée des GPU.

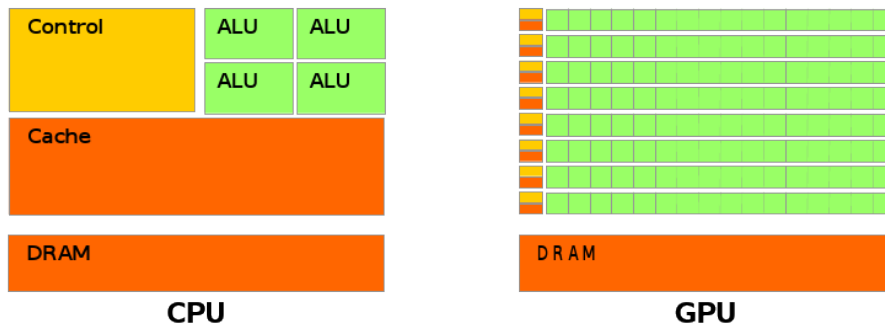


FIGURE 1.4 – Comparaison CPU-GPU

L'utilisation des processeurs graphiques prend tout son sens pour des applications hautement parallèles qui nécessitent des milliers de tâches indépendantes. Il est en effet nécessaire d'occuper totalement et durablement le processeur graphique pour amortir le coût de ses communications avec la machine hôte.

Remarque 2. Le débat processeur classique ou processeur graphique est loin d'être clos [104], en particulier avec l'apparition progressive de cartes utilisant les deux architectures comme le futur Knights Corner d'Intel dérivé du défunt projet Larrabee : une carte PCI servant de coprocesseur contenant 50 cœurs x86 annoncé pour 2012 avec un potentiel de calcul d'un TéraFLOPS.

Pourquoi s'intéresser ici à ce type d'architecture ? En réalité, la cryptologie dans son ensemble a déjà commencé à mesurer le potentiel des GPU : transposition de la méthode de factorisation basée sur les courbes elliptiques [7], cryptosystèmes asymétriques [94], RSA [68], etc. On peut également citer le logiciel libre Pyrit⁵ qui permet de casser la sécurité des clés Wi-fi WPA et WPA2 en utilisant la puissance de calcul de différentes architectures, dont les GPU Nvidia et ATI.

Il existe néanmoins certaines limites à leur utilisation comme unités de calcul haute performance pour des applications de bas niveau comme l'arithmétique, en particulier les coûts de communication entre la carte et la mémoire centrale de l'architecture qui l'héberge. De plus, les langages proposés restent des langages propriétaires fournis par les constructeurs. Pour être multiplate-forme, il faudra utiliser des langages de plus haut niveau comme OpenCL ou HMPP.

Nous avons utilisé des cartes Nvidia et le langage CUDA associé. Nous allons décrire l'architecture de la famille Tesla et l'interface de programmation correspondante. Pour illustrer le potentiel des processeurs graphiques, nous finissons par un exemple concret : l'implémentation de recherche de similarités dans une base de données musicales utilisant la puissance des processeurs graphiques, qui a été un projet annexe à cette thèse.

5. <http://code.google.com/p/pyrit/>

1.4.1 Les cartes Tesla

L'architecture Tesla est illustrée à la figure 1.5. Il existe de nombreuses références à cette technologie. Nous nous appuyons en particulier sur [71] et le tutoriel de Cyril Zeller [102].

1.4.1.1 Architecture

Un processeur graphique Tesla embarque un ou plusieurs multiprocesseurs (MP) composé(s) d'un même nombre d'unités de calcul nommées *Single Processors (SP)*, d'une unité de contrôle, d'une zone de mémoire partagée et d'un certain nombre de registres. Il contient également une zone mémoire de taille conséquente et accessible par tous les SP qui sert d'interface avec la mémoire centrale de la machine hôte.

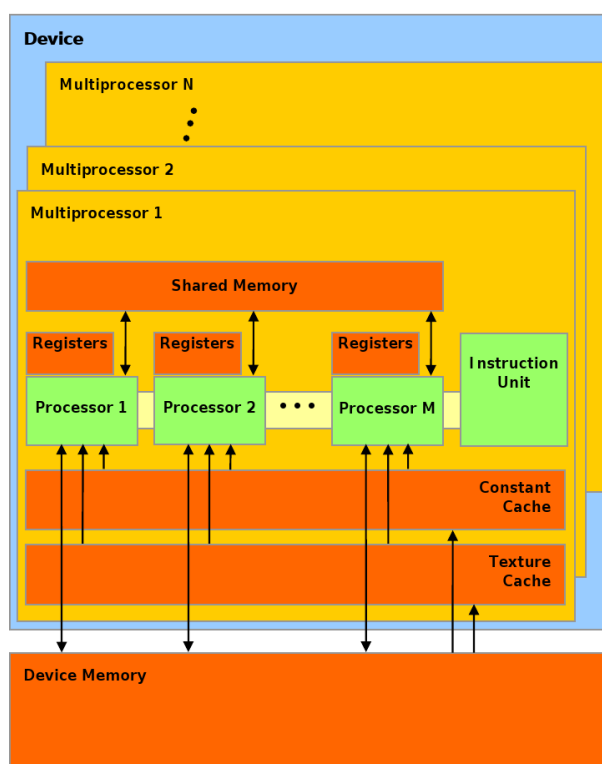


FIGURE 1.5 – Architecture des GPU Tesla (Nvidia)

1.4.1.2 CUDA

Ces cartes graphiques sont programmables grâce à la technologie CUDA (Compute Unified Device Architecture), un environnement de programmation parallèle qui étend le langage C. Le guide de programmation [73], le manuel de référence [74] ainsi qu'une compilation des bonnes pratiques CUDA [72] sont disponibles sur la *Cuda Zone*⁶.

Une application CUDA est un programme séquentiel qui lance des *kernels* parallèles sur la carte. Le terme *kernel* désigne l'exécution d'une fonction par un certain nombre de *threads*. À tout moment, un seul kernel est exécuté par le processeur graphique. Chaque thread est étiqueté par un identifiant unique et agit sur des données différentes des autres threads.

Les threads sont répartis en un certain nombre de blocs de même taille. L'ensemble des blocs de threads forme une grille qui correspond au kernel. Tous les threads d'un même bloc sont exécutés sur le même multiprocesseur avec deux conséquences immédiates : ils peuvent communiquer entre eux *via* la mémoire partagée et ils peuvent se synchroniser. En revanche, les blocs de threads ne peuvent pas être synchronisés. Ils sont exécutés concurremment et dans un ordre aléatoire.

6. http://www.nvidia.com/object/cuda_home_new.html

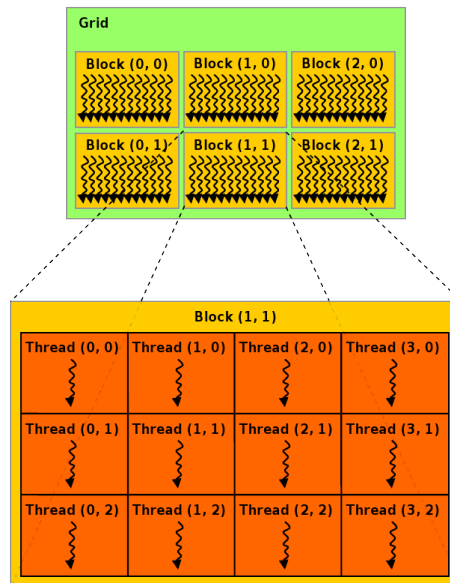


FIGURE 1.6 – Schéma d'exécution du code CUDA

L'exemple 2 montre le lancement d'un kernel CUDA. Le mot clé `__global__` identifie une fonction qui est lancée sur la carte à partir d'une fonction exécutée sur la machine hôte (`__host__`).

Exemple 2. Lancement d'un kernel CUDA

```

__global__ void do_smth(int *param) {
    //Traitement sur la carte
}
__host__ int main() {
    ...
    dim3 dimBlock(8,8,8); //Bloc de 512 threads
    dim3 dimGrid(8,8);    //Grille de 64 blocs
    //Lancement du kernel sur la carte - 512*64 = 32768 threads
    do_smth<<<dimGrid, dimBlock>>>(param);
    ...
}

```

1.4.1.3 Exécution d'une grille

Chaque multiprocesseur exécute des ensembles de blocs dits actifs les uns après les autres. Les registres et la mémoire partagée sont divisés de manière égale pour tous les threads actifs (ceux qui appartiennent aux blocs actifs). Le nombre de blocs actifs dépend donc des registres et de la mémoire partagée utilisés par chaque thread. S'il ne peut pas y avoir au moins un bloc actif, le kernel ne peut pas être lancé. Nous verrons au chapitre 3 que cela pose problème pour une de nos applications.

Chaque bloc actif est divisé en unités appelées *warp* de threads de taille 32 sur nos architectures. Tous les threads d'un warp sont exécutés physiquement au même moment sur la carte.

Un point important à noter est l'asynchronisme de l'exécution des kernels et du code séquentiel : le CPU n'est pas bloqué dans l'attente de la fin des exécutions sur la carte mais peut continuer à travailler de manière parallèle jusqu'à rencontrer une instruction bloquante.

1.4.1.4 Mémoires

L'aspect primordial dans le GPGPU est la gestion de la mémoire. Pour obtenir les meilleures performances, il est indispensable d'identifier les avantages et les inconvénients que chacune des zones

mémoires disponibles. Au nombre de six, elles sont décrites dans le tableau 1.1. Les transferts de données de la RAM aux mémoires globale, texture et constante de la carte graphique sont réalisés par le code CPU.

Les transferts entre la mémoire globale et les mémoires rapides peuvent être réalisés uniquement par les kernels.

Mémoire	Accès	Taille	Portée	Latence
Registres	R/W	8192/MP	Thread	-
Partagée	R/W	16Ko/MP	Bloc	≈ 10 cycles
Globale	R/W	Dépend de l'architecture	Grille	400-600 cycles
Locale	R/W	16Ko/T	Thread	400-600 cycles
Texture	R	Dépend de l'architecture	Grille	Forte
Texture cache	R	6 ou 8Ko/MP	Bloc	-
Constant	R	64Ko	Grille	Faible
Constant cache	R	8Ko/MP	Bloc	-

TABLE 1.1 – Description des zones mémoires pour la CUDA Capability 1.1

Pour nos applications arithmétiques, nous notons trois faits importants : le nombre de registres est limité, la taille de la mémoire partagée est relativement faible et l'accès la mémoire globale est coûteux. Il faut également noter que les registres ne sont pas indexables. Les tableaux temporaires déclarés dans les kernels sont stockés en mémoire dite « locale », qui physiquement correspond à une partie de la mémoire globale, avec une latence en conséquence.

1.4.1.5 Optimisations et bonnes pratiques

Il existe un certain nombre de règles à respecter pour obtenir les meilleures performances. Elles sont décrites dans [72]. On rappelle ici certaines d'entre elles, primordiales pour nos applications :

- R1** minimiser les transferts entre la carte et la machine hôte ;
- R2** maximiser l'occupation de la carte : avoir un nombre maximal de blocs actifs ;
- R3** éviter les branches divergentes entre threads d'un même warp : éviter les instructions conditionnelles ;
- R4** minimiser l'utilisation de la mémoire globale : lui préférer la mémoire partagée et y charger les données utilisées plusieurs fois.
- R5** utiliser des accès coalescents pour les lectures et les écritures dans la mémoire globale. La latence pour accéder aux données stockées en mémoire globale est importante (plusieurs centaines de cycles). Si le placement des données et le code CUDA suivent certaines règles, il est possible de lire ou d'écrire plusieurs blocs contigus de la mémoire globale en une seule transaction. Plus précisément, les lectures et les écritures de données de 32 ou 64 bits pour les threads d'un demi-warp (16 threads) peuvent être effectuées en une seule transaction mémoire (deux dans le cas de données de 128bits). Pour cela, le i -ème thread du demi-warp doit accéder à la i -ème case mémoire d'un segment aligné sur une adresse mémoire multiple de 16, comme illustré par la figure 1.7. On parle alors d'accès coalescents.

Depuis 2005 et la *révolution* GPGPU, de très nombreuses applications sont portées sur les processeurs graphiques, avec des performances par rapport au CPU souvent alléchantes. Pour mesurer le potentiel de ces technologies et mettre en pratique les règles ci-dessus, nous avons développé une version GPU d'une application de recherche de similarités dans une base de données. Ce travail s'est effectué en collaboration avec Pascal Ferraro et Pierre Hanna du LaBRI⁷. Il a été publié dans [31].

7. Laboratoire Bordelais de Recherche en Informatique

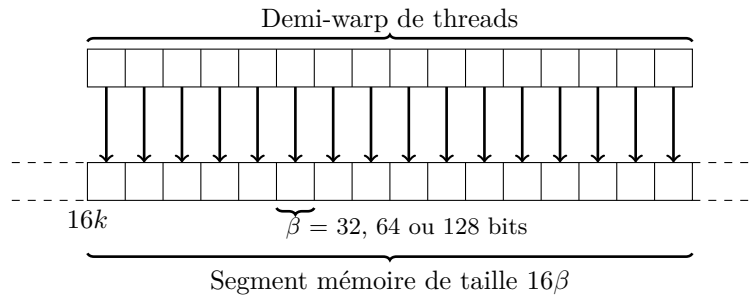


FIGURE 1.7 – Accès coalescents à la mémoire globale

1.4.2 Un exemple concret : les requêtes par fredonnement sur GPU

La recherche de similarités dans les bases de données est devenue un défi majeur dans de nombreux domaines en raison du nombre toujours croissant de données stockées sur les supports numériques. Elle s'appuie généralement sur des fonctions de comparaison qui permettent de mesurer la ressemblance ou la dissemblance entre ensembles de données. Au vu des tailles des bases de données actuelles, ces traitements sont souvent réalisés par des architectures parallèles comme les grilles de calculs.

L'application visée ici nécessite une fonction de comparaison de séquences. Après avoir défini les requêtes par fredonnement, nous détaillons donc la méthode d'alignement de Smith-Waterman, puis nous décrivons notre implémentation sur GPU, et enfin les résultats expérimentaux obtenus, qui prouvent l'intérêt de l'utilisation des processeurs graphiques avec un code parallèle jusqu'à 160 fois plus rapide que le code séquentiel.

1.4.2.1 Requête par fredonnement

Un des buts principaux des systèmes d'extraction de données musicales est la recherche de morceaux de musique dans de grandes bases de données à partir d'une description ou d'un exemple. L'application visée ici est appelée *requête par fredonnement*. Elle consiste à retrouver le titre d'une chanson à partir de son fredonnement par une recherche dans un ensemble de fichiers de référence au format symbolique MIDI. Pour ce faire, l'application se divise en deux étapes : la conversion d'un fichier audio en une séquence de notes et la comparaison de celle-ci avec l'ensemble des références, chacune également représentée par une séquence de notes. Seule cette seconde étape nous intéresse ici. Elle est réalisée par l'algorithme 1.

Pour chaque séquence s_i de la base de références, on calcule un score de ressemblance avec la séquence s_r à l'aide d'une fonction de comparaison sc . Plus ce score est élevé, plus les deux séquences de notes sont semblables.

Algorithme 1: Requête par fredonnement

Données : La séquence s_r (requête), un ensemble de m séquences s_i (références), une fonction sc de comparaison de séquences retournant un score de ressemblance

Résultat : s_i tel que $sc(s_r, s_i) \geq sc(s_r, s_j) \forall j \neq i, j \in [0, m - 1]$

```

1  $scmax \leftarrow -1, imax \leftarrow -1$ 
2 pour  $i$  de 0 à  $m - 1$  faire
3    $s \leftarrow sc(s_r, s_i)$ 
4   si  $s > scmax$  alors  $scmax \leftarrow s, imax \leftarrow i$ 
5 fin
6 retourner  $s_{imax}$ 

```

L'efficacité de cette méthode dépend de la fonction de score choisie. L'application requête par fredonnement utilise une méthode d'alignement de séquences empruntée à la bio-informatique.

1.4.2.2 Alignement de séquences

L'alignement entre deux séquences u et v définies sur un alphabet \mathcal{A} et de longueurs respectives l_u et l_v est une chaîne w sur l'alphabet des paires de lettres $(\mathcal{A} \cup \{-\}) \times (\mathcal{A} \cup \{-\})$, où $\{-\} \notin \mathcal{A}$ est la lettre vide et représente un *trou*. La projection sur la première composante correspond à u et la projection sur la seconde composante correspond à v . Ainsi, on peut définir trois types de paires de lettres dans w , correspondant à trois opérations. La paire (a, b) avec $a, b \in \mathcal{A}$ est appelée *substitution*, la paire $(a, -)$ est appelée *suppression*, et la paire $(-, a)$ est appelée *insertion*. L'exemple suivant donne deux alignements possibles entre deux séquences.

Exemple 3. Soient $\mathcal{A} = \{a, b, c, d, e, f, g\}$ et les séquences $u = cbcfab$ et $v = ecgabf$. Deux alignements possibles pour ces séquences sont :

$$\begin{pmatrix} - \\ e \end{pmatrix} \begin{pmatrix} c \\ c \end{pmatrix} \begin{pmatrix} b \\ g \end{pmatrix} \begin{pmatrix} - \\ a \end{pmatrix} \begin{pmatrix} c \\ b \end{pmatrix} \begin{pmatrix} f \\ f \end{pmatrix} \begin{pmatrix} a \\ - \end{pmatrix} \begin{pmatrix} b \\ - \end{pmatrix} \quad \text{et} \quad \begin{pmatrix} c \\ e \end{pmatrix} \begin{pmatrix} b \\ - \end{pmatrix} \begin{pmatrix} c \\ c \end{pmatrix} \begin{pmatrix} f \\ - \end{pmatrix} \begin{pmatrix} - \\ g \end{pmatrix} \begin{pmatrix} a \\ a \end{pmatrix} \begin{pmatrix} b \\ b \end{pmatrix} \begin{pmatrix} - \\ f \end{pmatrix}.$$

Quel est le meilleur alignement ?

Chacune des opérations est pondérée : l'insertion et la suppression par une pénalité g et les substitutions par une matrice s de taille $(\#\mathcal{A}) \times (\#\mathcal{A})$ définissant les coûts des transformations $(a, b) \forall a, b \in \mathcal{A}$. Le *score de ressemblance* d'un alignement est alors défini comme la somme des scores de chacune de ses paires alignées. Sans perte de généralité, on pose $l_u \geq l_v$.

Notre application utilise un algorithme d'alignement *local* qui identifie des régions de similarités entre deux séquences. Proposé par Smith et Waterman dans [92], cet algorithme construit dynamiquement ligne par ligne une matrice M de taille $(l_u + 1) \times (l_v + 1)$ telle que

$$\begin{cases} M(0, 0) = 0 \\ M(0, j) = 0 \quad \forall j \in [0, l_v] \\ M(i, 0) = 0 \quad \forall i \in [0, l_u] \end{cases}$$

et $\forall i \in [1, l_u], \forall j \in [1, l_v]$:

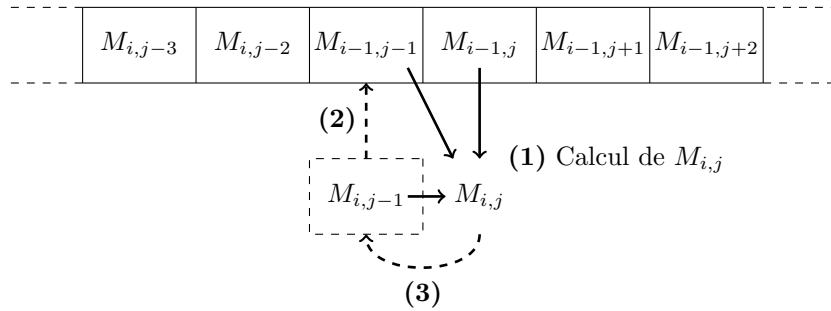
$$M(i, j) = \max \begin{cases} M(i-1, j-1) + s(u_i, v_j) & \text{substitution} \\ M(i-1, j) + g & \text{insertion} \\ M(i, j-1) + g & \text{suppression} \\ 0 & \end{cases}$$

La complexité de l'algorithme Smith-Waterman est égale à $\mathcal{O}(l_u, l_v)$. Basée sur cet algorithme, notre fonction $sc(s_r, s_i)$ renverra la valeur maximale de la matrice.

La requête peut comporter des erreurs dues au fredonnement : une série de notes plus hautes (aiguës) ou plus basses (graves) que les notes du morceau de référence. On parlera alors de transpositions locales. Pour prendre en compte ces éventuelles erreurs, la fonction sc ajoute une dimension à la méthode Smith-Waterman en calculant une matrice pour chaque transposition locale. Cette méthode (LT) a été proposée par Allali et al. dans [2]. La complexité du problème devient $\mathcal{O}(\Delta \times l_u \times l_v)$ où Δ est le nombre de transpositions autorisées. En pratique, on fixera $\Delta = 12$, qui correspond aux 12 demi-tons de l'échelle chromatique.

1.4.2.3 Implémentation sur GPU

La parallélisation de cette application pourrait se faire à deux niveaux : parallélisation de la boucle de l'algorithme 1, ou parallélisation de la fonction sc , c'est-à-dire parallélisation de l'algorithme Smith-Waterman. Des implémentations existent déjà pour cette dernière en bio-informatique pour l'alignement de séquences d'ADN/ARN [70, 63]. Pour être efficaces, ces implémentations doivent cependant travailler sur des séquences très longues (au moins plusieurs milliers de nucléotides) car l'algorithme est dynamique et donc mal approprié au parallélisme. Dans notre application où les séquences sont composées de quelques dizaines de notes au maximum, une parallélisation à ce niveau se révélerait particulièrement inefficace.

FIGURE 1.8 – Calcul de la case $M_{i,j}$ de l’algorithme Smith-Waterman

Notre approche consiste donc à paralléliser l’application au niveau de la boucle sur les séquences de référence et suit le schéma suivant :

1. conversion du fichier audio en séquence de notes s_r de taille l_r ;
2. lecture de la base de données des fichiers MIDI et création des m séquences de référence s_i de taille l_i ;
3. copie de ces données dans les mémoires du processeur graphique ;
4. lancement de m tâches t_i sur la carte ;
5. chaque tâche t_i calcule $sc(s_r, s_i)$ en parallèle et stocke le résultat obtenu ;
6. copie des résultats sur la machine hôte.

Seule l’étape 5. est réalisée sur la carte graphique. Toutes les autres opérations sont réalisées par la machine hôte.

Le facteur majeur limitant les performances de calcul sur les processeurs graphiques est la gestion de la mémoire. Nous allons donc identifier les données utilisées par l’application et définir où et comment les stocker pour minimiser les latences et les transferts.

La requête est de petite taille (quelques dizaines de notes, chaque note représentée par deux flottants de 32 bits) et sera utilisée plusieurs fois par les m comparaisons. Les *texture caches* de chacun des multiprocesseurs sont de taille suffisante pour la stocker. On copie donc cette séquence en mémoire texture.

Pour minimiser les transferts entre la mémoire centrale de la machine hôte et la mémoire de la carte (règle R1), la base de données est transférée une seule fois. Seule la mémoire globale de la carte peut la stocker en totalité. Chaque tâche t_i requiert la séquence référence s_i qui est utilisée plusieurs fois pour la construction de la matrice Smith-Waterman. Pour éviter la multiplication des lectures en mémoire globale (R4), chaque tâche charge sa séquence s_i en mémoire partagée. Cette étape en amont du calcul est réalisée par tous les threads d’un bloc. Pour minimiser les transferts, les données sont ordonnées de manière à utiliser les lectures coalescentes (R5) : dans notre cas, il suffit simplement de transposer le tableau contenant toutes les séquences de manière à ce que la première note de s_0 soit contiguë à la première note de s_1 elle-même contiguë à la première note de s_2 , etc.

La matrice Smith-Waterman est construite dynamiquement, le calcul du score de chaque case $M(i, j)$ nécessitant la connaissance des scores aux cases $M(i-1, j)$, $M(i-1, j-1)$ et $M(i, j-1)$. Cependant, on souhaite connaître uniquement le score maximal de la matrice, il n’est donc pas nécessaire de la stocker en totalité. Il suffira donc d’un espace mémoire égal à la longueur de la séquence la plus courte : pour calculer la ligne i il est suffisant connaître la ligne $i-1$, et utiliser deux variables en plus pour éviter l’écrasement des données, comme illustré à la figure 1.8.

La taille en mémoire pour une matrice Smith-Waterman est donc égale à $\min(l_r, l_i) \times 8$ octets (une note étant représentée par deux flottants de 32 bits). La méthode de transposition locale ajoute une dimension et nécessite de stocker 12 matrices, soit en mémoire $12 \min(l_r, l_i) \times 8$ octets. La mémoire

partagée est répartie entre tous les *threads* d'un bloc. On pourra y stocker nos matrices si le nombre B de *threads* par blocs le permet, c'est-à-dire si $B \times (12 \min(l_r, l_i) + l_i) \times 8$ octets est inférieur à la taille de la mémoire.

Sur nos architectures la mémoire partagée est de taille 16Ko et n'est pas suffisamment grande pour stocker nos matrices. Elles sont donc placées en mémoire globale en respectant les conditions permettant les lectures et les écritures coalescentes (R5).

La pondération g et la matrice s sont stockées en mémoire constante. Le résultat de chacune des comparaisons est un flottant correspondant au score de ressemblance entre les séquences considérées. Ces résultats sont stockés en mémoire globale et sont rapatriés sur la machine hôte à la fin de l'exécution.

1.4.2.4 Résultats expérimentaux

Le tableau 1.2 présente les trois architectures sur lesquelles les tests ont été effectués.

	Type PC	CPU	GPU	#SP	Mémoire
W1	Station de travail	3GHz Intel Core 2 Duo	GeForce 9800 GX2	128	512Mo
W2	Station de travail	2.8GHz Intel Xeon QuadCore	GeForce 8800 GT	112	512Mo
L1	Portable	2.53GHz Intel Core 2 Duo	GeForce 9400 M	16	RAM ⁸

TABLE 1.2 – Plate-formes de tests

Nos expérimentations sont basées sur un corpus de requêtes proposé aux MIREX⁹ 2007 et 2008. Il s'agit de 2797 fichiers audio de 48 fredonnements différents au format WAV et des 48 fichiers MIDI *vérité* correspondants. La première base de données (appelée BD1) utilisée contient 2000 fichiers MIDI issus de la collection Essen¹⁰ plus les 48 fichiers *vérité*. La seconde base (appelée BD2) correspond aux 5982 fichiers MIDI de cette collection (plus les 48 fichiers *vérité*). Enfin, la troisième base (appelée BD3), issue du RISM A/II¹¹ et proposée au MIREX 2005, contient 17433 fichiers. Les fichiers MIDI des deux premières bases sont plus longs (plusieurs dizaines ou centaines de notes) que les fichiers *vérité* et ceux de la BD3 (plusieurs dizaines de notes au maximum).

Nous avons implémenté en CUDA l'algorithme classique Smith-Waterman (SW) et la variante basée sur les transpositions locales (LT). Enfin, sachant que les requêtes correspondent au début des fichiers de référence, nous avons implémenté deux variantes de ces algorithmes qui réalisent l'alignement entre la requête et $l_r + 10$ notes du fichier de référence. Ces deux algorithmes sont notés respectivement SW10 et LT10. Comme attendu, les scores de ressemblance obtenus entre les versions CPU et GPU des différentes méthodes sont identiques.

Le tableau 1.3 présente les temps de calcul pour chacun des algorithmes sur les trois plate-formes pour les trois bases de données. Il n'y a pas de différences entre les deux versions de l'algorithme de Smith-Waterman, ce qui n'est guère étonnant car les fichiers de référence ne sont pas très longs. Par contre, la méthode utilisant les transpositions locales est plus coûteuse que les autres, elle réalise en effet 12 constructions de matrices au lieu d'une seule.

8. Le GPU n'a pas de mémoire globale dédiée. Il utilise la mémoire centrale de l'hôte.

9. Music Information Retrieval Evaluation eXchange

10. <http://essen.themefinder.org/>

11. Répertoire international des ressources musicales <http://www.rism.info/>

BD	Configuration	SW10	SW	LT10	LT
BD1	L1	7 : 33	7 : 33	7 : 50	40 : 54
	W1	4 : 12	5 : 05	5 : 01	20 : 16
	W2	6 : 00	6 : 00	6 : 01	14 : 42
BD2	L1	7 : 53	7 : 51	15 : 10	79 : 45
	W1	6 : 55	6 : 54	6 : 10	25 : 46
	W2	6 : 18	6 : 18	6 : 16	20 : 40
BD3	L1	9 : 20	9 : 19	41 : 31	44 : 48
	W1	5 : 15	6 : 05	10 : 09	25 : 27
	W2	7 : 25	7 : 27	7 : 53	21 : 15

TABLE 1.3 – Temps de calcul ($mm : ss$) de différents algorithmes sur les trois plateformes en utilisant les trois bases de données

Le résultat le plus intéressant est le gain en vitesse de calcul apporté par l'utilisation des processeurs graphiques. Le tableau 1.4 présente les temps de calcul des algorithmes LT et LT10 avec BD1 et BD2 sur la plateforme W2, ainsi que l'accélération de l'implémentation GPU.

		DB1 and LT	DB1 and LT10	DB3 and LT10
(1)	Version CPU (minutes)	595 (9h55)	326 (5h26)	1282 (21h22)
(2)	Version GPU ($mm : ss$)	14 : 42	6 : 01	7 : 53
	Accélération (1)/(2)	$\times 40$	$\times 54$	$\times 160$

TABLE 1.4 – Comparaison des performances CPU et GPU de l'application requête par fredonnement

Il est clair que l'utilisation des GPU pour cette application permet des gains de temps non négligeables. Les différences entre les gains observés pour les trois expérimentations s'expliquent par les algorithmes et les bases de données utilisées. En effet, la méthode classique LT opère sur l'ensemble des notes des fichiers de références ce qui implique un nombre plus important d'accès à la mémoire globale, en particulier sur la base DB1 où les séquences sont longues. La différence entre les deux dernières colonnes du tableau s'expliquent par la taille des chaînes de la base DB3, souvent inférieure à $l_r + 10$.

Une question se pose néanmoins : comment peut-on avoir une application 160 fois rapide sur un GPU qui dispose de seulement 112 cœurs ? Il faut se rappeler que l'application requête par fredonnement se décompose en deux parties : une partie traitement du signal qui transforme un fichier son en suite de notes, et une partie alignement. Lorsque l'on veut traiter 2797 requêtes, il y a 2797 transformations à réaliser. Or, dans la version GPU, ce traitement peut être réalisé sur le CPU en parallèle aux tâches lancées sur le GPU : pendant l'alignement de la requête i , le processeur de la machine hôte peut prétraiter la requête $i + 1$.

Au niveau séquentiel, sur un processeur on compare la requête à la base de données en 27 secondes. Sans compter le coût de communication CPU-GPU, la comparaison se ferait séquentiellement sur la carte en 19 secondes. Cet écart s'explique par le fait que notre code est optimisé, en particulier pour la gestion de la mémoire. Le cœur de l'application, l'algorithme Smith-Waterman, a été réécrit entre autres pour minimiser les transferts mémoire. On ne stocke par exemple qu'une seule ligne de la matrice de Smith-Waterman, avec des accès optimisés pour la lecture et l'écriture.

Les processeurs graphiques se révèlent donc particulièrement performants pour une application nécessitant peu de mémoire (plusieurs Mo) et dans laquelle tous les traitements parallèles sont totalement indépendants et qui requièrent une arithmétique suffisamment intense pour masquer les latences d'accès aux données.

Ces architectures parallèles sont performantes pour le calcul scientifique. Cependant, peuvent-elles nous permettre d'accélérer les calculs arithmétiques décrits dans le chapitre suivant ? Utiliser les

architectures SMP permet de paralléliser les applications à n'importe quel niveau. Cela signifie en particulier pour la cryptographie asymétrique que l'on peut se placer sur n'importe quelle couche arithmétique. Nous avons cependant mentionné le surcoût lié au parallélisme. Nous verrons que l'intensité arithmétique de certaines opérations ne sera pas suffisante pour le masquer.

Il est plus difficile de paralléliser entièrement une application sur les processeurs graphiques, car ce type d'architecture requiert des tâches suffisamment coûteuses en temps de calcul pour masquer d'une part les latences d'accès aux mémoires et d'autre part le fait que les unités de calcul sont moins rapides que les processeurs classiques. Pour tirer avantage des cartes graphiques, on va donc placer le parallélisme au niveau arithmétique le plus haut, ou paralléliser des instances de l'application.

CHAPITRE 2

ARITHMÉTIQUE POUR LA CRYPTOGRAPHIE

Sommaire

2.1	La cryptographie à clé publique	21
2.2	Arithmétique multiprécision	24
2.2.1	Représentations des entiers multiprécision	24
2.2.2	Addition et soustraction	24
2.2.3	Multiplication	24
2.2.4	Division	28
2.2.5	Résumé des complexités	29
2.2.6	L'arithmétique multiprécision en pratique	29
2.3	Arithmétique modulaire	30
2.3.1	Addition et soustraction	31
2.3.2	Multiplication	31
2.4	Arithmétique des courbes elliptiques	35
2.4.1	La cryptographie basée sur les courbes elliptiques	35
2.4.2	Définition des courbes elliptiques	36
2.4.3	Projections et équations de courbes	37
2.4.4	La multiplication scalaire	38

Les protocoles de cryptographie asymétrique (ou cryptographie à clé publique) reposent sur des fonctions qui nécessitent des calculs arithmétiques dans différentes structures mathématiques. Face aux enjeux de sécurité actuels qui nécessitent des tailles de clés de plus en plus grandes, ces arithmétiques se doivent d'être efficaces.

Après avoir défini la cryptographie asymétrique et identifié ses bases mathématiques, nous présentons dans ce chapitre les trois arithmétiques sur lesquelles nous avons travaillé. Pour l'arithmétique multiprécision nous nous appuyons sur [18, 65, 69, 56] ; pour l'arithmétique modulaire sur [18, 65] ; et pour l'arithmétique des courbes elliptiques sur [43, 21, 90].

2.1 La cryptographie à clé publique

Utilisée depuis l'antiquité [91], la cryptographie classique consiste à rendre un message incompréhensible lorsqu'il voyage d'une source vers une destination à travers un réseau. Si les réseaux en question ont bien évolué en plus de deux millénaires, l'idée principale est toujours la même : le message doit être impossible à déchiffrer par une autre entité que les parties communicantes, même s'il transite par un réseau non sécurisé.

Dès 1883, Kerckhoffs [53] a posé comme principe que la sécurité d'un cryptosystème doit reposer uniquement sur un secret possédé par les parties communicantes et généralement appelé *clé secrète*.

Le corollaire d'une telle hypothèse est qu'un éventuel adversaire a la connaissance ou la maîtrise de tout le reste : protocoles et algorithmes utilisés, réseau par lequel va transiter le message, etc.

Le but de la cryptographie à clé privée, également appelée cryptographie symétrique est de garantir cette confidentialité. Les entités communicantes possèdent un secret commun avec lequel ils pourront chiffrer et déchiffrer leurs messages. Si Alice et Bob souhaitent communiquer, ils choisissent ensemble une fonction f inversible ainsi qu'une clé secrète s connue d'eux seuls. Lorsque Alice veut envoyer un message m à Bob, elle calculera $c = f(m, s)$ et transmettra c à Bob. Pour retrouver le message original, Bob calculera simplement $m = f^{-1}(c, s)$.

La multiplication des échanges a posé la limite de ce type de cryptographie : si un adversaire intercepte un seul message, il ne pourra pas le décrypter ou retrouver la clé secrète. Mais s'il en intercepte un nombre suffisant il pourra, à l'aide de techniques d'analyses statistiques par exemple, obtenir des informations sur la clé de chiffrement [93]. En théorie, la réponse à ce problème est triviale : utiliser autant de clés secrètes que de messages. En pratique, la réalisation de tels protocoles est difficile, voire impossible hors d'un cadre gouvernemental, car la logistique induite est particulièrement contraignante (gestion de dictionnaires de clés) et difficile à mettre en œuvre (stockage « sûr » des clés, etc.).

La cryptographie asymétrique permet de communiquer de manière sécurisée sans connaître au préalable de secret commun. Elle permet entre autre l'échange de clés, notamment grâce au protocole de Diffie Hellman (1976 [23]) : il n'est pas nécessaire qu'Alice et Bob connaissent un secret commun *a priori* : ils peuvent le fabriquer. L'idée de cet échange de clés est de remplacer les fonctions mathématiques inversibles utilisées par la cryptographie symétrique par des fonctions mathématiques à sens unique. Une fonction à sens unique $f : X \rightarrow Y$ est une fonction telle que pour tout $x \in X$, $f(x)$ est facile à calculer (c'est-à-dire en temps polynomial), mais étant donné $y = f(x) \in Y$, il est difficile de retrouver x (en temps polynomial). On ne sait pas prouver que de telles fonctions existent (leur existence démontrerait $P \neq NP$) mais l'on peut cependant trouver des fonctions pour lesquelles il n'existe pas à l'heure actuelle d'algorithme pour retrouver x en fonction de y en temps polynomial.

L'échange de clés de Diffie-Hellman que nous présentons ci-dessous est basé sur le problème du logarithme discret dans le sous-groupe multiplicatif d'un corps fini $\mathbb{Z}/p\mathbb{Z}$: si connaissant g un générateur de $(\mathbb{Z}/p\mathbb{Z})^*$ et un entier e il est facile de calculer $f = g^e \bmod p$, retrouver e connaissant f , g et p est un problème difficile pour des paramètres bien choisis.

Alice et Bob choisissent ensemble un corps fini de cardinal p et un générateur g de son sous-groupe multiplicatif. Ces deux paramètres sont publics, et leur connaissance par un éventuel adversaire ne doit lui donner aucun avantage. De son côté, Alice choisit une clé secrète a , calcule l'exponentiation modulaire $A = g^a \bmod p$ et transmet A à Bob. Ce dernier choisit une clé secrète b et calcule $B = g^b \bmod p$ qu'il transmet à son tour. Alice connaît donc a et $B = g^b \bmod p$ (sans connaître b) et peut donc calculer $B^a \bmod p = g^{ba} \bmod p$. De façon similaire, Bob connaît b et $A = g^a \bmod p$ et calcule $A^b \bmod p = g^{ab} \bmod p$. Ils possèdent maintenant un secret commun : $g^{ab} \bmod p$.

Un des protocoles de cryptographie asymétrique les plus utilisés est le protocole RSA, proposé par Rivest, Shamir et Adleman dans [80]. La sécurité de ce protocole repose sur la factorisation entière : s'il est facile de calculer $n = pq$ connaissant p et q , il est difficile de retrouver ces deux opérandes en connaissant seulement n .

Les différentes arithmétiques que nous présentons par la suite servent essentiellement à ces deux types de cryptosystèmes : ceux qui reposent sur le problème de la factorisation d'entiers, comme RSA ; et ceux qui reposent sur le problème du logarithme discret, dans un groupe multiplicatif comme El Gamal [29] ou dans un groupe additif (cryptosystèmes basés sur les courbes elliptiques).

Pour garantir un niveau de sécurité suffisant face aux algorithmes de cryptanalyse connus, les agences nationales ou internationales émettent des recommandations de sécurité concernant la taille des objets mathématiques à utiliser. Le tableau 2.1 reprend les recommandations européennes du rapport Ecrypt II (2010, [27]).

La première colonne présente le type de menace auquel le protocole peut faire face ainsi que sa pérennité. Le type d'attaquant (organisations ou agences de renseignement) est déterminé par ses ressources en termes de financement et de moyens matériels. Par exemple, les petites organisations

sont définies comme ayant un budget limité (\$10 000) tandis qu'à l'autre extrémité les agences de renseignement disposent de moyens conséquents : un budget de 300 millions de dollars et des ressources matérielles performantes (FPGA¹, ASIC²).

Protection <i>Utilisation</i>	Sécurité (bits)	RSA	Log. Discret	ECC
Attaques temps réel	32	-	-	-
<i>Pour l'authentification seulement</i>	48	480	480	96
	56	640	640	112
Court Terme contre de petites organisations	64	816	816	128
Court Terme contre des organisations moyennes Moyen Terme contre de petites organisations	72	1008	1008	144
Court Terme contre des agences Long Terme contre de petites organisations <i>Moins de deux ans de protection</i>	80	1248	1248	160
Niveau standard <i>Environ 10 ans de protection</i>	96	1776	1776	192
Protection moyen terme <i>Environ 20 ans de protection</i>	112	2432	2432	224
Protection long terme <i>Environ 30 ans de protection</i>	128	3248	3248	256
	160	5312	5312	320
	192	7936	7936	384
Avenir « prévisible » Protection contre les ordinateurs quantiques (omettant l'algorithme de Shor [88])	256	15424	15424	512

TABLE 2.1 – Tailles des clés pour les protocoles utilisant les problèmes de factorisation et de logarithme discret.

La seconde colonne représente le niveau de sécurité estimé en bits : il faudra au minimum 2^s opérations arithmétiques pour venir à bout d'une instance du protocole en utilisant les algorithmes de cryptanalyse connus. Le niveau standard de sécurité établi par Ecrypt II est de 96 bits, soit $\simeq 10^{29}$ opérations arithmétiques pour casser le protocole. Ce niveau de sécurité correspond également à la taille des clés à utiliser pour la partie symétrique des échanges.

Exemple 4. En juin 2011, la machine la plus puissante était capable d'effectuer 8×10^{15} opérations flottantes simple précision par seconde (8 PetaFLOPS). Si l'on simplifie le problème en considérant un algorithme de cryptanalyse totalement parallèle et des opérations simple précision uniquement, il lui faudrait plus de trois cent mille ans pour casser une instance d'un protocole utilisant un tel niveau de sécurité.

La troisième colonne du tableau 2.1 donne la taille du module de chiffrement RSA à utiliser. Les deux dernières colonnes donnent les tailles des groupes dans lesquels le logarithme discret sera défini : taille du sous-groupe multiplicatif du corps $\mathbb{Z}/p\mathbb{Z}$ pour des protocoles comme El Gamal (colonne 4), et taille du groupe des points d'une courbe (colonne 5) pour les protocoles basés sur les courbes elliptiques. Nous détaillons plus loin ce dernier cas en expliquant pourquoi il nécessite des tailles plus réduites que les autres types de cryptographie.

Remarque 3. Les estimations de durée des protections définies dans ce tableau sont basées sur les algorithmes de cryptanalyse actuels. Ces niveaux de sécurité doivent donc être mis à jour de façon régulière pour tenir compte des évolutions dans le domaine cryptanalytique, mais également

1. *Field-programmable gate array* : circuit logique reconfigurable
2. *Application-Specific Integrated Circuit* : circuit intégré spécialisé

de l'évolution du matériel. En particulier, la dernière ligne du tableau est intéressante mais doit être considérée avec précaution : une sécurité de 256 bits permettrait de se protéger contre les ordinateurs quantiques seulement dans le cas où l'algorithme probabiliste de factorisation quantique de Shor [88] ne s'applique pas.

L'évolution des algorithmes de cryptanalyse contraint les algorithmes de chiffrement à utiliser des nombres de plus en plus grands pour garantir un niveau de sécurité suffisant. Les arithmétiques sous-jacentes aux protocoles de cryptographie asymétrique se doivent donc d'être efficaces.

2.2 Arithmétique multiprécision

La cryptographie à clé publique est donc basée sur des fonctions mathématiques qui nécessitent des opérations arithmétiques entre entiers de grandes tailles. Nous présentons dans cette section les principaux algorithmes permettant l'addition, la soustraction et la multiplication d'entiers multiprécision. De nombreux ouvrages traitent de ce sujet, en particulier [18, 56].

2.2.1 Représentations des entiers multiprécision

Un entier multiprécision e est un nombre de taille arbitraire n en base β , habituellement représenté par un tableau de n mots machine, c'est-à-dire avec β égal à la précision des opérations arithmétiques entières disponibles sur l'architecture considérée (en général $\beta = 2^{32}$ ou $\beta = 2^{64}$). Sous forme mathématique, l'entier e s'écrit donc :

$$e = \sum_{i=0}^{n-1} e_i \beta^i \quad \text{avec } 0 \leq e_i < \beta.$$

Remarque 4. D'autres systèmes de représentation sont également utilisés en cryptographie asymétrique. On peut citer en particulier les systèmes de représentation modulaire (RNS) dans lesquels un entier est représenté par l'ensemble de ses restes modulo un ensemble d'entiers premiers entre eux (l'unicité étant garantie par le théorème des restes chinois) ; ou encore les systèmes de représentation adaptés [77].

Notation 1. Dans ce manuscrit, n représente la taille d'un entier en base β et N sa taille en base 2.

Notation 2. Les complexités \mathcal{O} sont exprimées en termes d'opérations sur mots (soit en fonction de n).

2.2.2 Addition et soustraction

Pour additionner deux entiers multiprécision a et b , chaque mot a_i de a est ajouté au mot b_i de b tel que $a_i + b_i = c_i + r_i \beta$ où $0 \leq c_i < \beta$ et $0 \leq r_i \leq 1$ est la retenue de $a_i + b_i$. Dans les méthodes avec propagation de retenue, la valeur r_{i-1} obtenue au pas $i-1$ est ajoutée au pas i tel que $a_i + b_i + r_{i-1} = c_i + r_i \beta$.

Pour la soustraction $a - b$, le résultat $c_i = a_i - b_i$ sera réduit modulo β s'il est négatif et une retenue égale à -1 sera produite. En effet, si $c_i < 0$ alors $c_i = -\beta + |c_i|$. Le résultat c est donc $a - b$ si $a > b$, ou le complément à la base β^n si $b > a$.

Ces deux algorithmes ont une complexité linéaire $\mathcal{O}(n)$, négligeable face à celles de la multiplication multiprécision comme nous allons le voir.

2.2.3 Multiplication

Dans sa version classique, la multiplication multiprécision entre des opérandes de taille n calcule n^2 produits de mots, avec une complexité en $\mathcal{O}(n^2)$, ce qui en fait une opération particulièrement coûteuse par rapport aux opérations de décalage, d'addition ou de soustraction. Baisser cette complexité a été un défi majeur en arithmétique, car la multiplication est au cœur de très nombreuses applications. En 1963, Karatsuba propose dans [52] la première amélioration notable de l'algorithme classique qui

fait passer la complexité à $\mathcal{O}(n^{\log_2(3)})$. Sa méthode a été généralisée par Toom [99] et Cook [22] avec une complexité en $\mathcal{O}(n^{\log_r(2^r-1)})$ où r représente le découpage de chacun des opérandes. En 1971, Schönhage et Strassen proposent dans [84] une multiplication basée sur la transformation de Fourier rapide (FFT pour Fast Fourier Transform), une méthode pour calculer la transformée de Fourier discrète. La complexité de la multiplication passe alors en $\mathcal{O}(n \log n \log \log n)$. On trouvera dans [33] une nouvelle amélioration de la complexité asymptotique de la multiplication multiprécision qui devient $n \log n 2^{\mathcal{O}(\log^* n)}$ où $\log^* n$ représente le logarithme itéré de n . Cependant, cette opération est plus avantageuse que la méthode de Schönhage et Strassen pour des nombres « astronomiquement grands » (voir la conclusion de [33]).

2.2.3.1 Version scolaire

La version classique de la multiplication est appelée multiplication scolaire en référence à la méthode de multiplication enseignée à l'école. Dans cette multiplication, réalisée par l'algorithme 2, tous les mots a_i de a sont multipliés par tous les mots b_j de b , soit au total n^2 produits, dont les résultats sont décalés à gauche de $i + j$ mots et additionnés pour obtenir le résultat final. La complexité de cette multiplication est égale à $\mathcal{O}(n^2)$.

Algorithme 2: Multiplication entière scolaire

Données : $a = (a_{n-1}, \dots, a_0)_\beta$, $b = (b_{n-1}, \dots, b_0)_\beta$
Résultat : $c = ab$

```

1  $c \leftarrow 0$ 
2 pour  $i$  de 0 à  $n - 1$  faire
3   | pour  $j$  de 0 à  $n - 1$  faire
4   | |  $c \leftarrow c + a_i b_j \beta^{i+j}$ 
5   | fin
6 fin
7 retourner  $c$ 

```

2.2.3.2 Karatsuba

La multiplication de Karatsuba est une méthode de type *divide-and-conquer* pour calculer la multiplication multiprécision. Introduite en 1963 par Karatsuba dans [52], elle est la première multiplication à complexité sous-quadratique.

Dans la suite de cette section, nous exprimons les opérandes a et b sous forme de polynômes de degré m à coefficients entiers de la façon suivante :

$$\begin{aligned} \mathcal{A}(x) &= a_m x^m + \dots + a_1 x + a_0 \\ \mathcal{B}(x) &= b_m x^m + \dots + b_1 x + b_0 \end{aligned}$$

en supposant $(m + 1) | n$ tels que $\mathcal{A}(\beta^{n/(m+1)}) = a$ et $\mathcal{B}(\beta^{n/(m+1)}) = b$. Le polynôme $\mathcal{C}(x)$ résultat de la multiplication $\mathcal{A}(x)\mathcal{B}(x)$ est un polynôme de degré $2m$:

$$\mathcal{C}(x) = c_{2m} x^{2m} + \dots + c_m x^m + \dots + c_0.$$

Dans le cas de la multiplication de Karatsuba, on considère des polynômes de degré 1 :

$$\begin{aligned} \mathcal{A}(x) &= a_1 x + a_0 \\ \mathcal{B}(x) &= b_1 x + b_0 \end{aligned}$$

et le polynôme $\mathcal{C}(x)$ est degré 2 :

$$\mathcal{C}(x) = c_2 x^2 + c_1 x + c_0.$$

Dans le cas quadratique, les coefficients c_i sont calculés par :

$$\begin{aligned} c_2 &= a_1 b_1 \\ c_1 &= a_0 b_1 + a_1 b_0 \\ c_0 &= a_0 b_0 \end{aligned}$$

soit quatre multiplications.

Karatsuba a remarqué que seulement 3 multiplications suffisent pour calculer les quatre coefficients. En effet, $(a_0 + a_1)(b_0 + b_1) = a_0b_0 + a_1b_1 + a_0b_1 + a_1b_0$. Or les produits a_0b_0 et a_1b_1 sont calculés pour trouver les coefficients c_0 et c_2 . Le calcul $a_0b_1 + a_1b_0$ peut donc se faire à l'aide d'une seule multiplication :

$$a_0b_1 + a_1b_0 = (a_0 + a_1)(b_0 + b_1) - a_0b_0 - a_1b_1. \quad (2.1)$$

Cette méthode peut être utilisée récursivement pour calculer les nouveaux produits $(n/2) \times (n/2)$ créés. La récursion est stoppée lorsqu'un opérande est de taille 1. En appliquant cette récursion, la méthode de Karatsuba améliore la complexité de la multiplication d'un facteur asymptotique. En effet, si $T(n)$ est le temps pour calculer une multiplication entre opérandes de taille n , on a :

$$T(n) \leq 3T(n/2) + cn/2$$

où $cn/2$ représente le temps consacré aux décalages et aux additions. Si l'on considère $n = 2^k$, on a donc :

$$T(2^k) \leq 3T(2^{k-1}) + c2^{k-1} \leq 3^k c'.$$

Or $3^k = 3^{\log_2(n)} = n^{\log_2(3)}$. La complexité de cet algorithme est donc égale à $\mathcal{O}(n^{\log_2(3)})$.

2.2.3.3 Toom-3

En 1963, Toom [99] ouvre la voie aux méthodes d'évaluation/interpolation, en réalité une généralisation de la méthode de Karatsuba.

L'idée de la multiplication Toom-3 est d'écrire les opérandes sous forme de polynômes de degré 2 :

$$\begin{aligned} \mathcal{A}(x) &= a_2x^2 + a_1x + a_0 \\ \mathcal{B}(x) &= b_2x^2 + b_1x + b_0 \end{aligned}$$

puis d'évaluer ces polynômes en un nombre suffisant de points distincts x_i , multiplier les $\mathcal{A}(x_i)$ par les $\mathcal{B}(x_i)$ et enfin d'interpoler les coefficients du polynôme résultat $\mathcal{C}(x)$ à partir du résultat de ces multiplications.

Théorème 2 (Lagrange). *Il existe un unique polynôme \mathcal{P} univarié de degré d défini par $d+1$ points.*

L'interpolation des coefficients d'un polynôme \mathcal{P} de degré d nécessite de connaître ses valeurs en $d+1$ points distincts, le théorème 2 garantissant son unicité.

Dans la méthode Toom-3, les polynômes $\mathcal{A}(x)$ et $\mathcal{B}(x)$ sont de degré 2, le polynôme $\mathcal{C}(x)$ correspondant à leur multiplication est donc de degré 4 et de la forme : $\mathcal{C}(x) = c_0 + c_1x + c_2x^2 + c_3x^3 + c_4x^4$. Suivant le théorème, il suffit donc de connaître ses valeurs en 5 points pour interpoler ses coefficients. La méthode Toom-3 peut ainsi se résumer à la relation matricielle suivante :

$$\begin{pmatrix} x_0^0 & x_0^1 & x_0^2 & x_0^3 & x_0^4 \\ x_1^0 & x_1^1 & x_1^2 & x_1^3 & x_1^4 \\ x_2^0 & x_2^1 & x_2^2 & x_2^3 & x_2^4 \\ x_3^0 & x_3^1 & x_3^2 & x_3^3 & x_3^4 \\ x_4^0 & x_4^1 & x_4^2 & x_4^3 & x_4^4 \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \\ c_4 \end{pmatrix} = \begin{pmatrix} \mathcal{A}(x_0)\mathcal{B}(x_0) \\ \mathcal{A}(x_1)\mathcal{B}(x_1) \\ \mathcal{A}(x_2)\mathcal{B}(x_2) \\ \mathcal{A}(x_3)\mathcal{B}(x_3) \\ \mathcal{A}(x_4)\mathcal{B}(x_4) \end{pmatrix}$$

L'interpolation des coefficients du polynôme est donc réalisée par la multiplication de l'inverse de la matrice de Vandermonde par le vecteur des $\mathcal{A}(x_i)\mathcal{B}(x_i)$.

En pratique, un choix judicieux des x_i pour l'évaluation de $\mathcal{A}(x_i)$ et $\mathcal{B}(x_i)$ permet de simplifier les calculs nécessaires à la résolution du système linéaire. En particulier, on choisit généralement $x_0 = 0$ et $x_4 = \infty$ ce qui permet de calculer directement les coefficient $c_0 = a_0b_0$ et $c_4 = a_2b_2$ (respectivement). Trois autres points sont communément utilisés : $x_1 = 1$, $x_2 = -1$, et $x_3 = 2$ qui permettent en pratique

de calculer c_1 c_2 et c_3 grâce aux relations suivantes [18] :

$$\begin{aligned}
 d_1 &= (a_0 + a_1 + a_2)(b_0 + b_1 + b_2) \\
 d_{-1} &= (a_0 + a_2 - a_1)(b_0 + b_2 - b_1) \\
 d_2 &= (a_0 + 2a_1 + 4a_2)(b_0 + 2b_1 + 4b_2) \\
 e_1 &= (3c_0 + 2d_{-1} + d_2)/6 - 2c_4 \\
 e_2 &= (d_1 + d_{-1})/2 \\
 c_1 &= d_1 - e_1 \\
 c_2 &= e_2 - c_0 - c_4 \\
 c_3 &= e_1 - e_2
 \end{aligned}$$

Cinq multiplications $a_i b_j$ avec des opérandes de taille $n/3$ (au lieu de neuf dans le cas quadratique) sont nécessaires au calcul, d'où :

$$T(n) \leq 5T(n/3) + cn$$

où cn correspond aux différentes opérations annexes : décalages, additions, multiplications par des constantes et divisions.

Bien entendu, la méthode peut être appliquée récursivement pour les multiplications $a_i b_j$ nécessaires. Si on pose $n = 3^k$, on a :

$$T(3^k) \leq 5T(3^{k-1}) + cn \leq c'5^k.$$

Comme $k = \log_3(n)$, la complexité de la méthode Toom-3 est égale à $\mathcal{O}(n^{\log_3(5)})$.

2.2.3.4 Toom-Cook r

La méthode Toom-Cook r (due à Toom et discutée dans sa thèse par Cook [22]) généralise le schéma évaluation-interpolation : les polynômes représentant les opérandes sont découpés en polynômes de degré $r - 1$ de la forme :

$$\begin{aligned}
 \mathcal{A}(x) &= a_{r-1}x^{(r-1)} + \dots + a_1x + a_0 \\
 \mathcal{B}(x) &= b_{r-1}x^{(r-1)} + \dots + b_1x + b_0.
 \end{aligned}$$

Le schéma de la multiplication Toom-Cook est alors le suivant :

1. évaluation des polynômes $\mathcal{A}(x)$ et $\mathcal{B}(x)$ en $2r - 1$ points distincts x_i ;
2. multiplications $\mathcal{A}(x_i)\mathcal{B}(x_i)$;
3. interpolation des coefficients du polynôme $\mathcal{C}(x)$ à partir des $\mathcal{A}(x_i)\mathcal{B}(x_i)$.

Dans [16], Brodato et Zanoni présentent une méthode pour trouver les points x_i les mieux adaptés à l'interpolation des polynômes.

La complexité de cette multiplication peut être calculée de manière similaire à celle de la méthode Toom-3. Soit $n = r^k$, on a :

$$T(r^k) \leq 2(r - 1)T(r^{k-1}) + cr^{k-1} \leq (2r - 1)^k$$

d'où la complexité de la méthode Toom-Cook r : $\mathcal{O}(n^{\log_r(2r-1)})$.

Remarque 5. La méthode Toom-Cook avec $r = 2$ correspond à la multiplication de Karatsuba.

Les étapes d'évaluation et d'interpolation de cette méthode de multiplication peuvent être améliorées en utilisant la transformation de Fourier rapide comme nous le présentons dans la section suivante.

2.2.3.5 Algorithmes à complexité quasi-linéaire

Les algorithmes de multiplication les plus rapides ont une complexité quasi-linéaire. Ils utilisent la méthode de transformation de Fourier discrète pour calculer rapidement l'évaluation et l'interpolation des polynômes.

Soit un anneau R . Un élément ω est une racine N -ième principale de l'unité si $\omega^N = 1$ et $\sum_{j=0}^{N-1} \omega^{jk} = 0$ pour $1 \leq k \leq N$. On suppose N pair dans la suite. L'idée est d'évaluer les polynômes \mathcal{A} et \mathcal{B} aux racines de l'unité. Cette évaluation peut être réalisée par le calcul de la transformée de Fourier discrète.

Définition 2. La transformée de Fourier discrète N -point sur un anneau R d'un vecteur $a = (a_0, \dots, a_{N-1})$, noté $DFT(a)$, est un vecteur $\tilde{a} = (\tilde{a}_0, \dots, \tilde{a}_{N-1})$ tel que :

$$\tilde{a}_j = \sum_{k=0}^{N-1} \omega^{jk} a_k.$$

La transformation de Fourier rapide (FFT) est une méthode de type *divide-and-conquer* pour calculer efficacement $DFT(a)$ en $\mathcal{O}(n \log n)$ opérations. Elle calcule récursivement les transformées de Fourier $(N/2)$ -points des coefficients pairs et les transformées de Fourier $(N/2)$ -points des coefficients impairs. En effet, si l'on pose :

$$\begin{aligned} \mathcal{A}_1(x) &= a_0 + a_2x + a_4x^2 + \dots + a_{N-2}x^{N/2-1} \\ \mathcal{A}_2(x) &= a_1 + a_3x + a_5x^2 + \dots + a_{N-1}x^{N/2-1} \end{aligned}$$

on a $\mathcal{A} = \mathcal{A}_1(x^2) + x\mathcal{A}_2(x^2)$. Pour évaluer \mathcal{A} aux points $\omega^0, \dots, \omega^{N-1}$ il suffit donc d'évaluer \mathcal{A}_1 et \mathcal{A}_2 aux points $(\omega^0)^2, \dots, (\omega^{N-1})^2$ puis de les ajouter suivant l'équation précédente. Or, comme $\omega^{N+k} = \omega^k$ par définition, cela revient à évaluer \mathcal{A}_1 et \mathcal{A}_2 en $N/2$ points $\omega^0, \omega^2, \dots, \omega^{N-2}$. Récursivement, on peut donc appliquer la même approche pour \mathcal{A}_1 et \mathcal{A}_2 en calculant $DFT(a_{pair})$ et $DFT(a_{impair})$ de taille $N/2$. Les algorithmes de transformation $DFT(a)$ et $DFT(a)^{-1}$ sont donnés dans [18].

On peut schématiquement présenter la méthode de multiplication FFT comme ceci :

- (1) Calcul de $\tilde{a} = DFT(a)$ et $\tilde{b} = DFT(b)$;
- (2) Multiplication point à point $\tilde{c}_j = \tilde{a}_j \tilde{b}_j$;
- (3) Calcul de $c = DFT(\tilde{c})^{-1}$.

Reste à trouver des anneaux dans lesquels il y a « assez » de racines principales de l'unité. Ce schéma de multiplication a été présenté pour la première en 1971 par Schönhage et Strassen dans [84]. Leur multiplication utilise l'anneau quotient $\mathbb{Z}/(2^l+1)\mathbb{Z}$ et calcule $ab \bmod 2^l + 1$ en $\mathcal{O}(n \log n \log \log n)$ par une série d'opérations modulo un nombre de type $2^l + 1$, en particulier pour la multiplication point à point à l'étape 2.

Dans [37], Gaudry, Kruppa et Zimmermann présentent une implémentation de cet algorithme pour la bibliothèque de calcul multiprécision GMP. La complexité asymptotique de la multiplication a été améliorée en 2007 par Fürer dans [33]. Les techniques de multiplication rapide et leurs applications sont discutées dans [6].

2.2.4 Division

Nous n'utilisons pas la division entière a/b dans la suite de ce manuscrit. Nous présentons cependant brièvement cette opération ainsi que sa complexité en termes d'opérations entre mots. On considère deux entiers multiprécision a et b respectivement de taille $t+n$ et n . La division retourne le quotient q et le reste r tels que $a = qb + r$.

La version classique de la division consiste à obtenir successivement les t mots q_j du quotient partiel en calculant pour j de $t - 1$ à 0 :

$$\begin{aligned} q_j &\leftarrow \lfloor (a_j\beta + a_{j-1})/b_{n-1} \rfloor \\ q_j &\leftarrow \min(q_j, \beta - 1) \\ a &\leftarrow a - q_j\beta^j b. \end{aligned}$$

En réalité, l'entier q_j doit être corrigé car il est potentiellement supérieur au mot quotient exact. Cela signifie que l'entier a obtenu est inférieur à 0 . On ajoute à a la quantité $i\beta^j b$ tel que $a + (i - 1)\beta^j b \leq 0 \leq a + i\beta^j b$. Le mot exact du quotient est alors $q_j - i$. Le reste de la division est le dernier entier a obtenu. Cette méthode a une complexité égale à $\mathcal{O}(n(t + 1))$.

Il existe des algorithmes plus rapides, décrits dans [18]. On peut noter en particulier un algorithme de type *Divide-and-conquer* qui calcule récursivement tous les mots du quotient et du reste en divisant le calcul de chacune des étapes de la récursion en deux parties : calcul récursif des poids forts du quotient (et du reste) puis calcul récursif des poids faibles du quotient (et du reste). Lorsque les opérandes des calculs récursifs sont suffisamment petits, l'algorithme effectue une division classique.

Selon [18], cet algorithme nécessite $D(n + t, n) = D(n, n - t/2) + 2M(n/2) + \mathcal{O}(n)$ opérations de mots. Asymptotiquement, les complexités des multiplications et des divisions multiprécision sont donc proches, la complexité de la division étant $\mathcal{O}(M(t + n, n))$, avec un facteur entre ces deux opérations de l'ordre de 2.

En pratique cependant, la division est bien plus coûteuse que la multiplication, en particulier car l'opération atomique de la division multiprécision est la division matérielle, plus coûteuse que la multiplication matérielle [30, 4].

2.2.5 Résumé des complexités

La complexité des algorithmes de multiplication donnée en $\mathcal{O}(\alpha n^\beta)$ dépend de la méthode utilisée. De manière générale, nous notons $M(n)$ la complexité pour calculer une multiplication multiprécision entre entier de taille n (nous notons $M(n, m)$ si les opérandes sont de tailles différentes). Le tableau 2.2 donne les notations ainsi que les complexités des différentes opérations arithmétiques multiprécision entre opérandes de taille n .

Opération	Notation	Complexité
Addition	$A(n)$	$\mathcal{O}(n)$
Soustraction	$S(n)$	$\mathcal{O}(n)$
Multiplication	$M(n)$	
Scolaire		$\mathcal{O}(n^2)$
Karatsuba		$\mathcal{O}(n^{\log_2(3)})$
Toom-Cook r -way		$\mathcal{O}(n^{\log_r(2r-1)})$
Schönhage-Strassen		$\mathcal{O}(n \log n \log \log n)$
Division	$D(m, n)$	$\mathcal{O}(M(m, n))$

TABLE 2.2 – Résumé des complexités des opérations arithmétiques entières

2.2.6 L'arithmétique multiprécision en pratique

Dans la suite de ce manuscrit, nous nous appuyons sur la bibliothèque d'arithmétique multiprécision GMP³ (GNU MultiPrecision). Cette bibliothèque utilisable en langage C et C++ embarque les algorithmes de multiplication que nous avons décrits dans les parties précédentes. Suivant la taille des entiers considérés, les algorithmes de multiplication changent en fonction de seuils dépendants de l'architecture considérée. La plupart de nos expérimentations ont été réalisées sur les nœuds de la

3. <http://gmplib.org/>

Intervalle (en mots machine)	Méthode
0-17	Scolaire
18-64	Karatsuba (Toom-2)
65-165	Toom-3
166-3711	Toom-Cook r pour différents r
3711-	FFT

TABLE 2.3 – Intervalles de tailles (en nombre de mots) pour les différentes méthodes de multiplication

plate-forme HPC@LR⁴ qui sont composés de deux processeurs Inter Westmere à six cœurs 64 bits cadencés à 2,13GHz. Le tableau 2.3 donne les seuils de la bibliothèque pour cette architecture.

La figure 2.1 présente l'accélération de la multiplication GMP par rapport à la méthode de multiplication scolaire pour des entiers entre 2^{10} et 2^{18} bits. Cette figure illustre les complexités théoriques que nous avons précédemment données pour les méthodes de multiplication. On peut noter l'accélération croissante en fonction des algorithmes pour les tailles considérées. Le changement d'algorithme le plus visible est celui du passage de l'algorithme de Toom-Cook à la FFT pour des entiers de très grandes tailles (à partir de 2^{17} bits).

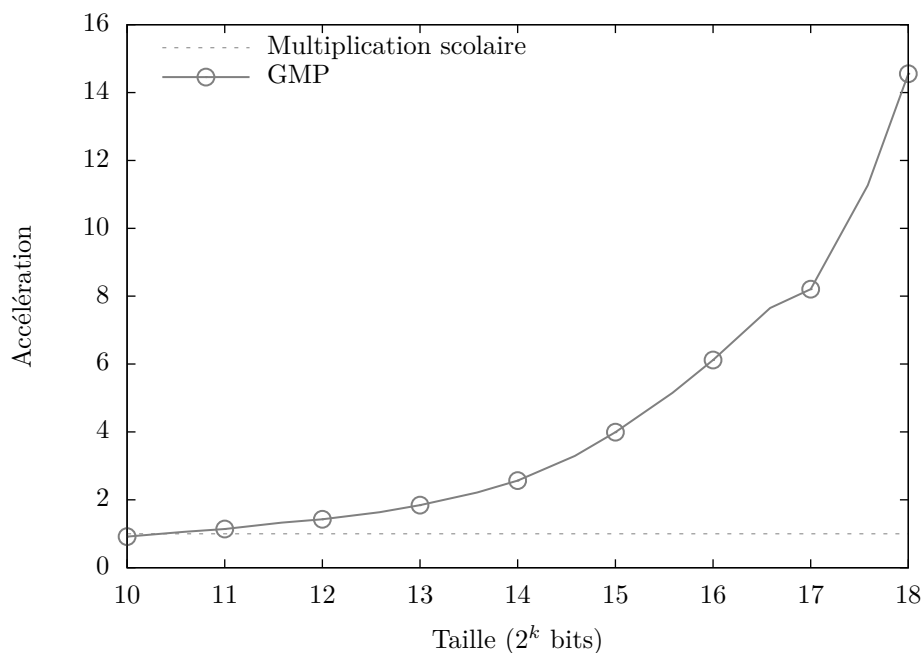


FIGURE 2.1 – Accélération de la multiplication GMP par rapport à la multiplication quadratique

2.3 Arithmétique modulaire

Les opérations arithmétiques des cryptosystèmes asymétriques sont généralement effectuées dans des structures mathématiques finies : groupes, anneaux ou corps. Nous présentons dans ce chapitre les algorithmes classiques pour l'addition, la soustraction et la multiplication modulaires. Nous considérons dans la suite un entier multiprécision p fixé de taille n en base β . Les opérations d'arithmétique modulaire sont définies dans l'anneau quotient $\mathbb{Z}/p\mathbb{Z}$ entre éléments de cet anneau. Lorsque p est premier, $\mathbb{Z}/p\mathbb{Z}$ est un corps fini que nous noterons \mathbb{F}_p . Dans la suite, nous représentons les éléments de $\mathbb{Z}/p\mathbb{Z}$ par des entiers dans $[0, p - 1]$.

4. <http://www.hpc-lr.univ-montp2.fr/>

Une opération modulaire consiste à calculer le reste de la division euclidienne du résultat entier de l'opération par p pour garantir que le résultat final appartient à l'intervalle $[0, p - 1]$, c'est-à-dire : $\forall \otimes \in \{+, -, \times, /\}$:

$$\otimes : [0, p - 1] \times [0, p - 1] \rightarrow [0, p - 1]$$

$$a, b \mapsto a \otimes b - qp \quad \text{avec } q = \left\lfloor \frac{a \otimes b}{p} \right\rfloor.$$

Dans le cas d'une addition la division est inutile car si $a, b < p$ alors $a + b < 2p$. La réduction est ainsi réalisée par une seule soustraction.

La multiplication modulaire est une opération bien plus coûteuse en temps de calcul, car $ab < p^2$ et donc q est généralement de la taille de p . Si une division suffit pour calculer ce quotient, son coût élevé en temps de calcul par rapport à la multiplication a conduit au développement de méthodes de calcul bien plus efficaces ; d'autant plus que la multiplication modulaire reste l'opération de bas niveau la plus coûteuse dans les protocoles de cryptographie asymétrique.

Deux types d'algorithmes peuvent être définis pour la calculer : ceux qui entrelacent la multiplication et la réduction ; et ceux qui calculent la multiplication complète, puis la réduction modulaire. Certaines méthodes pourront être exécutées par des algorithmes des deux types. L'avantage des algorithmes par entrelacement est la minimisation de la taille de la mémoire nécessaire à la multiplication modulaire. Ces algorithmes sont préférés pour les architectures où peu de mémoire est disponible, comme les architectures embarquées ou les cartes graphiques. Cependant, ces algorithmes sont constitués de multiplications en précision finie (multiplications entre mots), ou fortement déséquilibrée (un des opérandes est bien plus grand que le second) et utilisent généralement un schéma de multiplication quadratique. Réaliser la multiplication complète $c = ab$ puis la réduction $c \bmod p$ permet d'utiliser ces algorithmes de multiplication à complexité sous-quadratique, mais certains résultats intermédiaires doivent être conservés, avec une utilisation de la mémoire en conséquence.

2.3.1 Addition et soustraction

L'addition et la soustraction sont les deux opérations modulaires les plus simples. Soient $a < p$ et $b < p$, alors :

$$a + b \bmod p = \begin{cases} a + b & \text{si } a + b < p \\ a + b - p & \text{sinon} \end{cases} \quad \text{et} \quad a - b \bmod p = \begin{cases} a - b & \text{si } a - b \geq 0 \\ a - b + p & \text{sinon.} \end{cases}$$

Ces deux opérations ont une complexité égale à $\mathcal{O}(n)$.

2.3.2 Multiplication

De nombreux algorithmes ont été proposés pour accélérer la multiplication modulaire. Nous présentons ici les principaux ainsi que leur complexité.

2.3.2.1 L'algorithme classique

La méthode classique pour réduire le résultat d'une multiplication est d'effectuer une division euclidienne : si $c = ab$, $q = \lfloor c/p \rfloor$ et le résultat réduit est obtenu en calculant $r = c - qp$. La complexité de cette opération est égale à $2M(n) + D(2n, n)$. À cause du coût élevé de la division entière multiprécision, cette algorithme est rarement utilisé en pratique au niveau logiciel.

Remarque 6. Cette méthode peut également s'écrire sous forme de multiplications-réductions entrelacées, permettant des calculs en place.

2.3.2.2 Multiplication de Blakely

Un des algorithmes de réduction les plus simples a été introduit par Blakely en 1983 dans [12]. L'algorithme suit un schéma de multiplication de type *Doublement-et-addition*. Soit (a_{N-1}, \dots, a_0) la représentation binaire de l'opérande a . Calculer ab revient à calculer :

$$ab = 2(\dots(2(a_{N-1}b) + a_{N-2}) + \dots) + a_0b.$$

Cela revient à un parcours poids forts - poids faibles, où l'on multiplie le résultat partiel par 2 à chaque bit a_i . De plus, si $a_i = 1$, on ajoute b . L'algorithme effectue une réduction à chaque étape : le résultat partiel r n'est jamais supérieur à $2p$, les réductions consistent donc en une série de soustractions.

Algorithme 3: Multiplication Modulaire de Blakely

Données : $p < 2^n$, $a, b < p$ avec $a = (a_{N-1}, \dots, a_0)_2$
Résultat : $ab \bmod p$

- 1 $r \leftarrow 0$
- 2 **pour** i de $N - 1$ à 0 **faire**
- 3 $r \leftarrow 2r$
- 4 **si** $r \geq p$ **alors** $r \leftarrow r - p$
- 5 $r \leftarrow r + a_i b$
- 6 **si** $r \geq p$ **alors** $r \leftarrow r - p$
- 7 **fin**
- 8 **retourner** r

L'algorithme 3 calcule $ab \bmod P$ en au plus $3N$ additions de N bits soit une complexité égale à $3NA(N)$. En pratique, cet algorithme est très peu utilisé car des algorithmes plus performants existent, comme nous allons le voir dans la suite.

2.3.2.3 Multiplication de Montgomery

En 1985, Montgomery a introduit une méthode très efficace pour effectuer la multiplication modulaire avec un p fixé [67] en définissant un nouveau système de représentation des entiers.

Définition 3. La représentation de Montgomery de x est l'entier $\bar{x} = x\beta^n \bmod p$.

La représentation de Montgomery reste stable par l'addition. En effet, $\bar{a} \pm \bar{b} \bmod p = a\beta^n \pm b\beta^n = (a \pm b)\beta^n \bmod p$. De plus, Montgomery a défini une multiplication telle que :

$$\begin{aligned} \bar{a} \times_{Montg} \bar{b} &= \bar{a}\bar{b}\beta^{-n} \bmod p \\ &= a\beta^n b\beta^n \beta^{-n} \bmod p \\ &= ab\beta^n \bmod p \\ &= \bar{ab} \bmod p. \end{aligned}$$

Cette multiplication est réalisée par l'algorithme 4 qui calcule le plus petit entier q tel que $ab + qp$ est un multiple de β^n . Donc $r = (ab + qp)/\beta^n$ est un entier (inférieur à $2p$) tel que $r \equiv ab\beta^{-n} \pmod{p}$.

Algorithme 4: Multiplication modulaire de Montgomery

Données : $p < \beta^n$, $p \wedge \beta^n = 1$, $a, b < p$
Résultat : $r \equiv ab\beta^{-n} \bmod p$ avec $r < p$

- 1 $\mu \leftarrow -1/p \bmod \beta^n$ ▷ Pré-calcul
- 2 $c \leftarrow a \times b$
- 3 $q \leftarrow \mu \times c \bmod \beta^n$
- 4 $r \leftarrow (c + q \times p)/\beta^n$
- 5 **si** $r \geq p$ **alors** $r \leftarrow r - p$
- 6 **retourner** r

Théorème 3. L'algorithme de Montgomery calcule correctement $ab\beta^{-n} \bmod p$.

Démonstration. Comme $q = \mu ab = -p^{-1}ab \bmod \beta^n$, on a $ab + qp \equiv 0 \pmod{\beta^n}$. La division à la ligne 4 est donc exacte et $r \equiv ab\beta^{-n} \pmod{p}$. Comme $0 \leq ab < p^2$ et $0 \leq q < \beta^n$, on a $0 \leq ab + qp < p^2 + \beta^n p < p\beta^n + \beta^n p$. La division par β^n donne donc $0 \leq r < 2p$. Une seule soustraction suffit pour garantir $r < p$. \square

Cette multiplication modulaire nécessite l'inversion $1/p \bmod \beta^n$ qui est particulièrement coûteuse. Elle sera utilisée dans le cas où un grand nombre de multiplications modulo un même nombre p seront nécessaires, comme par exemple pour l'exponentiation modulaire.

Théorème 4. *Étant donnés a, b deux entiers inférieurs à p , $p < \beta^n$, l'algorithme 4 calcule correctement $ab\beta^{-n} \bmod p$ en $3M(n)$.*

Démonstration. Les opérandes a et b étant de taille n , la multiplication de la ligne 2 est $M(n)$. De même, comme le quotient q est de taille n (car résultat d'une opération modulo β^n avec des opérandes de taille au moins égale à n) et p de taille n par définition, le calcul de qp se fait en $M(n)$. Le calcul du quotient se fait également en $M(n)$, car bien que c soit de taille $2n$, le résultat de la multiplication étant réduit modulo n , il suffit de multiplier les n mots de poids faible de c par les n mots de poids faibles de μ et de réduire le résultat modulo β^n . \square

2.3.2.4 Méthode FIOS

En calculant des produits complets, l'algorithme 4 permet l'utilisation des multiplications rapides présentées précédemment, mais néglige un aspect qui sera important pour nous par la suite : la mémoire. En effet, tel quel, l'algorithme nécessite deux zones mémoires importantes : une pour stocker $c = ab$, de $2n$ mots si a et b sont de taille n et une seconde pour stocker q puis qp .

Cependant, certaines architectures particulières disposent de très peu de mémoire, comme les architectures embarquées ou les accélérateurs matériels. Sur de tels supports, une version *en place* de la multiplication de Montgomery est préférable. Il existe de nombreuses variantes de l'algorithme, présentées et analysées dans [58].

La version que nous présentons ici, appelée *Finely Integrated Operand Scanning (FIOS) Method*, nécessite seulement trois registres de taille β (d , s et q) pour calculer un produit de Montgomery complet.

Algorithme 5: Multiplication modulaire de Montgomery (FIOS)

```

Données :  $p < \beta^n$ ,  $a, b < p$ 
Résultat :  $r \equiv ab \bmod p$  avec  $r < 2p$ 
1  $\mu_0 \leftarrow -1/p \bmod \beta$  ▷ Pré-calcul
2  $r \leftarrow 0$ 
3 pour  $i$  de 0 à  $n - 1$  faire
4    $(d, s) \leftarrow r_0 + a_0 \times b_i$ 
5    $r_1 \leftarrow r_1 + d$ 
6    $q \leftarrow s \times \mu_0 \bmod \beta$  ▷ Calcul du quotient partiel
7    $(d, s) \leftarrow s + q \times p_0$  ▷ Calcul de  $qp$ 
8   pour 1 de 0 à  $n - 1$  faire
9      $(d, s) \leftarrow r_j + a_j \times b_i + d$ 
10     $r_{j+1} \leftarrow r_{j+1} + d$ 
11     $(d, s) \leftarrow s + q \times p_j$ 
12     $r_{j-1} \leftarrow s$ 
13   fin
14 fin
15  $(d, s) \leftarrow r_n + s$ 
16  $r_{n-1} \leftarrow s$ 
17  $r_n \leftarrow d$ 

```

Cet algorithme calcule la multiplication de Montgomery en $2n^2 + n$ multiplications de mots.

Remarque 7. Cet algorithme *en place* utilise un schéma de multiplication quadratique. Trouver un algorithme pour réaliser une multiplication de Montgomery sous-quadratique *en place* est un problème ouvert.

2.3.2.5 Multiplication modulaire de Barrett

Dans [5], Barrett a introduit une méthode pour calculer la multiplication modulaire sans division, particulièrement efficace pour effectuer un grand nombre d'opérations avec un même modulo p . Cet algorithme donne une approximation du quotient de la division ab/p avec une seule multiplication de

la manière suivante :

$$q = \left\lfloor \frac{ab}{p} \right\rfloor$$

$$q \approx \left\lfloor \frac{\left\lfloor \frac{C}{\beta^n} \right\rfloor \times \left\lfloor \frac{\beta^{2n}}{p} \right\rfloor}{\beta^n} \right\rfloor = q'$$

les divisions par β^n correspondant à de simples décalages de mots.

Algorithme 6: Multiplication modulaire de Barrett

Données : $\beta^n/2 \leq p < \beta^n$, $a, b < p$

Résultat : $ab \bmod p$

1 $\nu \leftarrow \beta^{2n}/p$

▷ Pré-calcul

2 $c \leftarrow ab$

3 $q \leftarrow \left\lfloor \frac{\lfloor c/\beta^n \rfloor \nu}{\beta^n} \right\rfloor$

4 $r \leftarrow c - q \times p$

5 **tant que** $r > p$ **faire** $r \leftarrow r - p$

6 **retourner** r

Théorème 5. *L'algorithme 6 est correct et la soustraction de la ligne 5 sera réalisée au plus trois fois.*

Démonstration. (D'après [18]) Comme la seule opération réalisée sur r est la soustraction de multiples de p , on a $r \equiv c \bmod p$. Il nous faut prouver $r < p$. Soit $c = c_1\beta^n + c_0$. On a $\beta^{n-1} < p < \beta^n \Rightarrow \beta^n < \beta^{2n}/p < \beta^{n+1}$ d'où

$$q \leq \frac{c_1\nu}{\beta^n} \leq \frac{c}{\beta^n} \frac{\beta^n}{p} \leq \frac{c}{p}$$

ce qui garantit $r > 0$.

De plus, $\nu > \beta^{2n}/p - 1$, donc $\nu p > \beta^{2n} - p$ et $q > c_1\nu/\beta^n - 1$. On a donc $\beta^n qp > c_1\nu p - \beta^n p > c_1(\beta^{2n} - p) - \beta^n p$. Comme $c = c_1\beta^n + c_0$, $c_1(\beta^{2n} - p) - \beta^n p = \beta^n(c - c_0) - p(\beta^n + c_1)$. Or par définition, $c_0 < \beta^n < 2p$, d'où : $\beta^n(c - c_0) > \beta^n c - 2\beta^n p$ et $c_1 < \beta^n$ d'où $-p(\beta^n + c_1) > -2\beta^n p$. Donc

$$\beta^n qp > \beta^n(c - c_0) - p(\beta^n + c_1) > \beta^n c - 4\beta^n p = \beta^n(c - 4p)$$

d'où $p(q+4) > c$. Il faut donc au plus trois soustractions à la fin de l'algorithme pour avoir $r < p$. □

Remarque 8. Suivant [65], en remplaçant la ligne 3 de l'algorithme 6 par

$$q \leftarrow \left\lfloor \frac{\lfloor c/\beta^{n-1} \rfloor \nu}{\beta^{n+1}} \right\rfloor$$

il y a au plus deux soustractions à la fin de l'algorithme pour avoir $r < p$. En effet, en suivant une démonstration similaire à la précédente, on a :

$$\beta^{n+1} qp > c_1\nu p - \beta^{n+1} p > c_1(\beta^{2n} - p) - \beta^{n+1} p.$$

En posant $c = c_1\beta^{n-1} + c_0$ avec $c_1 < \beta^{n+1}$ et $c_0 < \beta^{n-1}$, on a $c_1(\beta^{2n} - p) - \beta^{n+1} p = \beta^{n+1}(c - c_0) - p(\beta^{n+1} + c_1)$. Or $\beta^{n+1}(c - c_0) > \beta^{n+1}c - \beta^{n+1}$ car $c < \beta^{n-1} < p$ et $-p(\beta^{n+1} + c_1) > -2\beta^{n+1}p$. D'où $\beta^{n+1} qp > \beta^{n+1}(c - p) - 2\beta^{n+1}p$ et finalement $p(q+3) > c$.

Théorème 6. *La complexité de l'algorithme 6 est égale à $3M(n)$.*

Démonstration. La multiplication ab se fait en $M(n)$ et produit un résultat de taille $2n$. Le calcul du quotient est réalisé sans perte de précision par la multiplication des n mots de poids fort de ab par les n mots de poids fort de la constante, soit en $M(n)$. Enfin, le quotient étant de taille n , la multiplication qp est réalisée en $M(n)$. □

2.3.2.6 Multiplication modulaire bipartite

La multiplication de Montgomery $C = ab\beta^{-n} \pmod p$ nécessite trois multiplications entre entiers de taille n . Dans [49], Kaihara et Takagi ont proposé une méthode pour calculer $C = ab\beta^{-n/2}$ qui permet de réduire la complexité du calcul en utilisant deux processus indépendants. En effet :

$$\begin{aligned} C &= ab\beta^{-n/2} \pmod p \text{ (avec } b = b_1\beta^{n/2} + b_0) \\ &= a(b_1\beta^{n/2} + b_0)\beta^{-n/2} \pmod p \\ &= (ab_1 \pmod p + ab_0\beta^{-n/2} \pmod p) \pmod p \end{aligned} \tag{2.2}$$

Dans la suite de ce manuscrit, on supposera n pair.

La multiplication modulaire $ab_0\beta^{-n/2} \pmod p$ est calculée nativement par l'algorithme de Montgomery avec une complexité en $2M(n, n/2) + M(n/2)$. En effet, le calcul ab_0 s'effectue en $M(n, n/2)$ car b_0 est de taille $n/2$. Le quotient de Montgomery ($ab_0\mu \pmod{\beta^{n/2}}$) est calculé sans perte de précision par la multiplication des $n/2$ mots de poids faible de ab_0 par les $n/2$ mots de poids fort de la constante μ , soit en $M(n/2)$. Enfin, le quotient étant de taille $n/2$, le calcul qp s'effectue en $M(n, n/2)$.

La multiplication modulaire $ab_1 \pmod p$ est réalisée avec l'algorithme de Barrett avec une complexité en $2M(n, n/2) + M(n/2)$. En effet, le calcul ab_1 s'effectue en $M(n, n/2)$ car b_1 est de taille $n/2$. Le quotient de Barrett est calculé avec la multiplication des $n/2$ mots de poids fort de ab_1 par les $n/2$ mots de poids fort de la constante ν avec une perte de précision possible sur le bit de poids faible qui pourra être compensée par une soustraction finale en plus. Enfin, le quotient étant de taille $n/2$, la multiplication qp sera réalisée en $M(n, n/2)$.

La complexité de cette méthode est donc égale à $4M(n, n/2) + 2M(n/2)$. Cependant, ces deux calculs étant indépendants, ils pourront être réalisés de façon concurrente sur une architecture parallèle avec une complexité égale à $2M(n, n/2) + M(n/2)$.

2.4 Arithmétique des courbes elliptiques

Les courbes elliptiques tendent aujourd'hui à être de plus en plus utilisées en cryptographie asymétrique car elles nécessitent des objets mathématiques de taille plus petite que les autres types de cryptosystèmes. La sécurité de ces protocoles est basée sur le problème du logarithme discret dans le groupe des points d'une courbe elliptique, avec comme conséquence une opération majeure à calculer : la multiplication scalaire $[k]P = P + \dots + P$ (k fois) avec $k \in \mathbb{N}^*$ et P un point d'une courbe. Nous présentons ici les courbes elliptiques et donnons les opérations arithmétiques nécessaires au calcul de $[k]P$.

Nous nous appuyons sur les deux ouvrages de référence pour ce type de cryptographie [43, 21].

2.4.1 La cryptographie basée sur les courbes elliptiques

Dans un groupe $(G, +)$, si g est un générateur de G d'ordre o ($o.g = e$ où e est l'élément neutre de G), l'ensemble noté $\langle g \rangle$ des éléments ig avec $0 \leq i \leq o - 1$ forme un sous-groupe cyclique de G engendré par g . Dans ce contexte, le logarithme discret d'un élément h de $\langle g \rangle$ est l'unique entier $k \in [0, o - 1]$ tel que $h = kg$. Le problème du logarithme discret est alors ainsi formulé : *connaissant* $(G, +)$, g et h , *trouver* k .

Il existe des algorithmes dits *génériques* permettant de le résoudre dans un groupe G quelconque. On peut citer en particulier l'algorithme *Baby-step Giant-step* de Shanks [87] ou l'algorithme Pollard-Rho [78]. Shoup a montré dans [89] que si l est le plus grand entier divisant l'ordre du groupe, ces algorithmes nécessitent $\mathcal{O}(\sqrt{l})$ opérations de groupe.

Pour être efficace, un cryptosystème basé sur ce problème doit donc réunir deux conditions : une loi de groupe $+$ simple, c'est-à-dire calculable en temps polynomial ; et un entier l assez *grand* rendant la résolution du problème impossible en temps polynomial.

En 1985, Miller [66] et Koblitz [57] ont indépendamment introduit la cryptographie basée sur les courbes elliptiques en démontrant que le groupe formé par les points d'une courbe réunit ces deux conditions. En effet, ce groupe est un groupe abélien additif où la loi est exprimée comme un ensemble d'opérations dans le corps de définition de la courbe et est donc facilement calculable. L'opération kg correspond alors à la multiplication d'un point de la courbe par un scalaire k , appelée multiplication scalaire et notée $[k]P$.

Pour se prémunir des attaques par les algorithmes génériques, il faut cependant trouver des groupes d'ordre $o = cl$ où c est petit et l un grand nombre premier. On sait compter les points d'une courbe elliptique définie sur un corps premier, notamment par l'algorithme de Schoof [85]. On peut donc choisir une courbe elliptique, trouver son nombre de points et vérifier qu'il est premier ou presque premier (cofacteur c petit).

Théorème 7 (Hasse-Weil). *Soit E une courbe elliptique définie sur \mathbb{K} et q le nombre d'éléments de \mathbb{K} , alors*

$$q + 1 - 2\sqrt{q} \leq \#E(\mathbb{K}) \leq q + 1 + 2\sqrt{q}.$$

Une démonstration de ce théorème est donnée dans [90], chapitre 5.

Pour une courbe elliptique définie sur \mathbb{F}_p , ce théorème nous dit que le nombre de points est de l'ordre de p . Comme les attaques génériques sont réalisées en \sqrt{l} opérations, pour une sécurité de s bits, la courbe elliptique doit donc être définie sur un corps à $p > 2^{2s}$ éléments.

Remarque 9. Les meilleurs algorithmes de cryptanalyse connus à ce jour pour casser le problème du logarithme discret dans le groupe des points d'une courbe elliptique sont les algorithmes génériques. Des algorithmes plus performants existent dans le cas du sous-groupe multiplicatif d'un corps fini, ce qui explique les différences des tailles entre ces deux types de cryptosystèmes dans le tableau 2.1.

Différents cryptosystèmes reposant sur les courbes elliptiques sont aujourd'hui utilisés et sont décrits dans [11] : des protocoles de signature comme ECDSA [48] ; des protocoles de chiffrement comme ECIES [19] ; ainsi que des protocoles d'échange de clés comme ECMQV [1] et ECDH [19].

2.4.2 Définition des courbes elliptiques

Pour définir les opérations arithmétiques nécessaires à la multiplication scalaire, nous présentons ici les courbes elliptiques. Une définition mathématique formelle de ces objets est donnée dans [90].

Une courbe elliptique définie sur un corps \mathbb{K} est une courbe algébrique non-singulière définie par une équation de Weierstrass :

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6, \tag{2.3}$$

avec $a_1, a_2, a_3, a_4, a_6 \in \mathbb{K}$ et $\Delta \neq 0$, où Δ est le discriminant de la courbe calculé par les équations suivantes :

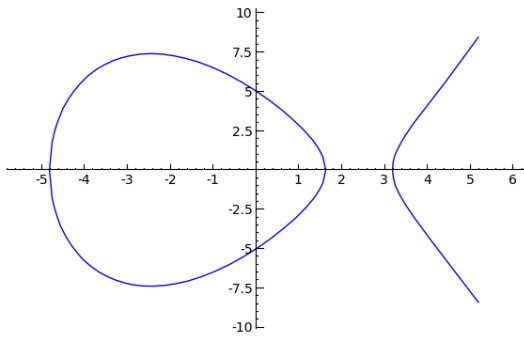
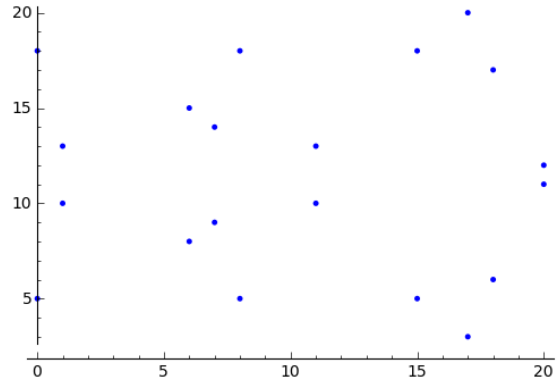
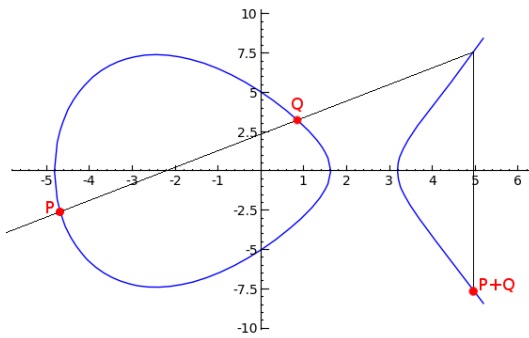
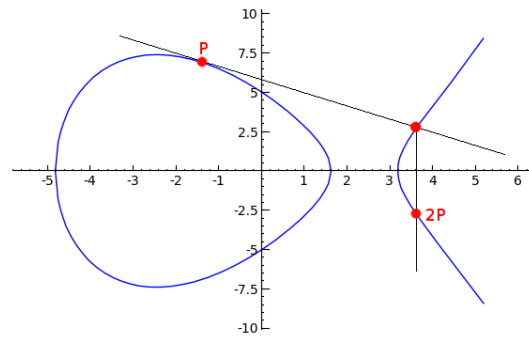
$$\begin{aligned} \Delta &= -b_2^2b_8 - 8b_4^3 - 27b_6^2 + 9b_2b_4b_6 \\ b_2 &= a_1^2 + 4a_2 \\ b_4 &= 2a_4 + a_1a_3 \\ b_6 &= a_3^2 + 4a_6 \\ b_8 &= a_1^2a_6 + 4a_2a_6 - a_1a_3a_4 + a_2a_3^2 - a_4^2 \end{aligned}$$

L'ensemble $E(\mathbb{K}) = \{(x, y) \in \mathbb{K} : y^2 = x^3 + ax + b\}$ auquel on ajoute un point à l'infini noté ∞ est un groupe additif avec ∞ comme élément neutre. La figure 2.2 représente la courbe elliptique définie par l'équation $y^2 = x^3 - 18x + 25$ sur \mathbb{R} . La figure 2.3 illustre une courbe définie sur le corps fini \mathbb{F}_{23} d'équation $y^2 = x^3 + 5x + 2$.

Les figures 2.4 et 2.5 illustrent l'interprétation géométrique (méthode dite de *la corde et de la tangente*) de la loi du groupe dans les cas où $P = Q$ et $P \neq Q$.

Dans la suite de ce manuscrit, nous nous concentrerons sur les courbes définies sur les corps finis premiers \mathbb{F}_p , où $p > 3$. Dans ce cas, l'équation de Weierstrass peut s'écrire de manière simplifiée :

$$y^2 = x^3 + ax + b, \text{ avec } \Delta = -16(4a^3 + 27b^2) \neq 0. \tag{2.4}$$


 FIGURE 2.2 – $E(\mathbb{R}) : y^2 = x^3 - 18x + 25$

 FIGURE 2.3 – $E(\mathbb{F}_{23}) : y^2 = x^3 + 5x + 2$

 FIGURE 2.4 – Addition de deux points sur \mathbb{R}

 FIGURE 2.5 – Doublement de point sur \mathbb{R}

L'opposé d'un point $P = (x, y) \in E(\mathbb{F}_p)$ est le point $-P = (x, -y) \in E(\mathbb{F}_p)$. Les formules d'addition et de doublement dans $E(\mathbb{F}_p)$ sont les suivantes. Soient $P, Q \in E(\mathbb{F}_p)$ où $P = (x_P, y_P)$ et $Q = (x_Q, y_Q)$:

$$R = P + Q = \begin{cases} x_R = \lambda^2 - x_P - x_Q, \\ y_R = \lambda(x_P - x_R) - y_P \end{cases} \quad \text{avec } \lambda = \begin{cases} \left(\frac{y_Q - y_P}{x_Q - x_P} \right) & \text{si } P \neq Q \\ \left(\frac{3x_P^2 + a}{2y_P} \right) & \text{si } P = Q \end{cases} \quad (2.5)$$

2.4.3 Projections et équations de courbes

Ces formules nécessitent une inversion dans \mathbb{F}_p , opération extrêmement coûteuse en pratique. Pour éviter cette inversion, on utilise généralement une forme projective de la courbe E .

On définit une relation d'équivalence \sim dans l'ensemble $\mathbb{F}_p^3 / \{(0, 0, 0)\}$ telle que

$$(X, Y, Z) \sim (\lambda^c X, \lambda^d Y, \lambda Z)$$

avec $\lambda \in \mathbb{F}_p^*$. Un point projectif est une classe d'équivalence notée $(X : Y : Z)$ telle que

$$(X : Y : Z) = \{(\lambda^c X, \lambda^d Y, \lambda Z) : \lambda \in \mathbb{F}_p^*\}.$$

Un point $(X, Y, Z) \in (X : Y : Z)$ est appelé un représentant de la classe. Enfin, on définit une relation injective entre l'ensemble des points projectifs $\mathbb{P}(\mathbb{F}_p)$ et l'ensemble des points affines $\mathbb{A}(\mathbb{F}_p)$:

$$\begin{aligned} \mathbb{A}(\mathbb{F}_p) &\hookrightarrow \mathbb{P}(\mathbb{F}_p)^* = \{(X : Y : Z) : X, Y, Z \in \mathbb{F}_p \text{ avec } Z \neq 0\} \\ (x, y) &\mapsto (X/Z^c : Y/Z^d : 1). \end{aligned}$$

En coordonnées projectives, l'ensemble $\mathbb{P}(\mathbb{F}_p)^0 = \{(X : Y : Z), X, Y, Z \in \mathbb{F}_p \text{ avec } Z = 0\}$ est appelé ligne à l'infini. On définit différents systèmes de coordonnées projectives en modifiant les paramètres c et d .

Nous utiliserons dans ce manuscrit les coordonnées projectives Jacobiennes définies par $c = 2$ et $d = 3$. Dans ce système, le point projectif $(X : Y : Z)$ avec $Z \neq 0$ correspond au point affine $(X/Z^2, Y/Z^3)$. L'équation projective de la courbe est $Y^2 = X^3 + aXZ^4 + bZ^6$ et le point à l'infini correspond à $(1 : 1 : 0)$.

L'opposé du point $(X : Y : Z)$ est le point $(X : -Y : Z)$. Le doublement d'un point $P = (X_P : Y_P : Z_P)$ est calculé par :

$$\begin{aligned} X_{2P} &= (3X_P^2 + aZ_P^4)^2 - 8X_P Y_P^2 \\ Y_{2P} &= (3X_P^2 + aZ_P^4)(4X_P Y_P^2 - X_{2P}) - 8Y_P^4 \\ Z_{2P} &= 2Y_P Z_P. \end{aligned} \tag{2.6}$$

Enfin, l'addition $P + Q$ avec $P = (X_P : Y_P : Z_P)$ et $Q = (X_Q : Y_Q : Z_Q)$ est calculée par :

$$\begin{aligned} X_{P+Q} &= (Y_Q Z_P^3 - Y_P Z_Q^3)^2 - (X_P Z_Q^2 + X_Q Z_P^2)(X_Q Z_P^2 - X_P Z_Q^2)^2 \\ Y_{P+Q} &= (Y_Q Z_P^3 - Y_P Z_Q^3)(X_P Z_Q^2(X_Q Z_P^2 - X_P Z_Q^2)^2 - X_{P+Q}) - Y_P Z_Q^3(X_Q Z_P^2 - X_P Z_Q^2)^3 \\ Z_{P+Q} &= Z_P Z_Q(X_Q Z_P^2 - X_P Z_Q^2). \end{aligned} \tag{2.7}$$

Grâce à ces opérations de groupe, nous pouvons maintenant définir la multiplication scalaire.

2.4.4 La multiplication scalaire

Considérons un point $P \neq \infty$ d'une courbe elliptique et un scalaire $k \in \mathbb{N}^*$. La multiplication scalaire, notée $[k]P$, est l'opération qui consiste à calculer :

$$Q = [k]P = \underbrace{P + P + \dots + P}_k. \tag{2.8}$$

Les algorithmes que nous présentons sont les versions additives des algorithmes d'exponentiation rapide. Pour des raisons évidentes, il est impensable de calculer $[k]P$ en effectuant $k - 1$ additions successives, qui aurait une complexité en $\mathcal{O}(k)$ opérations. Dans [42], Gordon présente une compilation des principales techniques d'exponentiation dans un groupe multiplicatif. La version additive de ces algorithmes est présentée en particulier dans [21].

L'algorithme classique de multiplication scalaire est la version classique de la méthode *Square-and-multiply*. Il opère bit-à-bit sur la représentation binaire de k des poids forts vers les poids faibles et les relations de récurrences suivantes :

$$[k]P = \begin{cases} P + [2] \left(\left[\frac{k-1}{2} \right] P \right) & \text{si } k \text{ est impair} \\ [2] \left(\left[\frac{k}{2} \right] P \right) & \text{si } k \text{ est pair} \end{cases} \tag{2.9}$$

Exprimé plus simplement, pour chaque bit de k l'algorithme effectue un doublement, suivi d'une addition si le bit est à 1. Cela revient à écrire le calcul de $[k]P$ sous forme de Horner, comme décrit dans l'exemple.

Exemple 5. Prenons $k = 367 = (101101111)_2$. Le calcul de $[367]P$ peut se faire en calculant :

$$Q = ([2]([2]([2]([2]([2]([2]([2]([2]P) + P) + P)) + P) + P) + P) + P)$$

Le coût de cet algorithme est en moyenne de :

$$\frac{t}{2}A + (t - 1)D, \tag{2.10}$$

où t est la taille de k en bits, A est le coût d'une addition sur E et D le coût du doublement sur E .

Remarque 10. Il existe une version qui opère bit-à-bit des poids faibles vers les poids forts.

Il existe de très nombreux algorithmes de multiplication scalaire. Une bonne partie d'entre eux peuvent se résumer par l'algorithme 8. Les différences apparaissent sur la représentation du scalaire k qui influe sur les traitements à réaliser pour chaque chiffre k_i de la représentation.

Algorithme 7: Doublement et addition

Données : $P \in E(\mathbb{F}_p)$, $k = (k_{n-1}, \dots, k_0)_2 \in \mathbb{N}$
Résultat : $Q = [k]P \in E(\mathbb{F}_p)$
 $Q \leftarrow \infty$
 $i \leftarrow n - 1$
tant que $i \geq 0$ **faire**
 $Q \leftarrow 2Q$
 si $k_i = 1$ **alors** $Q \leftarrow P + Q$
 $i \leftarrow i - 1$
fin
retourner Q

Algorithme 8: Multiplication scalaire Poids-forts Poids-faibles généralisée

Données : $P \in E(\mathbb{F}_p)$, $Rep(k) = (k_{l-1}, \dots, k_0)$ une représentation de k
Résultat : $Q = [k]P \in E(\mathbb{F}_p)$
1 $Q \leftarrow \infty$
2 **pour** i de $l - 1$ à 0 **faire**
3 $Q \leftarrow [prev]Q$
4 $Q \leftarrow Q + [med]P$
5 $Q \leftarrow [post]Q$
6 **fin**
7 **retourner** Q

Remarque 11. L'algorithme 7 est obtenu à partir de l'algorithme 8 en posant $Rep(k)$ comme la représentation binaire classique de k , $prev = 2$, $med = k_i$ et $post = 1$.

Définition 4 (Chaînes d'addition [17]). Une chaîne d'additions est un ensemble de s entiers u_i , calculant un entier k , tels que $u_0 = 1 < u_1 < u_2 < \dots < u_{s-1} = k$, et tels que tout u_i s'écrit comme la somme de deux termes précédemment calculés. Trouver la chaîne d'additions la plus courte calculant un entier k permet d'accélérer la multiplication scalaire mais est un problème difficile. Une méthode pour trouver de bonnes chaînes d'additions est donnée dans [64].

2.4.4.1 Algorithmes à fenêtre : représentation en base 2^w

L'algorithme 7 opère bit à bit, c'est-à-dire en base 2. Suivant le même principe, les algorithmes à fenêtre opèrent w -bits à w -bits, c'est-à-dire en base 2^w soit $Rep(k) = \sum_0^{N/w-1} k_i 2^{iw}$. Pour calculer la multiplication scalaire, on pourrait donc poser simplement $prev = 2^w$, $med = k_i$ et $post = 1$. Cela nécessiterait de pré-calculer $[k_i]P$ pour $k_i \in [2, 2^w - 1]$.

En pratique, on pré-calculer $[k_i]P$ pour k_i impair. À chaque étape de l'algorithme, on va calculer deux entiers s et u tels que $k_i = 2^s u$ et u impair. En posant $Rep(k) = k_{2^w}$, $prev = 2^{w-s}$, $med = u$ et $post = 2^s$, l'algorithme 8 calcule la multiplication scalaire avec un temps moyen égal à :

$$(N - 1)D + ((N/w) - 1) \left(\frac{2^w - 1}{2^w} \right) A.$$

Exemple 6. Soit $k = 4225528843 = (11111011110111000111010000001011)_2$. On choisit $w = 4$. Le tableau suivant donne l'écriture de k en base 2^4 , les valeurs u et s et la valeur de Q après chaque ligne de l'algorithme 8.

k=(1111	1011	1101	1100	0111	0100	0000	1011) ₂
k=(15	11	13	12	7	4	0	11) _{2⁴}
	15	11	13	2 ² × 3	7	2 ² × 1	0	11	
1.3	1	[240]	[4016]	[16116]	[1031616]	[4126492]	[264095552]	[4225528832]	
1.4	[15]	[251]	[4029]	[16119]	[1031623]	[4126493]	[264095552]	[4225528843]	
1.5	[15]	[251]	[4029]	[64476]	[1031623]	[16505972]	[264095552]		

Remarque 12. Un algorithme appelé multiplication scalaire à fenêtre glissante est dérivé de cette méthode : on fait glisser la fenêtre de w bits en passant les plages de bits à 0 (doublements successifs).

2.4.4.2 Non-adjacent form

L'inversion dans le groupe $E(\mathbb{F}_p)$ est une opération très simple. On peut donc utiliser une représentation signée pour l'entier k :

$$k = \sum_{i=0}^l k_i 2^i \text{ avec } k_i \in \{-1, 0, 1\}.$$

Cette représentation signée n'est pas unique. Or, les algorithmes précédents opèrent un doublement pour chaque chiffre k_i de la représentation, plus une addition lorsque $k_i \neq 0$. Pour minimiser le nombre d'opérations, on va donc choisir une représentation signée de taille $l \approx N$ et telle que le nombre de $k_i \neq 0$ soit minimal.

On appelle *Non-Adjacent Form (NAF)* la représentation de k telle que $k = \sum_{i=0}^l 2^i k_i$ avec $k_i \in \{-1, 0, 1\}$ et $k_i k_{i+1} = 0$, soit au moins un chiffre nul entre deux chiffres non-nuls. Cette représentation a plusieurs propriétés :

- est unique ;
- sa longueur l est au maximum égale à $N + 1$;
- la densité des chiffres non nuls dans $NAF(k)$ est environ $1/3$.

En utilisant cette représentation, on diminue donc le poids de Hamming de k et ainsi le nombre d'additions nécessaires. En posant $Rep(k) = NAF(k)$, $prev = 2$, $med = k_i$ et $post = 1$, l'algorithme 8 calcule la multiplication scalaire avec en moyenne l doublements et $l/3$ additions.

Selon le même principe, on peut utiliser une représentation NAF à fenêtre w . On a toujours $k = \sum_{i=0}^{l-1} k_i 2^i$ mais $k_i \in [-2^{w-1}, 2^{w-1} - 1]$ et il y a au moins $w - 1$ chiffres nuls entre deux chiffres non-nuls. La densité des ces derniers dans la représentation est alors égale à $1/(w + 1)$. L'algorithme 9 calcule $NAF_w(k)$ par une succession de divisions de k par 2. Les chiffres de la représentation sont alors les restes signés de k par 2^w (opération $k \bmod 2^w$) soit les entiers u tels que $u \equiv k \pmod{2^w}$ et $-2^{w-1} \leq u < 2^{w-1}$.

Algorithme 9: Calcul de $NAF_W(k)$

Données : Taille de la fenêtre w , $k \in \mathbb{N}^*$

Résultat : $NAF_W(k)$

```

1   $i \leftarrow 0$ 
2  tant que  $k \geq 1$  faire
3      si  $k$  est impair alors
4           $k_i \leftarrow k \bmod 2^w$ 
5           $k \leftarrow k - k_i$ 
6      sinon
7           $k_i \leftarrow 0$ 
8      fin
9       $k \leftarrow k/2$ 
10      $i \leftarrow i + 1$ 
11 fin
12 retourner  $(k_i - 1, k_i - 2, \dots, k_1, k_0)$ 

```

La longueur de la représentation NAF_w est au plus égale à $l + 1$. L'exemple suivant présente les trois premières représentations NAF_w d'un entier.

Exemple 7. Soit $k = 4225528843 = (11111011110111000111010000001011)_2$

NAF	100000 $\bar{1}$ 0000 $\bar{1}$ 00 $\bar{1}$ 00100 $\bar{1}$ 010000010 $\bar{1}$ 0 $\bar{1}$
NAF ₃	100000 $\bar{1}$ 0000 $\bar{1}$ 00 $\bar{1}$ 0010000 $\bar{3}$ 000000100 $\bar{3}$
NAF ₄	100000 $\bar{1}$ 0000 $\bar{1}$ 00000 $\bar{7}$ 0000 $\bar{3}$ 000001000 $\bar{5}$

Avec $Rep(k) = NAF_w(k)$, $prev = 2$, $med = k_i$ et $post = 1$ et le pré-calcul des k_i impairs avec $k_i \in [-2^{w-1}, 2^{w-1} - 1]$, l'algorithme 8 calcule la multiplication scalaire avec un temps moyen égal à :

$$lD + \frac{l}{w+1}A.$$

Remarque 13. La représentation NAF_w présentée ici est donnée en base 2. On également écrire la représentation NAF_w en base β (si une multiplication $[\beta]P$ efficace est connue). Cette méthode est décrite dans [97].

2.4.4.3 Double-Base number system

Le système de représentation double base [26] consiste à écrire un nombre comme la somme de produits de puissances de deux nombres u et v premiers entre eux, soit :

$$k = \sum_{i=0}^{l-1} k_i u^{a_i} v^{b_i} \text{ avec } k_i \in \{-1, 1\}.$$

L'entier est alors représenté par les triplets (k_i, a_i, b_i) qui le définissent. Pour un entier k , la représentation double-base n'est pas unique, comme le montre l'exemple suivant. Des algorithmes pour calculer ces représentations sont donnés dans [24].

Exemple 8. Soit $k = 4225528843$, $u = 2$, $v = 3$.

Chaîne 1	Chaîne 2	Chaîne 3
(1, 5, 17)	(1, 32, 0)	(1, 24, 5)
(1, 1, 16)	(-1, 26, 0)	(1, 19, 5)
(1, 0, 14)	(-1, 21, 0)	(1, 18, 4)
(1, 0, 13)	(-1, 18, 0)	(1, 10, 3)
(1, 0, 12)	(1, 15, 0)	(1, 9, 1)
(1, 1, 9)	(-1, 11, 0)	(1, 9, 0)
(1, 1, 6)	(-1, 10, 0)	(1, 3, 0)
(1, 1, 5)	(1, 3, 0)	(1, 1, 0)
(1, 0, 4)	(1, 1, 0)	(1, 0, 0)
(1, 1, 3)	(1, 0, 0)	
(1, 1, 1)		
(1, 0, 0)		

Pour être efficaces, les algorithmes de multiplication scalaire qui utilisent cette représentation doivent utiliser comme base deux entiers u et v pour lesquels les opérations $[u]P$ et $[v]P$ sont rapides. En pratique, on utilise une base $(2, 3)$ car des formules de doublement et de triplement efficaces sont connues pour un grand nombre de systèmes de coordonnées. En posant $Rep(k)$ une chaîne double-base $(2,3)$ de longueur l , $prev = 2^{a_i} 3^{b_i}$, $med = k_i$ et $post = 1$, l'algorithme 8 calcule la multiplication scalaire en l additions, $\max a_i$ doublements et $\max b_i$ triplements.

CHAPITRE 3

ARITHMÉTIQUES PARALLÈLES

Sommaire

3.1	Parallélisme sur architecture SMP à mémoire partagée	44
3.1.1	Arithmétique entière	44
3.1.2	Arithmétique modulaire	46
3.1.3	Courbes elliptiques	47
3.2	Parallélisme sur les processeurs graphiques	53
3.2.1	Représentation des entiers multiprécision	53
3.2.2	Gestion de la mémoire	54
3.2.3	Arithmétique modulaire	56
3.2.4	Arithmétique des courbes elliptiques	57
3.3	Conclusion et perspectives	61
3.3.1	Architecture SMP	61
3.3.2	Processeurs graphiques	62

Dans ce chapitre nous présentons nos implémentations d'opérateurs arithmétiques parallèles sur architecture SMP à mémoire partagée, puis la parallélisation de calculs d'opérations arithmétiques sur processeurs graphiques.

Sur architecture SMP, nous nous intéressons tout d'abord à la parallélisation de l'addition et de la multiplication entières. Nous avons également travaillé à la parallélisation du calcul d'ensembles d'additions. Nous verrons que si la multiplication tire parti du parallélisme, il n'en va pas de même pour l'addition : le surcôté lié au parallélisme ne peut pas être amorti pour les tailles d'entiers visées (plusieurs centaines ou milliers de bits).

Nous utilisons ensuite ces algorithmes pour les calculs modulaires, en particulier pour la multiplication. Si les algorithmes de Barrett et de Montgomery se prêtent mal au parallélisme logiciel du fait de la séquentialité de leurs calculs, un parallélisme au niveau des opérations entières permet des gains de performances en termes de temps de calcul.

Enfin, nous donnons nos premiers résultats pour la parallélisation de la multiplication scalaire sur les courbes elliptiques. La séquentialité des algorithmes fait que cette opération se prête mal au parallélisme. Cependant, en découpant le scalaire de manière efficace, nous pouvons obtenir des gains de performances intéressants.

Ces opérateurs ont été ajoutés à notre bibliothèque PACE (Prototyping Arithmetic for Cryptography Easily) que nous présentons dans l'annexe A. Cette bibliothèque définit des types particuliers pour chacune des arithmétiques qui nous intéressent, ainsi que les opérations arithmétiques correspondantes. Grâce à la généricité et au polymorphisme du langage C++, nous pouvons facilement utiliser différentes implémentations pour une même opération et comparer leurs performances.

Sur processeur graphique, nous proposons une implémentation pour l'arithmétique modulaire multiprécision ainsi que pour l'arithmétique des courbes elliptiques définies sur un corps fini premier en système de coordonnées jacobiniennes. Nous décrivons comment représenter les entiers multiprécision sur architecture Tesla ainsi que nos implémentations de l'addition, la soustraction et la multiplication de Montgomery. Grâce à ces opérateurs et en définissant l'addition et le doublement dans $E(\mathbb{F}_p)$, nous pouvons paralléliser des milliers d'instances de multiplication scalaire $[k]P$.

3.1 Parallélisme sur architecture SMP à mémoire partagée

La première règle que nous fixons pour réduire le surcoût lié au parallélisme est de ne pas lancer plus de tâches que d'unités de calcul présentes sur l'architecture considérée. Dans le cas contraire, les latences augmentent proportionnellement au nombre de tâches parallèles, alors que le calcul n'en tire pas parti : les tâches sont réparties sur l'ensemble des processeurs mais séquentialisées (quand une tâche finit, une autre commence).

Remarque 14. Lancer plus de tâches que de processeurs peut cependant être avantageux pour des architectures utilisant l'*hyperthreading*, c'est-à-dire un parallélisme au niveau de l'unité de calcul.

Dans nos applications, le coût lié au lancement et à l'arrêt des tâches parallèles est amorti en réalisant une seule fois ces opérations. Les calculs sont alors totalement parallèles : les opérateurs arithmétiques sont greffés sur des tâches déjà existantes sur les processeurs.

L'ensemble des opérateurs décrits ici sont génériques en termes de taille des opérandes. Nous avons défini des opérateurs génériques en nombre d'unités de calcul, mais les meilleures performances sont obtenues avec des opérateurs implémentés spécifiquement pour un nombre de tâches fixé. Cela signifie une implémentation explicite des opérations réalisées par chacune des T tâches.

3.1.1 Arithmétique entière

L'addition et la multiplication entières multiprécision peuvent être considérées comme deux opérations atomiques. Elles servent dans tous les calculs de plus haut niveau. Dès lors, paralléliser ces opérations permet de bénéficier du parallélisme aux niveaux arithmétiques supérieurs. Nous présentons les méthodes utilisées pour paralléliser ces deux opérations ainsi que les performances obtenues par la multiplication parallèle.

3.1.1.1 Addition et soustraction entières

Les algorithmes d'addition et de soustraction entières présentés dans le premier chapitre sont des algorithmes à propagation de retenues. Il existe d'autres méthodes pour réaliser ces opérations et notamment des méthodes parallèles dont des versions en précision fixe sont décrites dans [69]. Cependant, ces opérations tirent parti d'un parallélisme au niveau matériel.

Nous avons implémenté l'algorithme d'addition classique entre entiers de taille n . Chacune des T tâches réalise une addition de n/T mots. Après une barrière de synchronisation, chaque tâche réalise la propagation de retenue sur l'ensemble des mots restants. Nous avons remarqué qu'au niveau logiciel, la synchronisation prend plus de temps que l'addition pour des tailles d'opérandes inférieures à 200 000 bits. En pratique, l'addition est donc réalisée par une seule tâche.

3.1.1.2 Ensemble d'additions

Un cas important à considérer est le calcul d'un ensemble d'additions. Nous le rencontrons en particulier pour la multiplication entière et la méthode de multiplication modulaire que nous présentons au chapitre 4. On pose le problème ainsi : comment additionner un ensemble de m entiers de type $c_i \beta^{d_i}$.

Le cas le plus simple est défini par $d_i = 0$ pour $i \in [0, m - 1]$ et des entiers taille n . Séquentiellement, additionner l'ensemble de ces opérations a une complexité égale à $(m - 1)A(n)$. Une méthode classique pour paralléliser ces opérations consiste à calculer un arbre de réduction où à l'étape 0 on

effectue $a_0 = m/2$ additions et à l'étape i , $a_i = a_{i-1}/2$ additions. La complexité parallèle de ce type d'algorithme est donc égale la profondeur de l'arbre fois la complexité de l'addition entre entiers de taille n , soit $\lceil \log_2 m \rceil A(n)$. Nous avons implémenté cet algorithme, en théorie efficace. En pratique, on doit réaliser $\lceil \log_2 m \rceil$ synchronisations entre les différentes tâches : une après chaque groupe a_i . Or la synchronisation est plus coûteuse que l'addition pour des tailles d'entiers cryptographique. Cette méthode est donc efficace lorsque le ratio entre le temps d'exécution de $\lceil \log_2 m \rceil - 1$ synchronisations et le temps de calcul $m - 1$ additions est supérieur à 1.

Une méthode plus efficace consiste à paralléliser les $m - 1$ additions en confiant à chaque tâche $m - 1$ additions partielles entre opérandes de taille n/T . La tâche accumule les retenues et une seule propagation est alors nécessaire, soit une barrière de synchronisation.

Nous le verrons par la suite, la parallélisation de la multiplication entière nécessite l'addition d'un certain nombre d'entiers de type $c_i \beta^{d_i}$. L'arbre d'additions est d'autant moins efficace qu'il s'agit ici d'additionner des entiers de petites tailles. Si nous avons implémenté cette méthode en arrangeant l'arbre pour calculer des additions partielles de manière efficace, en pratique la méthode qui donne les meilleurs temps de calcul pour les tailles considérées est celle qui consiste à calculer toutes les additions sur un même processeur.

3.1.1.3 Multiplication entière

Notre première implémentation de la multiplication entière utilise le schéma de multiplication quadratique : si $k_1 \times k_2$ processeurs sont disponibles sur l'architecture, les opérandes sont divisés en k_1 et k_2 morceaux. Chaque tâche réalisera une multiplication séquentielle de complexité égale à $M(n/k_1, n/k_2)$.

On peut réduire ce nombre de processeurs en utilisant les schémas de multiplication sous-quadratique de Karatsuba et de Toom-Cook. Si la taille des opérandes est assez importante, on pourra par exemple utiliser une parallélisation de type Toom-3 si le nombre de processeurs est un multiple de 5, Karatsuba si le nombre de processeurs est un multiple de 3, etc. Cependant, l'utilisation d'algorithmes sous-quadratique induit des tâches hétérogènes et une séquentialité dans la suite des traitements. Dans ces travaux, nous utilisons pour le moment uniquement la version quadratique de la multiplication parallèle.

Nous avons implémenté deux types d'algorithmes : des algorithmes génériques qui peuvent être lancés sur un nombre arbitraire d'unités de calcul et des algorithmes spécifiques à un nombre de tâches fixé. Dans ce dernier cas, nous définissons explicitement chacune des tâches. Cette méthode permet de minimiser le surcoût lié au parallélisme et aux échanges de données. Les multiplications $M(n/k_1, n/k_2)$ réalisées par les tâches sont séquentielles et utilisent les algorithmes optimisés pour les tailles considérées.

Les zones mémoires nécessaires pour stocker les variables temporaires sont allouées une seule fois. Nous maintenons ainsi un ensemble de tâches prêtes à l'emploi qui sont lancées par une unique instruction sur l'ensemble de tâches. Dans nos algorithmes génériques, cela nous permet de pré-calculer certaines constantes annexes à la multiplication (taille et position des morceaux que la tâche aura à multiplier par exemple).

3.1.1.4 Résultats expérimentaux

La figure 3.1 présente l'accélération de nos implémentations parallèles par rapport à la multiplication séquentielle en forçant l'algorithme de multiplication quadratique (`mpn_mul_basecase`) dans les deux cas. Nous vérifions ainsi que les accélérations obtenues correspondent à la complexité théorique attendue, c'est-à-dire qu'utiliser T processeurs permet une accélération de T par rapport à la version séquentielle de l'algorithme. La figure montre clairement qu'une fois le surcoût introduit par le parallélisme amorti par l'intensité arithmétique des opérations, nous obtenons bien les résultats attendus.

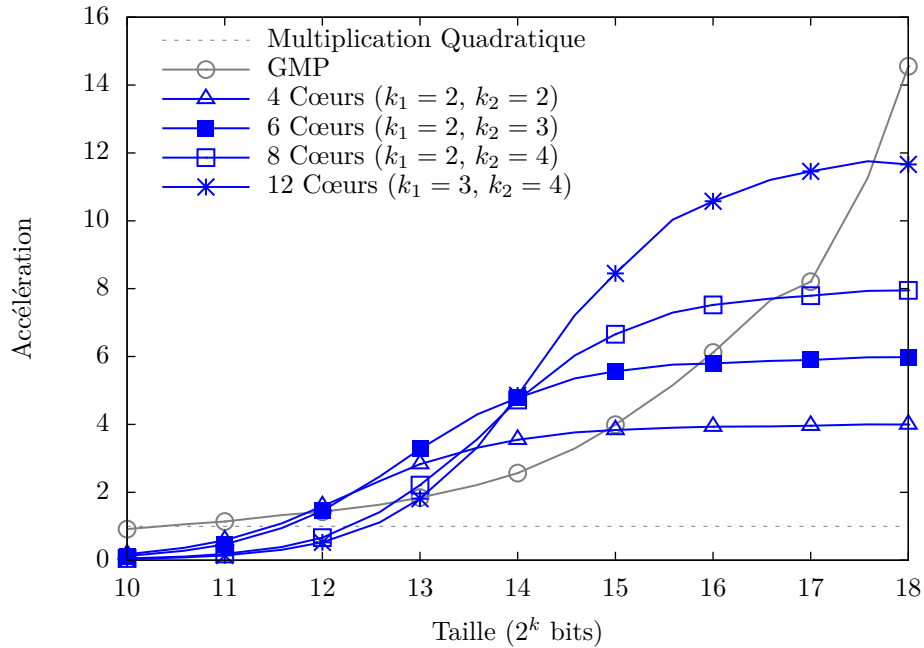


FIGURE 3.1 – Comparaison des performances entre différentes implémentations de la multiplication entière utilisant un schéma de parallélisation quadratique et la multiplication séquentielle quadratique

Nous avons ajouté à la figure la courbe des performances de la multiplication classique de GMP `mpn_mul` qui utilise les algorithmes optimisés en fonction de la taille des opérandes (voir §2.2.6). Si nos implémentations parallèles sont plus rapides pour des entiers de tailles comprises entre 2^{12} et 2^{17} bits, il est clair que les versions parallèles utilisant une multiplication quadratique ne donnent pas les meilleures performances.

Les multiplications séquentielles de complexité $M(n/k_1, n/k_2)$ réalisées par nos tâches peuvent utiliser les algorithmes à complexités sous-quadratiques en utilisant la fonction `mpn_mul` de GMP. La figure 3.2 présente l'accélération de nos opérateurs utilisant ces algorithmes par rapport à la multiplication classique de GMP.

Pour des entiers de petites tailles, le surcoût lié au parallélisme est toujours présent. Utiliser les algorithmes sous-quadratiques réduit les gains observés précédemment : en utilisant T tâches, on obtient une accélération d'environ $T/2$ au lieu de T . Le parallélisme est cependant intéressant pour une large plage d'entiers. On note une décroissance des performances pour des entiers de taille supérieure à 2^{17} bits, ce qui correspond à l'utilisation de la multiplication FFT sur notre architecture.

3.1.2 Arithmétique modulaire

Les opérations d'arithmétique modulaire sont composées d'un certain nombre d'opérations entières. Pour qu'ils bénéficient des avantages des architectures parallèles, on peut donc utiliser les opérateurs définis précédemment.

Les algorithmes de Barrett et de Montgomery sont composés de 3 multiplications $M(n)$ et d'un ensemble d'opérations simples (addition, calcul du quotient ou du reste d'un nombre par une puissance de la base). Nos opérateurs parallèles lancent le nombre de tâches choisi en amont du calcul et les terminent à la fin du calcul de la multiplication modulaire. Sur ces tâches actives, on lance nos opérateurs parallèles de multiplication. Les opérations simples sont elles réalisées par une seule tâche. La complexité parallèle de ces algorithmes en utilisant $k_1 \times k_2$ processeurs est alors égale à

$$3M(n/k_1, n/k_2). \quad (3.1)$$

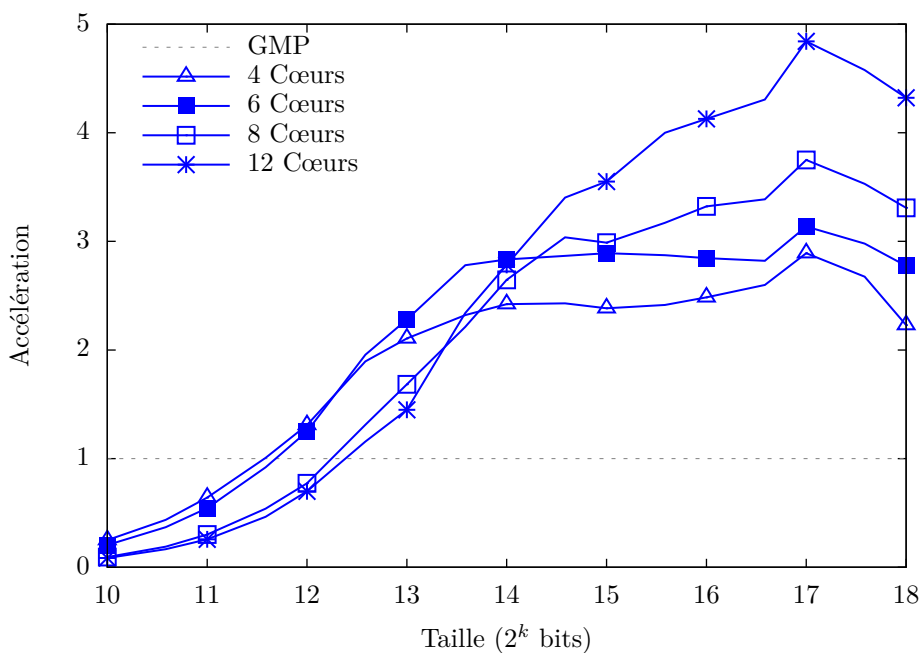


FIGURE 3.2 – Comparaison des performances entre les opérateurs de multiplications utilisant un schéma de parallélisation quadratique basés sur les algorithmes de multiplication optimisés de GMP

La multiplication bipartite utilise un parallélisme au niveau de l'arithmétique modulaire et calcule deux produits $ab_1 \bmod p$ et $ab_0\beta^{-n/2} \bmod p$ indépendants. Le premier produit utilise la multiplication de Barrett, le second celle de Montgomery. Chacun d'eux a une complexité égale à $2M(n, n/2) + M(n/2)$. En utilisant les algorithmes de Barrett et de Montgomery décrits précédemment, si $k_1 \times k_2$ processeurs sont disponibles et avec k_2 pair, en parallélisant chaque produit modulaire sur la moitié des processeurs, on réalise une multiplication bipartite avec une complexité égale à

$$2M(n/k_1, n/k_2) + M(n/2k_1, n/k_2). \quad (3.2)$$

On peut également séquentialiser les deux produits modulaires. Ainsi, chacun bénéficiera de la totalité des processeurs : si l'on dispose de $k_1 \times k_2$ processeurs, la multiplication modulaire bipartite est réalisée avec une complexité égale à

$$4M(n/k_1, n/2k_2) + 2M(n/2k_1, n/2k_2).$$

Dans le chapitre 4, nous présentons une parallélisation de la multiplication modulaire au niveau de l'arithmétique modulaire. Nous comparons alors les performances obtenues par chacune des implémentations.

3.1.3 Courbes elliptiques

Nous nous intéressons ici à la parallélisation de la multiplication scalaire sur les courbes elliptiques, soit au calcul $[k]P$ avec $k \in \mathbb{N}^*$ et $P \in E(\mathbb{F}_p)$. Paralléliser cette opération est un problème difficile du fait de la séquentialité des algorithmes qui effectuent leurs calculs en utilisant les chaînes d'additions ou les formes de Horner, ce qui entraîne une dépendance forte entre les opérations (voir §2.4.4).

Remarque 15. On peut paralléliser l'arithmétique à un autre niveau : par exemple paralléliser les opérations du groupe $E(\mathbb{F}_p)$, ou encore paralléliser les opérations entières ou les opérations modulaires. Cependant, comme nous l'avons vu précédemment, il faut des entiers de tailles supérieures à 2000 bits pour avoir des opérations parallèles efficaces. Or un des intérêts de la cryptographie basée sur les courbes elliptiques est ses tailles de clés réduites.

Une parallélisation simple de la multiplication scalaire revient à trouver un ensemble de n entiers K_i tels que :

$$k = K_0 + K_1 + \dots + K_{n-1}.$$

La multiplication $[k]P$ s'écrit alors

$$[k]P = [K_0]P + [K_1]P + \dots + [K_{n-1}]P.$$

Chaque opération $[K_i]P$ est indépendante. L'algorithme 10 donne le schéma d'un calcul parallèle : chaque tâche calcule $Q_i = [k_i]P$. Une fois ces calculs réalisés, les points Q_i sont additionnés par un arbre d'addition classique suivant l'algorithme 11. Pour chaque pas de l'arbre entre 1 et $\log_2 n$, les tâches dont les identifiants i sont multiples du pas additionnent le point Q_i et le point Q_{i+pas} .

Algorithme 10: Multiplication scalaire parallèle

Données : (K_{n-1}, \dots, K_0) tels que $k = \sum_{i=0}^{n-1} K_i$, $P \in E(\mathbb{F}_p)$
Résultat : $[k]P$
1 pour i de 0 à $n-1$ calculer en parallèle
2 | $Q_i \leftarrow [K_i]P$
3 fin
4 $Q \leftarrow \text{Addition}(Q_0, \dots, Q_{n-1})$
5 retourner Q

Algorithme 11: Addition

Données : $Q_0, \dots, Q_{n-1} \in E(\mathbb{F}_p)$
Résultat : $Q = \sum_{i=0}^{n-1} Q_i$
1 pour i de 0 à $n-1$ calculer en parallèle
2 | $s \leftarrow 0$
3 | tant que $s < n$ faire
4 | | si $i \equiv 0 \pmod{s}$ et $i + s < n$ alors
5 | | | $Q_i \leftarrow Q_i + Q_{i+s}$
6 | | fin
7 | | $s \leftarrow 2s$
8 | | synchroniser
9 | fin
10 fin
11 retourner Q_0

L'algorithme 11 est exécuté en

$$\lceil \log_2(n) \rceil A.$$

La difficulté ici est de trouver le bon ensemble de valeurs (K_0, \dots, K_i) permettant de minimiser le temps de calcul de la multiplication scalaire.

Dans la suite, nous considérons une représentation de k de taille l où la proportion de chiffre k_i non-nuls est égale à $1/(w+1)$, comme les représentations NAF_w . Dans la représentation binaire classique, on a $w = 1$.

3.1.3.1 Approche naïve

La méthode la plus simple est de diviser l'entier k en n morceaux de tailles égales. On pose $k = \sum_{i=0}^{n-1} k_i \beta^i$ avec $\beta = 2^{l/n}$ et $K_0 = k_0$, $K_1 = k_1 \beta$, \dots , $K_{n-1} = k_{n-1} \beta^{n-1}$. Le calcul de $[k]P$ s'écrit alors :

$$[k]P = \left[\sum_{i=0}^{n-1} k_i \beta^i \right] P = [k_{n-1} \beta^{n-1}]P + \dots + [k_1 \beta]P + [k_0]P \quad (3.3)$$

$$= [k_{n-1}]([\beta^{n-1}]P) + \dots + [k_1](\beta P) + [k_0]P. \quad (3.4)$$

Si P est connu, le pré-calcul des points $P_i = [\beta^i]P$ peut se faire en amont du calcul. Dans ce cas, le temps de calcul parallèle de la multiplication scalaire est égal à :

$$\frac{l}{n}D + \left(\frac{l}{n(w+1)} + \log_2(n) \right) A.$$

En revanche, si P n'est pas connu *a priori*, le temps de calcul parallèle est égal à :

$$lD + \left(\frac{l}{n(w+1)} + \log_2(n) \right) A.$$

Par rapport aux algorithmes séquentiels utilisant la représentation binaire classique ou les représentations NAF_w , le seul gain en termes de temps de calcul est sur le nombre d'additions à réaliser, qui passe de $l/(w+1)$ à $l/[n(w+1)] + \log_2(n)$. De plus, les temps de calcul de chacune des tâches sont totalement différents.

Nous présentons une méthode permettant d'homogénéiser les temps de calcul sur chacune des tâche et de réduire le temps parallèle théorique nécessaire au calcul de $[k]P$.

3.1.3.2 Répartition homogène des calculs

Si les pré-calculs $[\beta^i]P$ ne sont pas possibles, nous répartissons les calculs de façon homogène en découpant k en n morceaux de tailles l_0, \dots, l_{n-1} respectivement tel que

$$k = k_0 + k_1\beta^{l_0} + k_2\beta^{l_0+l_1} + \dots + k_{n-1}\beta^{\sum_{i=0}^{n-2} l_i}$$

et

$$\sum_{i=0}^{n-1} l_i = N$$

comme illustré à la figure 3.3.

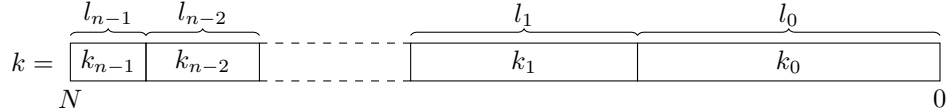


FIGURE 3.3 – Découpage non homogène de k

On pose $D = A$. Le calcul de $[k_0]P$ se fait en

$$\left[l_0 + \frac{l_0}{w+1} \right] D,$$

le calcul de $[k_1\beta^{l_0}]P$ en

$$\left[l_0 + l_1 + \frac{l_0}{w+1} \right] D,$$

le calcul $[k_i\beta^{i-1}]P$ en

$$\left[\sum_{j=0}^i l_j + \frac{l_i}{w+1} \right] D.$$

Pour avoir des temps de calcul homogènes sur chaque tâche, il faut donc :

$$\frac{(w+1)+1}{(w+1)}l_0 = l_0 + \frac{(w+1)+1}{(w+1)}l_1 = \dots = l_0 + l_1 + \dots + l_{n-2} + \frac{(w+1)+1}{(w+1)}l_{n-1}$$

soit

$$\begin{aligned} l_1 &= \frac{1}{w+2} l_0 \\ l_2 &= \frac{1}{(w+2)^2} l_0 \\ \dots & \\ l_i &= \frac{1}{(w+2)^i} l_0. \end{aligned} \tag{3.5}$$

De plus, comme $\sum_{i=0}^{n-1} l_i = N$, on a :

$$\sum_{i=0}^{n-1} \frac{1}{(w+2)^i} l_0 = \frac{\frac{1}{(w+2)^n} - 1}{\frac{1}{w+2} - 1} l_0 = N \tag{3.6}$$

d'où

$$l_0 = \frac{(w+1)(w+2)^{n-1}}{(w+2)^n - 1} N. \tag{3.7}$$

Grâce à cette relation, on peut donc fixer les longueurs l_i du découpage à partir de la taille N du scalaire k , d'un nombre de processeurs n , et des relations de récurrences 3.5.

3.1.3.3 Résultats expérimentaux

Nous avons testé sur la plate-forme HPC@LR les implémentations des deux approches.

La figure 3.4 présente les résultats obtenus par l'approche avec pré-calcul. Chaque courbe représente l'accélération d'une version parallèle d'un algorithme utilisant la représentation NAF_w par rapport à la version séquentielle du même algorithme.

Sur 2, 4 et 6 cœurs, notre implémentation atteint quasiment les résultats attendus, soit une accélération de 2, 4 ou 6 par rapport à la version séquentielle. L'influence de l'arbre d'additions est cependant visible pour 4 et 6 cœurs avec des gains variant de 3,7 à 3,9 et de 5,2 à 5,9 respectivement.

En revanche, les résultats expérimentaux sur 8, 10 et 12, les implémentations pâtissent du surcoût lié au parallélisme (qui augmente proportionnellement au nombre de tâches actives) et de la délocalisation de certains calculs sur un second processeur (les processeurs de l'architecture embarquant chacun 6 cœurs). On note toutefois que les gains augmentent en même temps que la taille des objets mathématiques, l'intensité arithmétique devenant suffisante pour masquer les latences.

Comme attendu, ces résultats sont constants quelque soit la taille des objets utilisés. Pour des tailles plus petites (inférieures à 200 bits), les gains sont équivalents même si on note une influence plus importante du parallélisme : par exemple, sur 4 cœurs pour une courbe définie sur un corps de taille 128, l'accélération varie de 3,4 à 3,6. En pratique cependant, les tailles utilisées en cryptographie sont supérieures à 160 bits.

La figure 3.5 présente les accélérations obtenues par la méthode sans pré-calcul pour différentes représentations NAF_w . Nous avons également implémenté une version parallèle sur deux tâches de la méthode poids faibles poids forts de la multiplication scalaire. Cet algorithme parcourt la représentation binaire de k de la droite vers la gauche en effectuant pour chaque bit non-nul une addition $Q = Q + P$, et un doublement $P = [2]P$ dans tous les cas. Ces deux opérations peuvent donc être traitées en parallèle (en alternant soigneusement les zones mémoires où stocker le résultat de $[2]P$ pour garantir d'utiliser le bon point P dans l'addition). En théorie, en parallèle cet algorithme calcule donc $[k]P$ dans le même temps que le calcul de l doublements. En pratique cependant, la figure montre que cette méthode n'est pas efficace, l'accélération par rapport à la version séquentielle étant quasiment nulle. Cela vient du fait qu'il faut synchroniser les calculs de l'addition et du doublement pour chaque bit de la représentation pour pouvoir garantir la validité du résultat final.

Les meilleures performances de la méthode sans pré-calcul sont obtenues par l'algorithme utilisant la représentation NAF_2 et sont proches des performances optimales attendues. En effet, on ne peut pas réaliser avec notre méthode la multiplication scalaire en moins de temps que le temps nécessaire à l doublements. En représentation NAF_2 , si l'on pose $D \approx A$, l'algorithme 8 a un coût de $l + l/3$

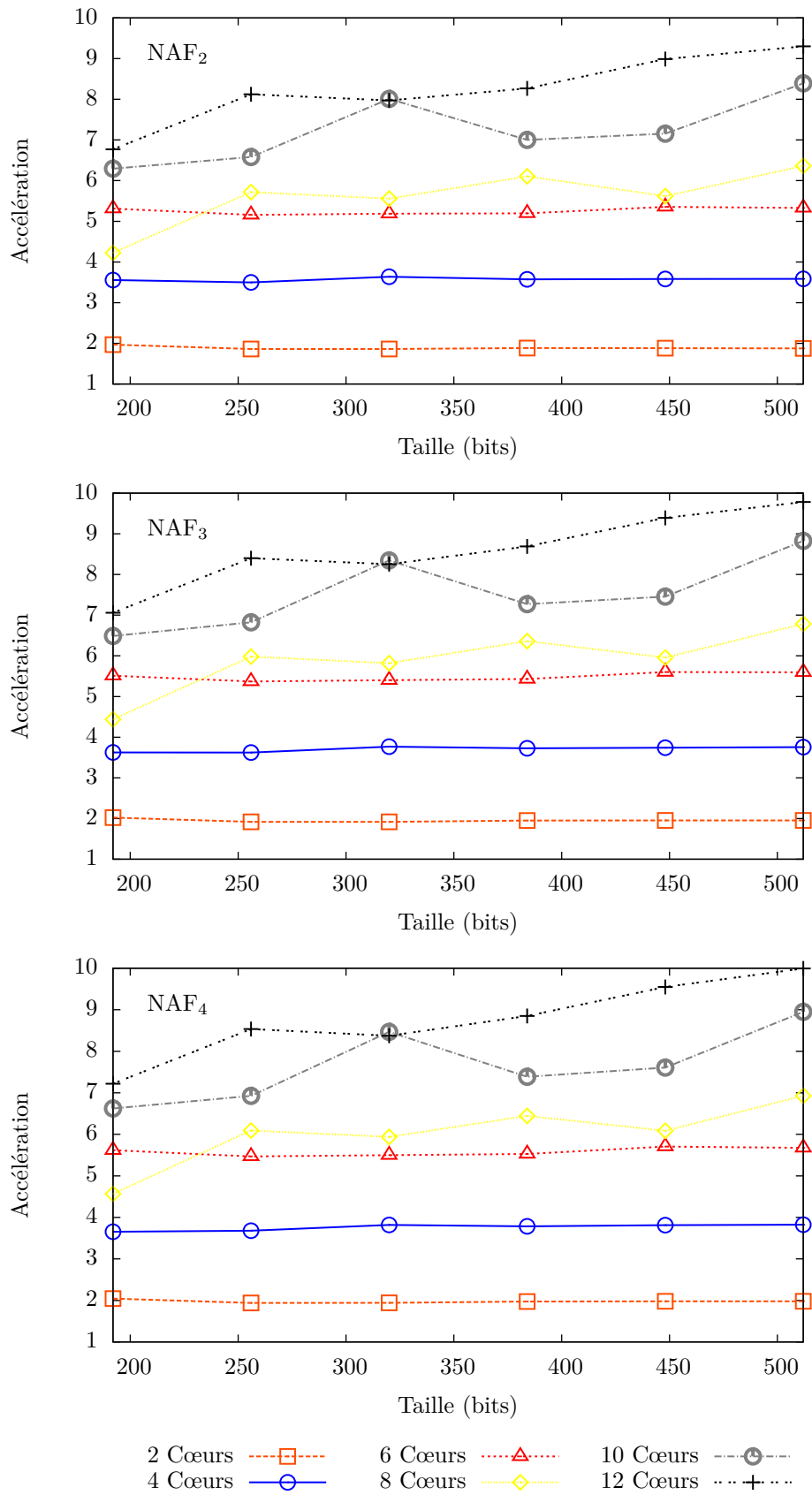


FIGURE 3.4 – Multiplication scalaire parallèle avec pré-calcul

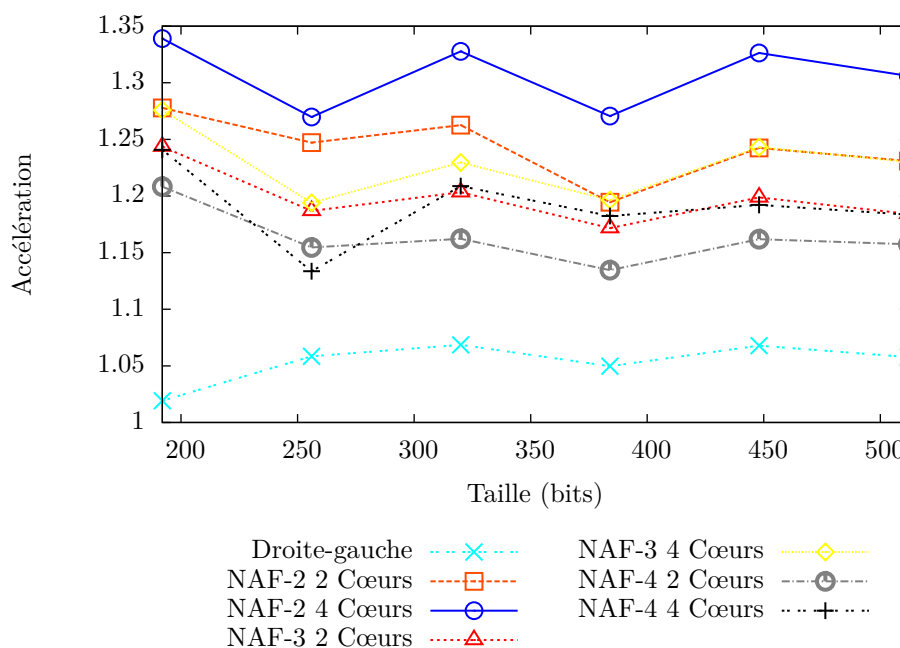


FIGURE 3.5 – Multiplication scalaire parallèle sans pré-calcul

doublements. L'accélération que l'on peut atteindre est alors au plus de 1.33, ce que l'on atteint en pratique.

Selon le même raisonnement, les accélérations théoriques pour NAF_3 et NAF_4 sont au plus égale à 1,25 et 1,2 respectivement, borne également atteinte en pratique.

On peut également comparer nos algorithmes parallèles à la meilleure implémentation séquentielle. Dans nos travaux, l'algorithme séquentiel le plus performant est l'algorithme NAF_4 . L'accélération de NAF_2 sur deux cœurs par rapport à cet algorithme varie de 1,08 à 1,15, sur quatre cœurs elle varie de 1,14 à 1,2. L'accélération de NAF_3 sur deux cœurs varie de 1,14 à 1,19 sur deux cœurs et de 1,15 à 1,22 sur quatre cœurs. Cependant, l'avantage de notre méthode est de s'appliquer à la plupart des algorithmes classiques de multiplication : pour chaque version séquentielle, on pourra lancer en parallèle l'algorithme séquentiel restreint à un certain nombre de bits de k .

3.2 Parallélisme sur les processeurs graphiques

Les algorithmes d'arithmétique multiprécision pour des tailles cryptographiques se prêtent mal à un parallélisme intensif (des dizaines ou des centaines d'unités de calcul). Nous traitons ici de la parallélisation de plusieurs centaines ou milliers de multiplications modulaires ou d'opérations sur les courbes elliptiques. Les threads CUDA vont réaliser un ensemble d'opérations similaires sur des données différentes soit un calcul de type SIMD. Ce type de parallélisme sera utile pour calculer en parallèle des instances d'un protocole de cryptographie asymétrique. On peut penser par exemple à un serveur réalisant des ensembles de vérification de signatures.

Les GPU Tesla utilisent une arithmétique simple précision (32 bits) ou double précision (64 bits). Nous présentons dans ce chapitre une bibliothèque permettant la représentation des entiers multiprécision, les calculs modulaires (addition et multiplication) ainsi que les opérations sur les courbes elliptiques en système de coordonnées jacobiniennes (addition, doublement et multiplication scalaire). Ces travaux ont été publiés dans [41].

La différence fondamentale qui va surgir entre la parallélisation d'instances sur le processeur graphique et la parallélisation d'instances sur les cœurs d'une architecture SMP est la gestion de la mémoire. Dans ce dernier cas, elle est totalement transparente : le coût des communications entre la RAM et les processeurs est généralement négligeable (et négligé) et les différents caches présents sur les processeurs peuvent les amortir. Ce n'est pas le cas des mémoires d'un processeur graphique : les mémoires à accès rapide sont de tailles trop restreintes pour accueillir des nombres de taille importante. En outre, si la mémoire globale de la carte a une taille suffisante, les latences de plusieurs centaines de cycles en font un mauvais candidat pour des calculs qui feront au plus également quelques centaines de cycles.

Notre bibliothèque est générique en termes de taille des entiers et de nombre de calculs à réaliser. Cependant, à cause des problèmes mentionnés précédemment, nous avons focalisé nos expérimentations sur des tailles utilisées pour la cryptographie basée sur les courbes elliptiques (entre 160 et 384 bits).

3.2.1 Représentation des entiers multiprécision

Les entiers multiprécision sont généralement représentés par des tableaux de mots machines dont la taille dépend de l'architecture. Nous analysons ici les spécifications de nos cartes pour déterminer le support que nous utiliserons pour stocker nos entiers multiprécision.

Sur les GPU considérés, les types utilisables sont les entiers de 32 bits ou les flottants-simple précision (32 bits). Ces derniers respectent la norme IEEE-754 et possèdent donc une mantisse de 23 bits. Nous souhaitons réaliser des opérations exactes entre les mots de la représentation.

Remarque 16. Sur nos architectures (CUDA Capability 1.1), les entiers et flottants double-précision ne sont pas supportés.

Sans support 64 bits pour les entiers, on ne peut pas réaliser une multiplication 32 bits par 32 bits sans erreur. Pour avoir un résultat exact sur 32 bits, il faut que les mots de la représentation soient codés sur 16 bits.

De même, pour avoir un résultat exact en utilisant les flottants simple-précision, il faut que les mots de la représentation soient codés sur 11 bits, pour avoir un résultat final de taille inférieure à 23 bits, la taille de la mantisse. On peut cependant réaliser une FMA (fused multiply-and-add : multiplication suivie d'une addition) car le résultat de la multiplication est de taille 22 bits et si l'addition génère une retenue, le résultat final est codé sur 23 bits. Nous donnons dans le tableau 3.1 le nombre de mots n nécessaire à la représentation d'entiers multiprécision de différentes tailles en utilisant ces deux types.

Nous donnons également le nombre de cycles théorique pour la version quadratique de la multiplication multiprécision. Cette opération nécessite n^2 multiplications de mots suivies de $(n-1)^2$ additions en utilisant les entiers et n^2 FMA en utilisant les flottants. Or, en 4 cycles les cœurs du processeur graphique peuvent calculer une FMA exacte sur des opérandes de 11 bits ou une multiplication suivie d'une addition dans le cas des entiers.

Opérandes	Entiers		Flottants	
	n	# cycles	n	#cycles
160-bit	10	724	15	900
192-bit	12	1252	18	1296
224-bit	14	1460	21	1764
256-bit	16	1924	24	2304
384-bit	24	4420	35	4900

TABLE 3.1 – Comparaison du nombre de cycles pour la multiplication et du nombre de mots pour les entiers multiprécision en utilisant les entiers ou les flottants 32 bits

$a =$	a_0	a_1	\dots	a_{n-2}	a_{n-1}
$b =$	b_0	b_1	\dots	b_{n-2}	b_{n-1}
\dots			\dots		
$y =$	y_0	y_1	\dots	y_{n-2}	y_{n-1}
$z =$	z_0	z_1	\dots	z_{n-2}	z_{n-1}

(a) Organisation classique

					padding	
$16k$	a_0	b_0	\dots	y_0	z_0	0
$16(k + \lceil m/16 \rceil)$	a_1	b_1	\dots	y_1	z_1	0
	\dots	\dots	\dots	\dots	\dots	0
	a_{n-2}	b_{n-2}	\dots	y_{n-2}	z_{n-2}	0
	a_{n-1}	b_{n-1}	\dots	y_{n-1}	z_{n-1}	0

(b) Organisation lectures coalescentes

FIGURE 3.6 – Organisation des données sur le CPU et sur le GPU

Suivant le tableau, il est évident que représenter les entiers multiprécision en utilisant des entiers 32 bits en base $\beta = 2^{16}$ est le meilleur choix : moins de mémoire utilisée et des opérations théoriquement plus rapides.

3.2.2 Gestion de la mémoire

Lorsque le GPU sera utilisé pour effectuer un ensemble de m opérations modulaires, les entiers multiprécisions sont transmis depuis la machine hôte et stockés dans la mémoire globale de la carte. Pour obtenir les meilleures performances, il est indispensable de tenir compte de la règle **R5** (voir §1.4.1.5) : arranger les données pour permettre des transactions coalescentes.

La figure 3.6 décrit l'organisation classique des entiers stockés sous forme de tableau à m lignes et n colonnes 3.6a et l'organisation nécessaire en mémoire globale pour garantir des transaction coalescentes 3.6b.

Pour assurer l'alignement requis, deux mots consécutifs a_i et a_{i+1} d'un même entier sont distants de l adresses de mots où l est le premier multiple de 16 supérieur à m . Il faut également s'assurer dans le code que tous les threads d'un demi warp vont accéder aux données au même moment : lecture et écriture doivent se faire en début de kernel ou à la suite d'une barrière de synchronisation.

La règle **R4** contraint un accès minimal à la mémoire globale. La première opération effectuée par le kernel est donc un transfert des entiers dans des mémoires plus rapides. Notre bibliothèque propose un stockage dans trois mémoires différentes : la mémoire partagée, la mémoire locale et les registres.

Remarque 17. La mémoire locale est en réalité une partie de la mémoire globale. Nous verrons que les performances en utilisant cette mémoire sont moindres que celles utilisant les mémoires à accès rapide.

En mémoire partagée et en mémoire locale, les données sont stockées de manière classique par des tableaux de n mots machine consécutifs. En revanche, les registres n'étant pas indexables il n'est pas possible de définir des tableaux les utilisant. Ce problème pourrait être contourné en définissant explicitement du code pour des tailles fixées. Dans ce cas, un mot serait représenté par une variable. Cependant, cette méthode ne permet pas de garder l'aspect générique de notre bibliothèque.

La généricité du C++ nous permet néanmoins de définir une structure récursive dans laquelle le nombre de variables sera fixé à la compilation grâce aux méthodes de *template metaprogramming*. Cette structure est définie par le code donné à la figure 3.7.

```

template <uint n>
struct IntegerReg : IntegerReg<n-1> {
    uint word;
    IntegerReg() : IntegerReg<n-1>() {word=0;}
};
template <>
struct IntegerReg<1> {
    uint word;
    IntegerReg() {word=0;}
};
#define AT(x, i) x.IntegerReg<i+1>::word

```

FIGURE 3.7 – Structure pour la représentation des entiers multiprécision en registre avec un accès indexable

Le paramètre n permet d'accéder au n -ième mot de l'entier considéré en utilisant la macro AT. Cependant, en C++ les paramètres génériques doivent être connus à la compilation, ce qui interdit l'utilisation de boucles classiques pour parcourir la structure. Il est donc nécessaire de définir des « boucles » où les indices seront connus au moment de la compilation. Une fois encore, on utilise une structure récursive avec deux paramètres génériques : le début et la fin de la boucle. Une spécialisation de cette structure permet d'arrêter la récursion. Ce type de structure est illustré par la figure 3.8 qui présente l'opérateur de comparaison `egal` entre entiers stockés dans les registres.

```

template<uint beg, uint end>
struct IntegerLoop {
    template<uint n>
    static bool egal(const IntegerReg<n>& a, const IntegerReg<n>& b) {
        return AT(a, beg)==AT(b, beg) && IntegerLoop<beg+1,end>::egal(a, b);
    }
};
template<uint end>
struct IntegerLoop<end, end> {
    template<uint n>
    static bool egal (const IntegerReg<n>& a, const IntegerReg<n>& b) {
        return AT(a, end)==AT(b, end);
    }
};

```

FIGURE 3.8 – Opérateur de comparaison pour les entiers stockés en registres.

Les boucles sont donc totalement déroulées dans le code machine généré par le compilateur CUDA. Ce dernier est cependant incapable de gérer totalement cette technique : même sur du code très simple, il peut renvoyer une erreur *ran out of registers* indiquant que l'ensemble des registres est utilisé. En réalité, il est incapable de gérer proprement leur utilisation pour du code machine trop long.

Le nombre de registres étant très limité, nous stockons deux mots de 16 bits dans un registre de 32 bits.

Dans la suite, **R** désignera la version du code utilisant un stockage des entiers multiprécision dans les registres, **LM** un stockage en mémoire locale et **SM** en mémoire partagée.

3.2.3 Arithmétique modulaire

Notre bibliothèque permet trois opérations modulaires : l'addition, la soustraction et la multiplication sur des données différentes d'un thread à l'autre. Il est donc nécessaire de stocker le modulo p pour chaque thread. La méthode de Montgomery utilisée pour la multiplication modulaire implique le calcul d'une constante (voir §2.3.2.4). Comme nous le verrons, un seul mot machine est nécessaire pour la stocker.

3.2.3.1 Addition et soustraction

Pour l'addition et la soustraction modulaire, la bibliothèque utilise les algorithmes classiques d'addition entière et de soustraction entière à propagation de retenue (§2.2.2) suivis potentiellement d'une correction pour avoir un résultat modulo p . Les deux algorithmes sont réalisés *en place*, c'est-à-dire $a \pm b \bmod p$ avec un registre en sus pour la propagation de la retenue.

Si $\beta = 2^{16}$, savoir si une retenue r_i a été générée par $a_i + b_i + r_{i-1}$ consiste à lire simplement le 16^e bit de a_i . De même, $a_i - b_i - r_{i-1}$ produit une retenue si a_i est négatif. Dans les deux cas, l'opération est suivie de l'application d'un masque `0xFFFF` pour obtenir un mot réduit modulo β .

Pour la version registre où l'on stocke deux mots de 16 bits consécutifs dans un même registre 32 bits, trouver la retenue générée par $a_i + b_i + r_{i-1}$ consiste à tester si le résultat obtenu par l'addition $a_i + b_i + r_{i-1}$ est inférieur à l'ancienne valeur a_i . Si le test est vrai, alors une retenue a été générée. Pour la soustraction, il suffit de vérifier si le résultat obtenu par $a_i - b_i - r_{i-1}$ est supérieur à l'ancienne valeur a_i .

3.2.3.2 Multiplication

Les mémoires rapides des GPU ont une taille limitée. Pour réduire son utilisation, nous avons utilisé la version FIOS de l'algorithme de Montgomery (§2.3.2.4) qui nécessite seulement trois registres en plus des deux opérandes et de la zone mémoire de taille n dédiée au résultat. Cet algorithme requiert le calcul de $-1/p \bmod \beta$ qui pourra être stocké sur un seul registre.

Pour les versions **LM** et **SM**, les multiplications atomiques de mots sont réalisées par la fonction `__umul24(x,y)` qui multiplie les 24 bits de poids faible des deux opérandes et produit un résultat de 32 bits (donc avec perte de précision). Cependant, en base $\beta = 2^{16}$ le résultat est de taille 32 bits, donc sans perte de précision.

Pour la version **R**, la multiplication atomique qui multiplie deux mots est décomposée en quatre multiplications utilisant la fonction `__umul24(x,y)` avec des paramètres de 16 bits.

3.2.3.3 Résultat expérimentaux

Pour évaluer les performances de la partie arithmétique modulaire de notre bibliothèque, nous comparons tout d'abord les temps de calcul obtenus pour les différentes opérations arithmétiques en utilisant les trois stockages de données utilisés. L'ensemble de nos tests sont réalisés sur une carte GeForce 9800GTX2. Nous comparons ensuite les performances entre une version GPU et une version CPU.

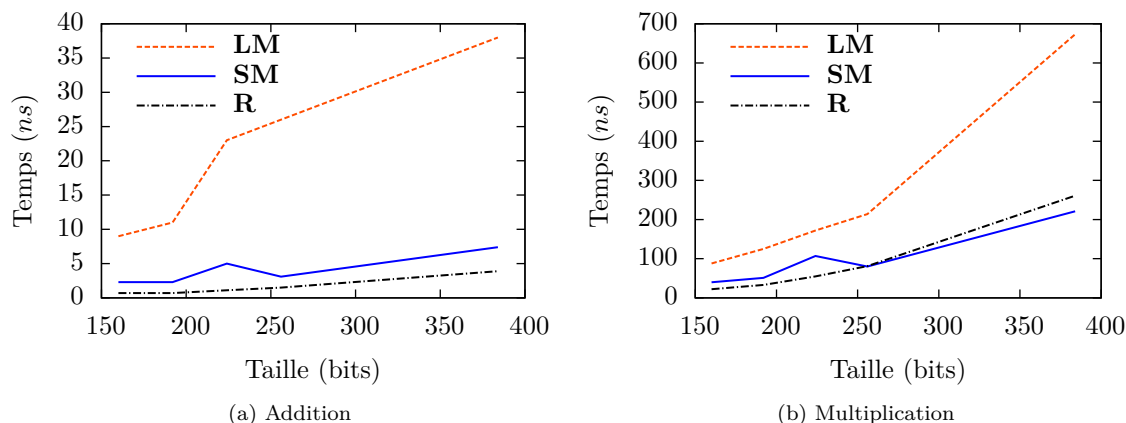


FIGURE 3.9 – Temps de calculs pour l'addition et la multiplication en fonction des mémoires utilisées

Comparaison des performances en fonction de la mémoire utilisée

La figure 3.9 présente les temps de calcul obtenus par les trois implémentations pour l'addition et la multiplication modulaire entre entiers de taille 160 à 384 bits. Les données sont générées aléatoirement et occupent l'ensemble des n mots. Les kernels sont composés de 1024 threads découpés en 64 blocs.

Avant le calcul les threads copient les opérandes qu'ils vont utiliser dans la zone mémoire cible (locale, partagée, registres). Ils exécutent ensuite 10000 opérations telles que $a+ = b \bmod p$ et $a \times = b \bmod p$.

Il est clair que les mémoires utilisées ont un impact direct sur les performances de l'application. L'utilisation des registres permet un gain de performances pour des entiers de taille 160 à 256 bits par rapport la version utilisant la mémoire partagée. Le fait que cette dernière devienne ensuite la plus rapide pour la multiplication s'explique par la mauvaise gestion du code lors d'un utilisation trop importante des registres. Chaque thread peut utiliser au plus 128 registres de 32-bits.

Comparaison CPU/GPU

Nous comparons maintenant les performances de notre bibliothèque à la bibliothèque mpFq, qui est actuellement l'implémentation la plus rapide pour des calculs modulaires sur CPU entre opérandes de tailles réduites. Elle réalise le même ensemble de calculs que la version GPU en séquentialisant 1024 boucles de 10000 opérations chacune. Ce code est lancé sur un cœur d'un processeur Intel Core(TM)2 Duo E8400 cadencé à 3GHz.

Pour une taille d'entiers inférieure à 384 bits, la version GPU est plus performante que la version CPU. Cependant, ces performances restent modérées et loin des performances que l'on pourrait attendre en utilisant 128 processeurs au lieu d'un seul. En réalité, la fréquence du processeur utilisé (3GHz) est bien plus importante que celle des unités de calcul du GPU ($\approx 600\text{MHz}$). De plus, nos opérations atomiques sont réalisées entre entiers de taille 2^{16} tandis que les opérations sur l'hôte sont réalisées en simple précision (32 bits). Pour une multiplication 32 bits par 32 bits nous devons donc réaliser 4 opérations. Enfin, la bibliothèque mpFq utilise un code optimisé pour bénéficier des instructions SIMD SSE-2 pour l'arithmétique entière multiprécision.

3.2.4 Arithmétique des courbes elliptiques

Pour évaluer les performances de notre bibliothèque d'arithmétique sur GPU, nous avons implémenté la multiplication scalaire $[k]P$ en système de coordonnées jacobiniennes.

3.2.4.1 Courbes elliptiques sur GPU

Notre bibliothèque propose des fonctions pour l'addition, le doublement et la multiplication scalaire dans $E(\mathbb{F}_p)$. Chaque thread du kernel réalise un ensemble d'opérations dans $E(\mathbb{F}_p)$ sur des données

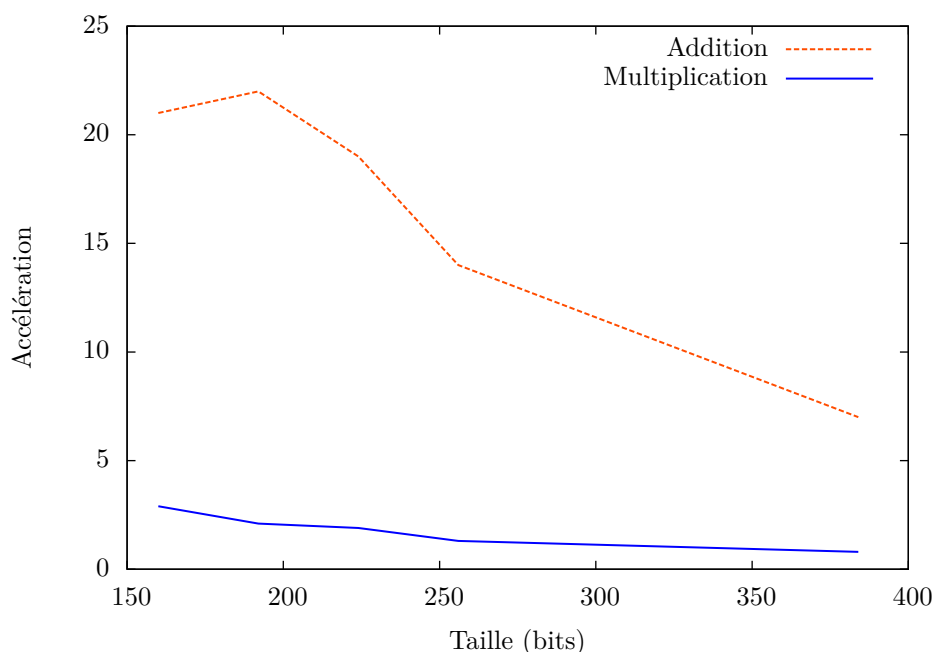


FIGURE 3.10 – Accélération de la version GPU par rapport à la version CPU

différentes. Un thread doit donc connaître un ensemble de données qui lui sont propres :

- p : la caractéristique du corps de définition de la courbe ;
- a : paramètre de la courbes nécessaire aux opérations de groupe ;
- $P = (X_P, Y_P, Z_P)$: au moins un point de la courbe sur lequel les traitements vont être réalisés (potentiellement plusieurs points selon les traitements effectués, par exemple deux points pour l'addition) ;
- k : paramètre de multiplication scalaire le cas échéant.

En plus de ces données, suivant l'opération réalisée chaque thread doit pouvoir stocker un point $Q = (X_Q, Y_Q, Z_Q)$ correspondant au résultat du calcul.

Le flot de données pour toutes les opérations décrites par la suite est le même :

1. envoi des données du CPU au GPU en respectant la règle **R5** ;
2. copie des données de la mémoire globale vers la mémoire cible (**LM, SM, R**) en utilisant des lectures coalescentes ;
3. conversion de a et des coordonnées de P_i en représentation de Montgomery ;
4. calcul de l'opération (addition, doublement ou multiplication scalaire) ;
5. conversion des coordonnées de Q_i de la représentation de Montgomery à la représentation classique ;
6. transfert des Q_i du GPU au CPU.

3.2.4.2 Addition et doublement dans $E(\mathbb{F}_p)$

Nous avons implémenté les opérations d'additions et de doublement de points en coordonnées jacobienne qui utilisent les opérateurs modulaire définis précédemment. Le doublement de point nécessite deux entiers de taille N pour stocker des valeurs intermédiaires, l'addition en nécessite trois. Ces variables sont définies dans la même zone mémoire que les données.

Comparaison LM-SM-R

Les temps de calcul obtenus pour les trois implémentations sont présentés à la figure 3.11. Le kernel est composé de 1024 threads divisés en 64 blocs de taille 16. Chaque thread exécute une boucle de 100 opérations qui réutilisent les résultats intermédiaires.

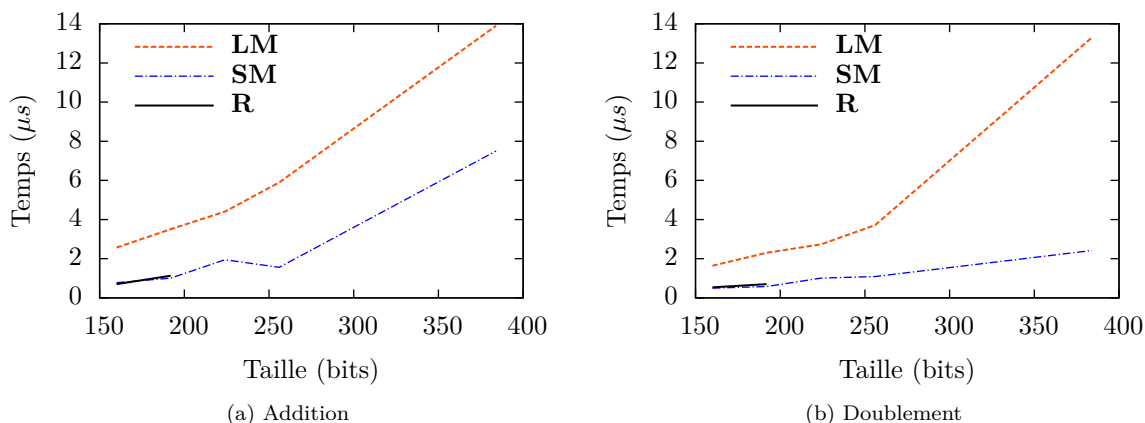


FIGURE 3.11 – Temps de calcul pour l'addition et le doublement dans $E(\mathbb{F}_p)$ en utilisant **LM**, **SM** et **R**.

Le nombre limité de registres n'a pas permis de lancer la version utilisant les registres pour des entiers de taille supérieure à 192 bits. L'utilisation de la mémoire locale donne une fois de plus les moins bonnes performances en raison des latences d'accès aux données. La version utilisant la mémoire partagée est toujours la meilleure hormis pour 160 bits où la version registres reste la plus rapide. Cependant, la taille réduite de la mémoire partagée limite le nombre de threads par bloc à 16, ce qui divise l'occupation de la carte par deux entre $N = 384$ et $N = 256$ pour l'addition. Pour le doublement en revanche l'occupation de la carte est la même.

Comparaison CPU/GPU

Pour comparer les performances entre notre meilleure implémentation GPU et à une version CPU, nous avons couplé PACE avec mpFq comme bibliothèque de représentation et d'arithmétique bas niveau. Cette implémentation réalise les mêmes tests (1024×100 opérations) sur les mêmes données. La figure 3.12 présente l'accélération de la version GPU par rapport à la version CPU. Pour les tailles considérées, notre implémentation est toujours au moins deux fois plus rapides que la version CPU pour le doublement de point, mais on note une décroissance constante de l'accélération. Pour l'addition en revanche, notre implémentation est moins de deux fois plus rapide et devient moins performantes pour 384 bits. Cette différence de performances entre les deux opérations s'explique par les opérations modulaires qui les composent : il y a 9 multiplications pour le doublement et 16 multiplications pour l'addition. Or, l'accélération de la multiplication de Montgomery est au plus de 3 entre les deux versions.

3.2.4.3 Multiplication scalaire

Si les GPU semblent avoir un intérêt limité pour l'arithmétique modulaire et l'arithmétique des courbes elliptiques, nous avons voulu savoir s'il en va de même pour les calculs de plus haut niveau. Nous avons donc implémenté la multiplication scalaire $[k]P$ en CUDA.

La figure 3.13 présente le débit en nombre de multiplications scalaires par seconde obtenu par la version CPU et par la version GPU utilisant la mémoire partagée. Dans ces tests, le scalaire k est de taille N et de poids de Hamming $N/2$. La version GPU est au plus trois fois plus rapide que la version CPU, et devient moins performante pour $N = 384$. Une fois encore, la faible ratio entre les version CPU et GPU de la multiplication de Montgomery se reporte sur les opérations arithmétiques de plus haut niveau. Il est difficile d'établir des comparaisons avec d'autres bibliothèques similaires, car les performances obtenues sont dépendantes de l'architecture considérée. Cependant, dans [95], les auteurs annoncent un débit de $1412.6 [k]P/s$ sur une carte GeForce 8800GTS en utilisant un système de coordonnées mixte (affine-jacobien). Notre implémentation est un peu meilleure ($1972[k]P/s$) mais sur une carte un peu plus performante.

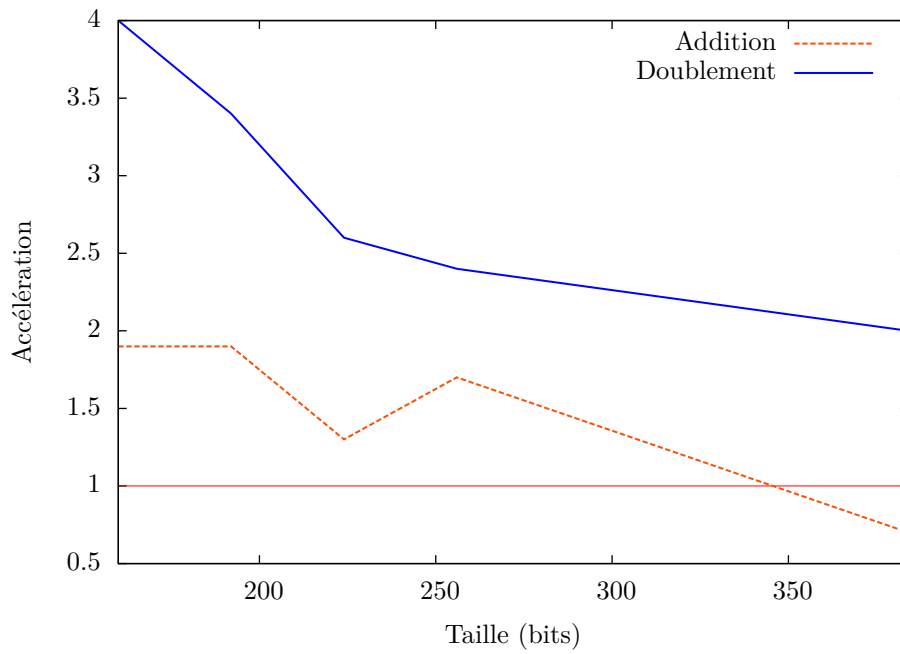


FIGURE 3.12 – Accélération de **SM** par rapport à la version CPU PACE et mpFq

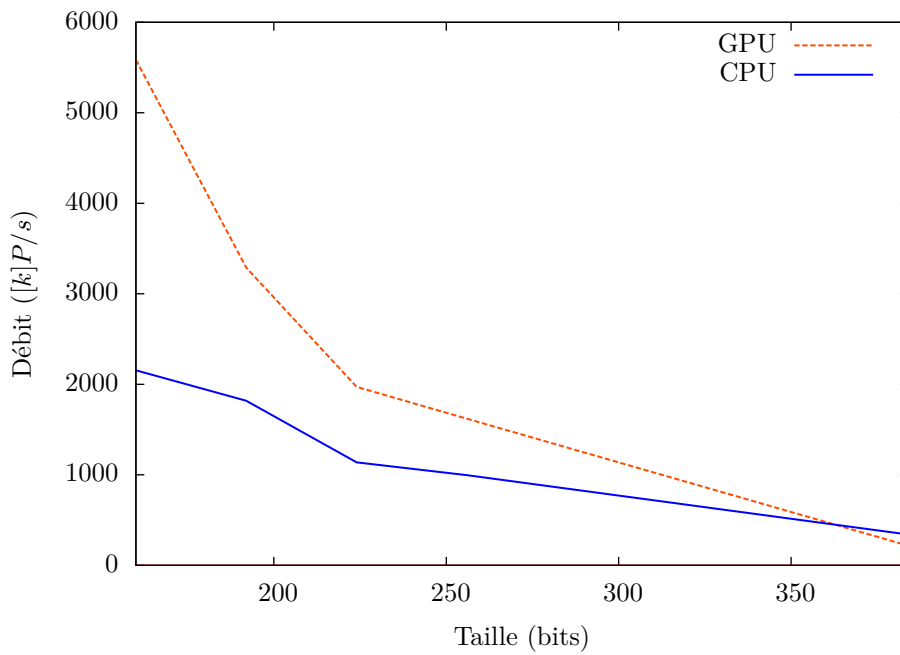


FIGURE 3.13 – Comparaison des débits en $[k]P/s$ des versions CPU et GPU

3.3 Conclusion et perspectives

Nous faisons ici le bilan des travaux présentés dans ce chapitre et présentons les perspectives qui nous semblent les plus prometteuses.

3.3.1 Architecture SMP

Nous avons implémenté les versions parallèles des opérateurs classiques d'arithmétique entière. Ceux-ci nous permettent d'écrire facilement du code parallèle pour les opérateurs des arithmétiques de plus haut niveau, en particulier les opérateurs d'arithmétique modulaire.

Comme attendu, la première conclusion que l'on peut tirer est que le surcoût lié au parallélisme est non négligeable pour des opérateurs arithmétiques entre entiers de taille réduite. En particulier, si les coûts liés au lancement et à l'arrêt des tâches peuvent être amortis, les barrières de synchronisation sont excessivement coûteuses (leur coût dépasse celui de l'addition pour des entiers d'une centaine de milliers de bits). S'il est difficile de se passer de ces synchronisations, nous montrons dans le chapitre suivant qu'un parallélisme à un niveau arithmétique supérieur permet des tâches plus indépendantes que celles utilisées pour les opérateurs d'arithmétique entière.

Nous avons utilisé le schéma de multiplication quadratique et implémenté la multiplication parallèle de Karatsuba. Un schéma de type Toom-Cook nous permettrait de réduire le nombre d'unités de calcul (ou d'augmenter la taille du découpage), mais les gains obtenus seront sans doute similaires à ceux de la multiplication quadratique.

On peut se poser la question de l'apport du parallélisme asynchrone pour ces opérateurs, en particulier pour des schémas de découpage sous-quadratique. En utilisant Cilk par exemple, on pourra tester un parallélisme dynamique où les tâches utilisent un modèle fork-join : elles se répartissent le travail en fonction des calculs qui peuvent être effectués indépendamment. Il reste cependant à quantifier l'impact de l'ordonnancement dynamique sur les performances, ainsi que les surcoûts introduits localement pour lancer ou fusionner les tâches.

La prochaine étape dans ces travaux consiste naturellement à implémenter la méthode de multiplication FFT parallèle. En théorie, on peut espérer une multiplication de complexité parallèle égale à $M(n)/T$ sur T processeurs. En pratique cependant, il n'est pas sûr que cette méthode soit efficace pour les tailles visées (quelques milliers de bits).

Au niveau des courbes elliptiques, nous proposons une première approche pour la parallélisation de la multiplication scalaire, qui permet de réduire le coût du calcul et de lancer des tâches homogènes avec des gains de performances intéressants en pratique. Cependant, dans cette approche si k fait N bits, la complexité du calcul parallèle ne peut pas être inférieure à $N - 1$ doublements.

Une autre technique pour paralléliser cette opération est de trouver un endomorphisme ϕ efficace pour séparer le calcul $[k]P$ en $[k]P = [k_1]P + [k_2]\phi(P) + \dots + [k_n]\phi^n(P)$ où les calculs $[k_1]P$, $[k_2]\phi(P)$, \dots , $[k_n]\phi^n(P)$ sont réalisés dans le même temps. Cette méthode a été introduite par Gallant, Lambert et Vanstone dans [35]. Dans le cas d'extensions de corps fini \mathbb{F}_{p^m} , un endomorphisme efficace est le Frobenius [34] :

$$\begin{aligned} \phi : \quad E &\rightarrow E \\ (x, y) &\mapsto (x^p, y^p). \end{aligned}$$

Cette méthode est particulièrement efficace pour les courbes définies sur les extensions de corps binaire.

Si la séquentialité de ce type d'algorithme est difficile à gérer dans le cas général, on peut cependant paralléliser l'arithmétique des courbes au niveau des opérations dans le groupe $E(\mathbb{F}_p)$ (voir les perspectives du chapitre 5). Utiliser ici le parallélisme asynchrone prendrait en particulier tout son sens : les opérations indépendantes pourraient être lancées en parallèle *à la volée*, ce type de parallélisme étant dynamique. Il en va de même pour certains algorithmes de multiplication scalaire : on pourra par exemple réaliser l'ensemble des doublements séquentiellement et lancer les additions en parallèle lorsque c'est possible.

3.3.2 Processeurs graphiques

Nous avons développé une bibliothèque d'arithmétique modulaire multiprécision et d'arithmétique des courbes elliptiques pour les GPU Nvidia utilisant la CUDA Capability 1.1. Pour cette première implémentation, on note un gain par rapport au CPU, mais bien moins important que celui attendu.

Il existe plusieurs causes à ces performances moyennes :

- le facteur entre la fréquence des CPU et la fréquence des SP est environ 5 ;
- notre représentation utilise une représentation d'entiers sur 16 bits avec des opérations 16-bits par 16-bits donne 32 bits, tandis que le CPU est capable de produire des opérations avec une précision double ;
- la gestion de la mémoire : il faut aller lire les données en mémoire globale, puis à la fin du calcul y stocker les résultats, avec des latences de plusieurs centaines de cycles. De plus, l'accès à la mémoire partagée n'est pas réellement gratuit. Nvidia annonce des latences d'une dizaine de cycle.

En réalité, ce type de parallélisation sera plus efficace pour des calculs plus haut niveau (comme par exemple la factorisation ECM proposée dans [7]) où chaque tâche effectuera un grand nombre d'opérations. Ainsi, l'intensité arithmétique permettra d'amortir les latences dues aux transferts mémoire.

Proposer une bibliothèque haute-performance générique semble difficile car les architectures évoluent en permanence. La nouvelle architecture Fermi proposée par Nvidia par exemple corrige certaines contraintes posées par les architectures Tesla. Dans ce cadre, implémenter cette bibliothèque dans des langages comme OpenCL ou HMPP serait une perspective intéressante pour le calcul multiprécision haute-performance sur GPU. Cependant, ces langages sont des langages d'abstraction. On ne peut pas garantir que le code spécifique qu'ils génèrent pour l'architecture considéré sera optimal. Pour nos applications, cela pourra poser problème au vu des temps de calcul très faibles des différentes opérations.

CHAPITRE 4

LA MULTIPLICATION MODULAIRE MULTIPARTITE

Sommaire

4.1 Réductions partielles	64
4.1.1 Réduction partielle de Montgomery	64
4.1.2 Réduction partielle de Barrett	65
4.2 Définition de la multiplication multipartite	67
4.2.1 La multiplication bipartite	67
4.2.2 La multiplication quadripartite	67
4.2.3 Généralisation	68
4.2.4 Analyse de complexité	70
4.3 Complexité parallèle de la multiplication multipartite	72
4.4 Ordonnement des calculs pour l'approche modulaire	72
4.4.1 Regroupement des produits médians	73
4.4.2 Diminuer le nombre de réductions modulaires	73
4.4.3 Méthodes heuristiques pour l'ordonnement des calculs	76
4.4.4 Conclusion	78
4.5 Modèle réaliste : un parallélisme à deux niveaux	79
4.6 Résumé des complexités parallèles de la multiplication modulaire	80
4.7 Résultats expérimentaux	81
4.7.1 Multiplication modulaire parallèle	81
4.7.2 Exponentiation parallèle modulaire	82
4.8 Conclusion et perspectives	88
4.8.1 Un carré multipartite	88
4.8.2 Une réduction multipartite	89

Dans le chapitre précédent, nous avons présenté une parallélisation de la multiplication modulaire au niveau de l'arithmétique entière multiprécision. Dans ce chapitre, nous nous intéressons à la parallélisation de l'arithmétique au niveau modulaire.

La multiplication bipartite introduite par Kaihara et Tagaki dans [49] est un parfait exemple d'une parallélisation efficace à ce niveau. En effet, le calcul $ab\beta^{-n/2} \bmod p$ est divisé en deux produits modulaires indépendants $ab_1 \bmod p$ et $ab_0\beta^{-n/2} \bmod p$ avec $b = b_1\beta^{n/2} + b_0$. Sur une architecture parallèle à deux unités de calcul, la complexité parallèle de cette méthode est inférieure aux complexités des algorithmes classiques de réduction (Barrett et Montgomery), même si ceux-ci utilisent une arithmétique entière parallèle.

Nous allons généraliser cette méthode en découpant les deux opérands en k morceaux [40]. Dans un schéma de multiplication quadratique, cela revient à calculer k^2 produits modulaires indépendants de type $a_i b_j \beta^e \bmod p$ avec $-n/2 \leq e \leq n/2$. La complexité totale de l'opération est ainsi augmentée, car nous introduisons des calculs redondants. Cependant, la complexité parallèle théorique de la multiplication modulaire est réduite, car elle correspond à la complexité du calcul du produit modulaire $a_i b_j \beta^e \bmod p$ le plus coûteux. En pratique, nous étudions les avantages et les inconvénients de cette méthode. En effet, dans un modèle réaliste, il est nécessaire de prendre en compte que le nombre de processeurs disponibles sur une architecture n'est pas infini. De plus, une parallélisation est toujours possible au niveau de l'arithmétique entière. Cela signifie d'une part que nous devons comparer notre nouvelle multiplication *multipartite* aux méthodes existantes tirant parti de ce parallélisme et d'autre part que nous devons considérer pour notre approche un double parallélisme : au niveau modulaire et au niveau multiprécision.

Dans ce chapitre, nous commençons par définir des versions modifiées des réductions de Montgomery et de Barrett, appelées **réductions partielles**. Ces méthodes ont pour but d'éliminer non pas n mais $t \leq n$ mots de poids faible (ou de poids fort) d'un entier c en garantissant un résultat $r \equiv c \bmod p$. Elles permettent en particulier de calculer les produits partiels $a_i b_j \beta^e \bmod p$ et d'exprimer leur complexité. Ensuite, nous présentons notre approche et classons les produits partiels en trois types. Grâce aux algorithmes de réduction partielle, nous analysons la complexité séquentielle théorique de la multiplication multipartite, puis sa complexité parallèle en étudiant différents ordonnancements. Enfin, nous présentons notre implémentation et nos résultats expérimentaux qui illustrent l'intérêt de la méthode pour des entiers de taille cryptographique.

4.1 Réductions partielles

La multiplication multipartite nécessite la réduction d'un certain nombre de produits de type $a_i b_j \beta^e$ où $-n/2 \leq e \leq n/2$. L'algorithme de Montgomery effectue la réduction $ab\beta^{-n} \bmod p$. Pour réduire nos produits lorsque $e < 0$, nous proposons une réduction *partielle* de Montgomery. L'algorithme de Barrett effectue la réduction $ab \bmod p$ lorsque ab est de taille $2n$. Pour certaines valeurs de e , nos produits seront de taille supérieure à n , et donc supérieur à p . Nous proposons une réduction *partielle* de Barrett, qui élimine non pas n mais t mots de poids fort.

4.1.1 Réduction partielle de Montgomery

La réduction classique de Montgomery calcule $r \equiv c\beta^{-n} \bmod p$ où c est un entier de taille inférieure ou égale à $2n$ en éliminant les n mots de poids faibles de c , comme illustré par la figure 4.1a. L'idée de la réduction partielle de Montgomery est de calculer $r \equiv c\beta^{-t} \bmod p$, c'est-à-dire d'éliminer seulement t mots de poids faible de c . De plus, si c est de taille m , elle doit garantir $0 \leq r < \beta^{m-t}$ si $m - t \geq n$ (figure 4.1b) ou $r < \beta^n$ dans le cas contraire. Cette réduction est obtenue en calculant q tel que $c + qp$ est un multiple de β^t . La valeur $(c + qp)/\beta^t$ est donc un entier. La réduction partielle de Montgomery est décrite par l'algorithme 12.

Algorithme 12: Réduction Partielle de Montgomery

Données : $0 < p < \beta^n$ avec $(p, \beta) = 1$, $0 \leq c < p^2$, $c < \beta^m$, $t \leq n$
Résultat : $r \equiv c\beta^{-t} \pmod{p}$ avec $0 \leq r < \max(\beta^{m-t}, \beta^n)$

- 1 $\mu = -p^{-1} \bmod \beta^t$ ▷ Précalcul
- 2 $q \leftarrow \mu c \bmod \beta^t$
- 3 $r \leftarrow (c + qp) / \beta^t$
- 4 **si** $r \geq \max(\beta^{m-t}, \beta^n)$ **alors** $r \leftarrow r - p$
- 5 **retourner** r

Théorème 8. *L'algorithme 12 est correct et sa complexité est :*

$$\begin{cases} M(m, t) + M(n, t) & \text{si } m < t \\ M(t) + M(n, t) & \text{sinon} \end{cases} \quad (4.1)$$

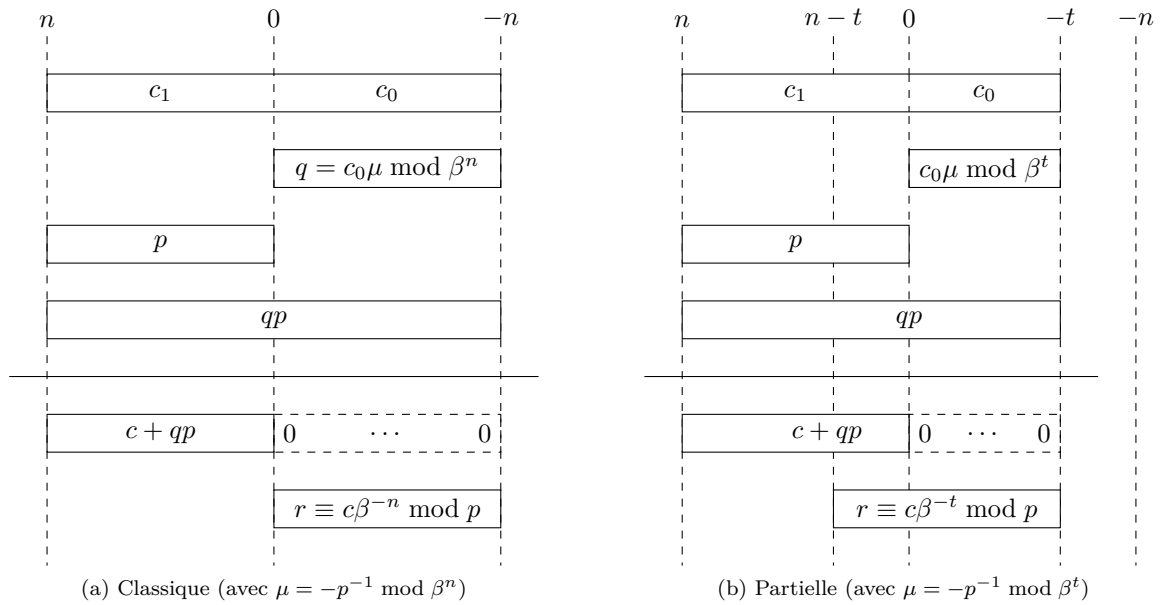


FIGURE 4.1 – Réductions de Montgomery

Démonstration. Comme $q = \mu c = -p^{-1}c \bmod \beta^t$, on a $c + qp \equiv 0 \pmod{\beta^t}$. La division à la ligne 3 est donc exacte et $r \equiv c \beta^{-t} \bmod p$. Comme $0 \leq c < \beta^m$ et $0 \leq q < \beta^t$, $0 \leq c + qp < \beta^m + \beta^t p$. La division par β^t à la ligne 3 donne $0 \leq r < \beta^{m-t} + p$. Dans les deux cas considérés, une seule soustraction est donc requise à la ligne 4 pour garantir $0 \leq r < \max(\beta^{m-t}, \beta^n)$.

Le calcul du quotient à la ligne 2 est réalisé par une multiplication entre les t mots de poids faible de c si $m > t$ ou les m mots de c si $m < t$ et μ de taille t . Cette multiplication a pour complexité $M(t)$ dans le premier cas, $M(t, m)$ dans le second. Dans tous les cas, q est de taille t . Le calcul qp à la ligne 3 se fait donc en $M(n, t)$ car p est de taille n . \square

4.1.2 Réduction partielle de Barrett

La réduction de Barrett classique (§2.3.2.5) annule les n mots de poids fort de $c = ab < \beta^{2n}$ tel que $r \equiv c \pmod{p}$ et $r < p < \beta^n$, comme illustré par la figure 4.2a. La réduction partielle de Barrett consiste à annuler seulement t mots de poids fort de c pour garantir un reste $r \equiv c \pmod{p}$ tel que $0 \leq r < \beta^{m-t}$ avec $t \leq m - n$ (figure 4.2b). Cette réduction partielle nécessite le calcul des t mots de poids forts du quotient q de Barrett tels que :

$$q = \left\lfloor \frac{\left\lfloor \frac{c}{\beta^{m-t}} \right\rfloor \left\lfloor \frac{\beta^{n+t}}{p} \right\rfloor}{\beta^t} \right\rfloor \beta^{m-n-t} \quad (4.2)$$

La réduction partielle de Barrett est obtenue avec l'algorithme 13, détaillé à la figure 4.2b.

Remarque 18. Dans cette réduction partielle, la constante ν est donc égale à $\nu = \lfloor \beta^{n+t}/p \rfloor$.

Algorithme 13: Réduction partielle de Barrett

Données : $p < \beta^n$, $0 \leq c < p^2$, $c = c_1 \beta^{m-t} + c_0 < \beta^m$ avec $0 \leq c_0 < \beta^{m-t}$, $t \leq m - n$

Résultat : $r \equiv c \pmod{p}$ with $0 \leq r < \beta^{m-t}$

- 1 $\nu \leftarrow \lfloor \beta^{n+t}/p \rfloor$ ▷ Précalcul
 - 2 $q \leftarrow \lfloor c_1 \nu / \beta^t \rfloor \beta^{m-n-t}$
 - 3 $r \leftarrow c - qp$
 - 4 **tant que** $r \geq \beta^{m-t}$ **faire** $r \leftarrow r - \beta^{m-n-t} p$
 - 5 **retourner** r
-

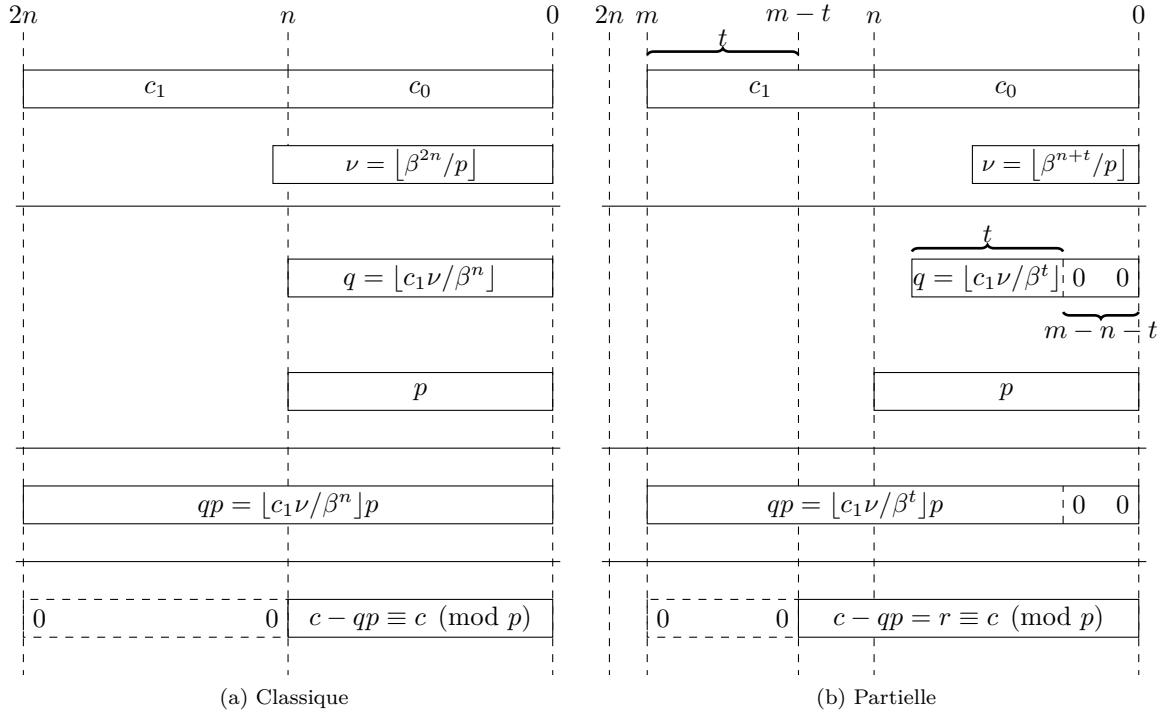


FIGURE 4.2 – Réductions de Barrett

Théorème 9. *L'algorithme 13 est correct et l'étape 4 est exécutée au plus deux fois. Sa complexité est :*

$$\begin{cases} M(s, t) + M(n, t) & \text{si } s < t \\ M(t) + M(n, t) & \text{sinon} \end{cases} \quad (4.3)$$

Démonstration. Comme toutes les opérations consistent à ajouter à c des multiples de p on a $r \equiv c \pmod{p}$. Il reste donc à prouver la relation $0 \leq r < \beta^{m-t}$. Comme $\nu \leq \beta^{n+t}/p$ et par définition $c = c_1\beta^{m-t} + c_0$ avec $0 \leq c_0 < \beta^{m-t}$, on a

$$q \leq \frac{c_1\nu}{\beta^t} \beta^{m-n-t} \leq \frac{c_1\beta^{m-t}}{p} \leq \frac{c}{p}$$

d'où $r = c - qp \geq 0$. De plus, comme $\nu = \lfloor \beta^{n+t}/p \rfloor$ et $q = \lfloor c_1\nu/\beta^t \rfloor \beta^{m-n-t}$, on a $\nu \geq \beta^{n+t}/p - 1$ et $q > (c_1\nu/\beta^t - 1)\beta^{m-n-t}$, ce qui implique $\nu p \geq \beta^{n+t} - p$ et $\beta^t q > c_1\nu\beta^{m-n-t} - \beta^{m-n}$, soit

$$\beta^t qp > c_1\nu\beta^{m-n-t}p - \beta^{m-n}p > \beta^t(c - c_0) - p(\beta^{m-n} + c_1\beta^{m-n-t}).$$

Or $c_0 < \beta^{m-t}$ et $c_1 < \beta^t$, d'où $\beta^t qp > \beta^t c - \beta^t \beta^{m-t} - \beta^t(2\beta^{m-n-t}p)$ qui donne la borne supérieure suivante :

$$r = c - qp < \beta^{m-t} + 2\beta^{m-n-t}p \quad (4.4)$$

Enfin, $p < \beta^n \implies \beta^{m-n-t}p < \beta^{m-t}$ donc si $r \geq \beta^{m-t}$, alors $r - \beta^{m-n-t}p \geq 0$. Suivant l'équation (4.4) deux soustractions sont au plus nécessaires à l'étape 4 pour garantir $r < \beta^{m-t}$.

Par définition, $\beta^{n-1} < p < \beta^n$ d'où $\beta^t < \beta^{n+t}/p < \beta^{t+1}$ et donc $\beta^t \leq \nu < \beta^{t+1}$. Le calcul à la ligne 2 est une multiplication entre c_1 de taille t et ν de taille $t+1$ de complexité asymptotique $M(t)$. Cependant, dans les cas où c_1 a seulement $s < t$ mots de poids forts (soit $c_1 = c_2\beta^u$ avec $u > 0$) ce calcul sera composé d'une multiplication de complexité $M(t, s)$ suivie d'un décalage de $m+u-n-2t$ vers la gauche. Le calcul qp correspond à la multiplication de p de taille n par les t mots de poids forts de q avec une complexité de $M(n, t)$. \square

Remarque 19. Clairement, les méthodes de réduction de Barrett et de Montgomery sont symétriques. La première permet une réduction par les poids forts, la seconde par les poids faibles. Pour éliminer un même nombre de mots t , les algorithmes ont exactement la même complexité.

Nous avons maintenant les outils nécessaires à la réduction de produits de type $a_i b_j \beta^e$ où $-n \leq e \leq n$ et les complexités de ces réductions. Dans la suite, nous nous référons à ces deux algorithmes pour la réduction de ces produits en fonction de l'exposant e .

4.2 Définition de la multiplication multipartite

En divisant un des opérandes en deux, la méthode bipartite sépare le calcul de $ab\beta^{-n/2}$ en deux produits indépendants de complexité plus faible que la complexité d'une multiplication de Montgomery classique. Découper le second opérande en deux divise le calcul en quatre produits de complexité encore inférieure. Que se passe-t-il si les opérandes sont découpées en k parties de taille égale ?

Dans cette section, nous rappelons le principe de la méthode bipartite, puis nous présentons la multiplication quadripartite où les deux opérandes sont divisés en 2 parties de taille égale. Enfin, nous définissons la multiplication *multipartite*, où les opérandes sont divisés en k morceaux de même longueur, soit k^2 produits indépendants à calculer.

4.2.1 La multiplication bipartite

La méthode de multiplication bipartite présentée au paragraphe 2.3.2.6 s'exprime facilement à l'aide des réductions partielles de Barrett et de Montgomery. Le calcul $ab\beta^{-n/2} \bmod p$ se divise en deux produits modulaires : $ab_1 \bmod p$ et $ab_0\beta^{-n/2} \bmod p$ avec $b = b_1\beta^{n/2} + b_0$ et $b_0 < \beta^{n/2}$. Ces produits correspondent au calcul de deux multiplications entières ab_1 et ab_0 de complexité $M(n, n/2)$ suivis de deux réductions qui visent à supprimer respectivement les $n/2$ mots de poids fort et les $n/2$ mots de poids faible du produit total ab de taille $2n$. La suppression des mots de poids fort est réalisée par la réduction partielle de Barrett (algorithme 13) en $M(n/2) + M(n, n/2)$ selon le théorème 9. Pour les mots de poids faible, la réduction du produit $ab_0\beta^{-n/2}$ est accomplie en $M(n/2) + M(n, n/2)$ en utilisant la réduction partielle de Montgomery (algorithme 12). Au final, les calculs $ab_1 \bmod p$ et $ab_0 \bmod p$ sont donc effectués avec une complexité égale à $2M(n, n/2) + M(n/2)$ chacun, qui représente la complexité parallèle de cette méthode. La complexité séquentielle est elle égale à $4M(n, n/2) + 2M(n/2)$.

4.2.2 La multiplication quadripartite

Une extension naturelle de la multiplication bipartite est de couper les deux opérandes en deux au lieu d'en couper un seul. Cette multiplication, que nous appellerons multiplication quadripartite, consiste alors à calculer :

$$\begin{aligned} ab\beta^{-n/2} \bmod p &= (a_1\beta^{n/2} + a_0)(b_1\beta^{n/2} + b_0)\beta^{-n/2} \bmod p \\ &= (a_1b_1\beta^{n/2} + a_1b_0 + a_0b_1 + a_0b_0\beta^{-n/2}) \bmod p \end{aligned}$$

Cette multiplication quadripartite requiert donc quatre produits indépendants de complexité $M(n/2)$ plus deux réductions modulaires :

- la réduction du produit bas $a_0b_0\beta^{-n/2} \bmod p$ est calculée avec une complexité de

$$M(n/2) + M(n, n/2)$$

par la réduction partielle de Montgomery. En effet, on supprime les $n/2$ mots de poids faible du produit.

- la réduction du produit haut $a_1b_1\beta^{n/2} \bmod p$ est calculée avec une complexité de

$$M(n/2) + M(n, n/2)$$

par la réduction partielle de Barrett. En effet, comme $a_1b_1\beta^{n/2}$ est de taille $3n/2$ et p de taille n , on doit éliminer les $n/2$ mots de poids fort du produit.

- les produits médians a_0b_1 et a_1b_0 , de complexité $M(n/2)$, n'ont pas à être réduits modulo p . En effet, comme $a < p = p_1\beta^{n/2} + p_0$, on a $a_1\beta^{n/2} \leq p_1\beta^{n/2} < p$ car p est impair. On a de plus $b_0 < \beta^{n/2}$ donc $a_1b_0 < a_1\beta^{n/2} < p$. Le même raisonnement conduit à $a_0b_1 < p$.

Remarque 20. Les réductions dans cette multiplication sont équivalentes aux réductions de la méthode bipartite. En particulier, les quotients partiels calculés par Barrett dans les deux méthodes sont égaux, de même que les quotients partiels calculés par Montgomery. Cette multiplication quadripartite revient à réaliser une multiplication entière parallèle puis une réduction bipartite.

Remarque 21. Les produits médians n'ont pas à être réduits, et les calculer de façon séquentielle se fait en $2M(n/2)$. La multiplication quadripartite peut donc être effectuée en parallèle avec une complexité égale à $2M(n/2) + M(n, n/2)$ ($M(n/2)$ pour la multiplication et $M(n/2) + M(n, n/2)$ pour la réduction) en utilisant non pas quatre mais trois unités de calcul. Nous verrons par la suite que, suivant la même idée, séquentialiser les produits centraux pour les effectuer sur la même unité de calcul permet de réduire la taille des architectures requises, sans augmenter la complexité totale du calcul.

4.2.3 Généralisation

Nous allons maintenant présenter une généralisation des deux multiplications précédentes dans laquelle les opérandes ne sont pas divisés en deux mais en k morceaux chacun, soit en base $\beta^{n/k}$. Sans perte de généralité, on suppose dans la suite que k divise n . Soient $a = \sum_{i=0}^{k-1} a_i\beta^{in/k}$ et $b = \sum_{j=0}^{k-1} b_j\beta^{jn/k}$, la multiplication modulaire $ab\beta^{-n/2} \bmod p$ s'écrit :

$$ab\beta^{-n/2} \bmod p = \left(\sum_{i=0}^{k-1} \sum_{j=0}^{k-1} a_i b_j \beta^{d_{i,j}} \bmod p \right) \bmod p, \quad (4.5)$$

avec $d_{i,j} = n(i+j)/k - n/2$ et $3n/2 - 2n/k \leq d_{i,j} \leq -n/2$.

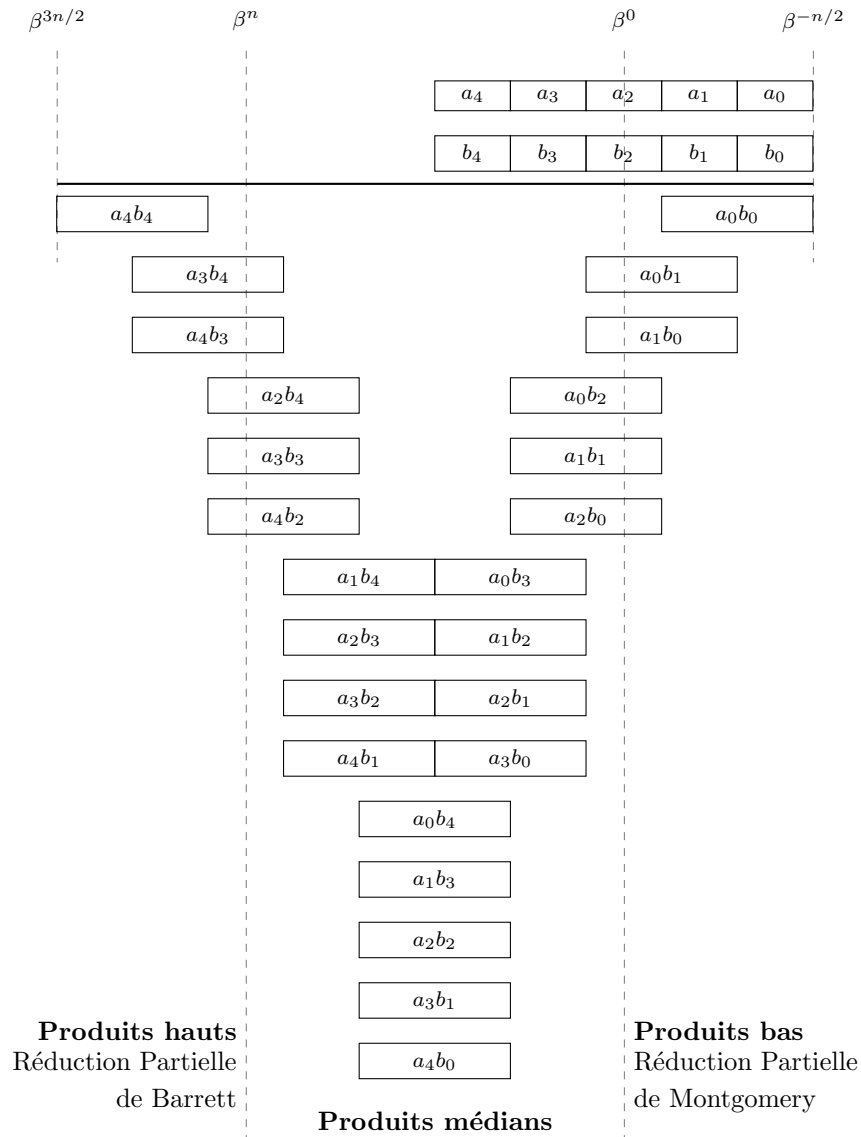
Cette multiplication modulaire s'écrit donc comme la somme de k^2 produits modulaires partiels $a_i b_j \beta^{d_{i,j}} \bmod p$ indépendants, calculables concurremment sur une architecture parallèle. Les résultats doivent ensuite être additionnés modulo p pour obtenir $ab\beta^{-n/2} \bmod p$. Ces $k^2 - 1$ additions modulaires peuvent également être parallélisées par un arbre binaire d'additions (voir chapitre 3) de hauteur $\log_2(k^2) = 2\log_2(k)$. La figure 4.3 présente la multiplication multipartite dans le cas où chacun des opérandes est divisé en $k = 5$ morceaux. Cet exemple sera détaillé dans la suite de cette section.

Similairement à la multiplication quadripartite, les produits partiels peuvent être divisés en trois groupes distincts en fonction de leur poids, c'est-à-dire de la quantité $d_{i,j}$.

Nous appellerons **produits bas** (ou produits de poids faible), les produits modulaires $a_i b_j \beta^{d_{i,j}} \bmod p$ tels que $d_{i,j} < 0$. Ces produits seront réduits avec l'algorithme de réduction partielle de Montgomery (algorithme 12). Dans la suite de ce chapitre, nous pourrons les caractériser de deux manières : $d_{i,j} < 0$ ou $0 \leq i+j \leq \lceil k/2 \rceil - 1$. En effet, comme $d_{i,j} = n(i+j)/k - n/2$, $d_{i,j} < 0 \implies i+j < k/2$. Dans l'exemple proposé à la figure 4.3, il y a 6 produits bas : $a_0b_0, a_0b_1, a_1b_0, a_0b_2, a_1b_1$ et a_2b_0 .

Symétriquement, nous appellerons **produits hauts** (ou produits de poids fort), les produits modulaires $a_i b_j \beta^{d_{i,j}} \bmod p$ tels que $d_{i,j} > n - 2n/k$. Dans ce cas, comme $a_i b_j$ est de taille $2n/k$, l'entier $a_i b_j \beta^{d_{i,j}}$ sera de taille supérieure à n et donc par définition supérieur à p . Les produits de ce type seront réduits avec l'algorithme de réduction partielle de Barrett (algorithme 13). Dans la suite de ce chapitre, nous pourrons les caractériser de deux manières : $d_{i,j} > n - 2n/k$ ou $\lceil 3k/2 \rceil - 1 \leq i+j \leq 2k - 2$. En effet, comme $i, j \leq k - 1$, $i+j \leq 2k - 2$. De plus, comme $d_{i,j} = n(i+j)/k - n/2$, $d_{i,j} > n - 2n/k \implies (i+j) > 3k/2 - 2$.

Dans l'exemple proposé à la figure 4.3, il y a 6 produits hauts : $a_4b_4, a_4b_3, a_3b_4, a_4b_2, a_3b_3$ et a_2b_4 .


 FIGURE 4.3 – Multiplication multipartite avec $k = 5$

Enfin, nous appellerons **produits médians** les produits $a_i b_j \beta^{d_{i,j}} \bmod p$ tels que $0 \leq d_{i,j} \leq n - 2n/k$. Comme $a_i b_j$ est de taille $2n/k$, l'entier $c = a_i b_j \beta^{d_{i,j}}$ sera au plus de taille n . Si le mot de poids fort p_{n-1} est tel que $p_{n-1} \geq \beta/2$, c sera inférieur à $2p$ et pourra donc être réduit par une simple soustraction. Dans la suite, nous considérerons uniquement ce cas.

Remarque 22. Dans le cas contraire, le résultat sera inférieur à $2^\alpha p$ où $\beta/2^{\alpha-1} > p_{n-1} \geq \beta/2^\alpha$.

Dans l'exemple proposé à la figure 4.3, il y a 13 produits médians. La figure illustre clairement le fait que ces produits n'ont pas à être réduits car ils sont de taille strictement inférieure à n (car k est impair).

Nous présentons dans la suite une analyse exacte de la complexité de la multiplication multipartite en considérant un $k > 1$ quelconque et un parallélisme au seul niveau modulaire. Nous illustrons cette complexité en détaillant la complexité des calculs de l'exemple précédent. Les produits médians n'ayant pas à être réduits, le terme **produit modulaire** désignera un produit haut ou un produit bas.

4.2.4 Analyse de complexité

La complexité de la multiplication multipartite est égale à la somme des complexités du calcul de ses produits.

La figure 4.3 montre clairement une symétrie parfaite dont l'axe est la ligne passant par $\beta^{n/2}$. De plus, nous avons remarqué que les complexités pour réduire t bits de poids faible et t bits de poids forts sont identiques. On peut intuitivement en conclure que la complexité de la réduction des produits hauts est égale à la complexité des produits bas. Plus précisément, pour chaque produit bas $a_i b_j \beta^{d_{i,j}}$ il existe un produit haut $a_{i'} b_{j'} \beta^{d_{i',j'}}$ de complexité égale et soumis à la relation $i' + j' = 2k - 2 - (i + j)$.

Nous présentons dans la suite des résultats valides pour les produits hauts et les produits bas en fonction d'une seule variable.

Notation 3. Dans la suite, nous appellerons rang des produits modulaires la variable l telle que $0 \leq l \leq \lceil k/2 \rceil - 1$ et définie par :

$$l = \begin{cases} i + j & \text{si } d_{i,j} < 0 \text{ (produits bas)} \\ 2k - 2 - (i + j) & \text{si } d_{i,j} > n - 2n/k \text{ (produits haut)} \end{cases}$$

Ainsi, dans l'exemple $k = 5$, les produits extrémaux $a_0 b_0$ et $a_4 b_4$ sont de rang 0, les produits $a_1 b_0$, $a_0 b_1$, $a_4 b_3$ et $a_3 b_4$ de rang 1, etc.

Remarque 23. À chaque poids $d_{i,j}$ correspond un rang l unique. À chaque rang l correspondent deux poids $d_{i,j}$ et $d_{i',j'}$ tels que $|d_{i,j}| + |d_{i',j'}| = 2n$.

Lemme 1. Si les opérandes de la multiplication multipartite sont divisés en $k > 1$ parties, il y a $l + 1$ produits modulaires à calculer pour chaque poids $d_{i,j}$.

Démonstration. Pour les produits bas, il y a $i + j + 1$ sommes telles que $l = i + j$. Comme

$$d_{i,j} = nl/k - n/2,$$

il y a bien $l + 1$ produits de même poids $d_{i,j}$. Pour les produits hauts, comme $0 \leq i, j \leq k - 1$ et $\lceil 3k/2 \rceil - 1 \leq i + j \leq 2k - 2$, pour chaque somme $i + j$ il y a $2k - 2 - (i + j) + 1$ produits hauts, soit $l + 1$. \square

Lemme 2. Si les opérandes de la multiplication multipartite sont divisés en $k > 1$ parties, le nombre de produits modulaires est égal à $\lceil k/2 \rceil (\lceil k/2 \rceil + 1)$ soit :

$$\begin{cases} \frac{k(k+2)}{4} & \text{si } k \text{ est pair} \\ \frac{(k+1)(k+3)}{4} & \text{si } k \text{ est impair} \end{cases}$$

Démonstration. Selon le lemme 1, il y a $l + 1$ produits modulaires pour chaque poids $d_{i,j}$, soit $2l + 2$ produits pour chaque rang l . Comme $0 \leq l \leq \lceil k/2 \rceil - 1$, le nombre de produits modulaires à calculer est donc égal à :

$$2 \sum_{l=0}^{\lceil k/2 \rceil - 1} (l + 1) = \left\lceil \frac{k}{2} \right\rceil \left(\left\lceil \frac{k}{2} \right\rceil + 1 \right).$$

\square

Corollaire 1. Le nombre de produits médians est égal à :

$$\begin{cases} \frac{k(3k-2)}{4} & \text{si } k \text{ est pair} \\ \frac{3k^2 - 4k - 3}{4} & \text{si } k \text{ est impair} \end{cases}$$

Démonstration. Si les deux opérandes sont découpés en k morceaux, il y a k^2 produits à calculer. Le nombre de produits médians est donc k^2 moins le nombre de produits modulaires donnés par le lemme 2. \square

Théorème 10. *La complexité du calcul de la réduction d'un produit modulaire est égale à :*

$$\begin{cases} M(n/2 - nl/k) + M(n, n/2 - nl/k) & \text{si } l \geq \lceil k/2 \rceil - 2 \\ M(n/2 - nl/k, 2n/k) + M(n, n/2 - nl/k) & \text{sinon} \end{cases}$$

Démonstration. Les produits bas $a_i b_j \beta^{d_{i,j}}$ sont réduits par l'algorithme 12 avec

$$t = |d_{i,j}| = |nl/k - n/2| = n/2 - nl/k.$$

Les produits hauts $a_i b_j \beta^{d_{i,j}}$ sont de taille $m = 2n/k + d_{i,j} = 2n/k + n(i+j)/k - n/2$. Comme p est de taille n , l'algorithme 13 les réduit modulo p avec $t = m - n = n(i+j+2)/k - 3n/2$. Par changement de variable, on a $n(i+j+2)/k - 3n/2 = n/2 - nl/k$.

Selon les théorèmes 8 et 9, la réduction de ces produits se fait donc en

$$M(n/2 - nl/k) + M(n, n/2 - nl/k)$$

si $n/2 - nl/k \leq 2n/k$ et en

$$M(n/2 - nl/k, 2n/k) + M(n, n/2 - nl/k)$$

dans le cas contraire. En effet, dans ce dernier cas, il y a seulement $2n/k$ mots non nuls parmi les $n/2 - nl/k$ mots à réduire, comme illustré à la figure 4.3 par les produits $a_0 b_0$ et $a_4 b_4$. \square

Théorème 11. *Si les deux opérandes sont divisés en k parties, la complexité du calcul des réductions des produits modulaires est égale à :*

$$\begin{aligned} & 2 \sum_{l=\lceil \frac{k}{2} \rceil - 2}^{\lceil \frac{k}{2} \rceil - 1} (l+1) \left[M\left(\frac{n}{2} - \frac{nl}{k}\right) + M\left(n, \frac{n}{2} - \frac{nl}{k}\right) \right] \\ & + 2 \sum_{l=0}^{\lceil \frac{k}{2} \rceil - 3} (l+1) \left[M\left(\frac{n}{2} - \frac{nl}{k}, \frac{2n}{k}\right) + M\left(n, \frac{n}{2} - \frac{nl}{k}\right) \right] \end{aligned}$$

Démonstration. Selon le lemme 2, il y a $\lceil \frac{k}{2} \rceil (\lceil \frac{k}{2} \rceil + 1)$ produits $a_i b_j$ de complexité $M(n/k)$ à calculer. De plus, selon le lemme 10, il y a $(l+1)$ produits à réduire pour chaque l différent. En appliquant le théorème 10 à chaque valeur de l , on prouve le théorème. \square

Remarque 24. Les complexités du calcul des parties hautes et des parties basses dans la multiplication multipartite sont donc égales. Cette égalité et la symétrie des calculs vient du coefficient $\alpha = 1/2$ choisi pour la représentation de Montgomery dans la multiplication bipartite : $ab\beta^{-\alpha n} \bmod p$. La partie Montgomery du calcul deviendra prépondérante pour $\alpha \in]0.5, 1]$ et la partie Barrett du calcul pour $\alpha \in [0, 0.5[$. Les extrémités 0 et 1 de l'intervalle correspondent respectivement à la multiplication de Montgomery et la multiplication de Barrett classiques.

Nous pouvons exprimer maintenant la complexité totale de la multiplication multipartite.

Théorème 12. *Si les opérandes de la multiplication modulaire multipartite sont divisés en $k > 1$ parties, la complexité totale de son calcul est égale à :*

$$\begin{aligned} & k^2 M\left(\frac{n}{k}\right) \\ & + 2 \sum_{l=\lceil \frac{k}{2} \rceil - 2}^{\lceil \frac{k}{2} \rceil - 1} (l+1) \left[M\left(\frac{n}{2} - \frac{nl}{k}\right) + M\left(n, \frac{n}{2} - \frac{nl}{k}\right) \right] \\ & + 2 \sum_{l=0}^{\lceil \frac{k}{2} \rceil - 3} (l+1) \left[M\left(\frac{n}{2} - \frac{nl}{k}, \frac{2n}{k}\right) + M\left(n, \frac{n}{2} - \frac{nl}{k}\right) \right] \end{aligned} \tag{4.6}$$

Démonstration. La multiplication multipartite consiste à calculer k^2 produits $a_i b_j$ de complexité $M(n/k)$ puis à réduire un certain nombre de ces produits. La complexité de ces réductions est donnée par le théorème 11. \square

La complexité séquentielle de la multiplication multipartite est supérieure à la complexité des multiplications de Montgomery, de Barrett et bipartite. En effet, notre méthode ajoute de la redondance pour le calcul des quotients partiels des produits de même poids $d_{i,j}$, et donc pour la multiplication de ces quotients par p . Cependant, la multiplication multipartite est une méthode visant un calcul parallèle sur une architecture disposant de plusieurs unités de traitements. Dans la suite de ce chapitre, nous détaillons les différents ordonnancements possibles des produits modulaires sur plusieurs tâches en visant une complexité optimale.

4.3 Complexité parallèle de la multiplication multipartite

La complexité parallèle de la multiplication multipartite correspond à la complexité de son produit le plus coûteux, et plus précisément du calcul modulaire ayant la plus grande complexité, c'est-à-dire le plus grand nombre de mots à éliminer.

Théorème 13. *Dans un modèle où le parallélisme est placé au seul niveau de l'arithmétique modulaire, si les opérandes de la multiplication multipartite sont divisés en $k > 1$ parties, la complexité parallèle de la multiplication multipartite est égale à :*

$$\begin{cases} 2M(n/2) + M(n, n/2) & \text{si } k = 2 \\ M(n/3) + M(n/2) + M(n, n/2) & \text{si } k = 3 \\ M(n/k) + M(n/2, 2n/k) + M(n, n/2) & \text{si } k > 3 \end{cases}$$

Démonstration. Le théorème 10 donne la complexité de la réduction d'un produit modulaire. Il est clair qu'elle est maximale pour $l = 0$ car $n/2 > 0$ et comme $0 \leq l$, $nl/k > 0$ pour $k > 1$. De plus, les cas $k = 2$ et $k = 3$ correspondent aux cas particuliers où $l > \lceil k/2 \rceil - 2$. \square

Lemme 3. Si $k_1 \times k_2$ processeurs sont disponibles, la complexité parallèle de la multiplication modulaire multipartite est :

$$\begin{cases} M\left(\frac{n}{k_1}, \frac{n}{k_2}\right) + M\left(\frac{n}{2}, \frac{n}{k_1} + \frac{n}{k_2}\right) + M\left(n, \frac{n}{2}\right) & \text{si } n/k_1 + n/k_2 < n/2 \\ M\left(\frac{n}{k_1}, \frac{n}{k_2}\right) + M\left(\frac{n}{2}\right) + M\left(n, \frac{n}{2}\right) & \text{sinon.} \end{cases} \quad (4.7)$$

Démonstration. Si les opérandes a et b sont divisés respectivement en k_1 et k_2 parties, la multiplication $a_i b_j$ a pour complexité $M(n/k_1, n/k_2)$ avec un résultat de taille $n/k_1 + n/k_2$. Une fois encore, les complexités maximales sont atteintes pour les deux produits extrémaux, soit $d_{i,j} = -n/2$ (produit bas) et $d_{i,j} = 3n/2 - n/k_1 - n/k_2$ (produit haut). En appliquant les théorèmes 8 et 9 sur ces deux produits, on prouve le lemme. \square

Dans la suite de ce chapitre, nous allons étudier l'ordonnement des calculs sur une architecture parallèle avec deux objectifs (parfois incompatibles comme nous le verrons) : utiliser le moins de processeurs possibles tout en garantissant la complexité ci-dessus et réduire cette complexité parallèle sans augmenter le nombre de processeurs nécessaires.

4.4 Ordonnement des calculs pour l'approche modulaire

Dans cette section, nous considérons un parallélisme au seul niveau de l'arithmétique modulaire. Nous détaillerons un double niveau de parallélisme dans la section 4.5.

À ce stade, nous pouvons poser deux faits :

(1) la complexité donnée au théorème 13 est garantie en utilisant k^2 processeurs ;

(2) la complexité donnée au théorème 13 diminue quand k augmente.

Dans un modèle théorique, la complexité limite de la multiplication multipartite est égale à

$$\mathcal{O}(1) + \mathcal{O}(n/2) + M(n, n/2).$$

Dans un modèle réaliste, le nombre de processeurs est malheureusement limité. Nous allons donc détailler une approche permettant de diminuer le nombre de processeurs nécessaires au calcul de la multiplication multipartite en démontrant que séquentialiser certains produits partiels n'augmente pas la complexité parallèle du calcul.

Remarque 25. Pour une complexité identique, le nombre de processeurs nécessaires peut être réduit en utilisant les schémas de multiplication sous-quadratique de Karatsuba ou de Toom-Cook (en augmentant toutefois les interactions entre les tâches parallèles). Dans [82], les auteurs utilisent la multiplication de Karatsuba pour améliorer l'implémentation matérielle de la méthode bipartite.

4.4.1 Regroupement des produits médians

La première étape de notre approche consiste à séquentialiser le calcul de plusieurs produits médians sur un même processeur.

Théorème 14. *La complexité donnée au théorème 13 est garantie en utilisant un nombre de processeurs égal à :*

$$\begin{cases} \left\lceil \frac{k(3k+2)}{6} \right\rceil & \text{si } k \text{ est pair} \\ \left\lceil \frac{3k^2+4k+3}{6} \right\rceil & \text{si } k \text{ est impair} \end{cases}$$

Démonstration. La complexité des produits médians générés par un découpage en k est égale à $M(n/k)$. Or, d'après le théorème 10 et sachant que $0 \leq l$, la complexité maximale d'un produit modulaire est égale à $M(n/k) + M(n/2, \min(n/2, 2n/k)) + M(n, n/2)$. Sans présumer des rapports exacts $M(n/2, \min(n/2, 2n/k))/M(n/k)$ et $M(n, n/2)/M(n/k)$, il est clair que pour $k > 1$, on a $M(n/2, \min(n/2, 2n/k)) \geq M(n/k)$ et $M(n, n/2) \geq M(n/k)$. Cela signifie qu'au minimum trois produits médians peuvent être calculés séquentiellement dans le même temps que le calcul d'un produit modulaire. On ajoute donc le nombre de produits modulaires (lemme 2) au nombre de produits médians (lemme 1) divisé par trois pour obtenir le nombre de processeurs nécessaires au calcul. \square

Remarque 26. Nous verrons par la suite qu'il est également possible de répartir certains de ces produits médians sur les processeurs qui calculent déjà un produit modulaire.

Exemple 9 (Ordonnancement des calculs de la multiplication multipartite avec $k = 5$ (figure 4.3)). Selon le théorème précédent, pour garantir une complexité parallèle égale à $M(n/5) + M(n/2, 2n/5) + M(n, n/2)$, il suffit de 17 processeurs. L'ordonnancement des calculs est donné par le tableau 4.1.

4.4.2 Diminuer le nombre de réductions modulaires

Chacun des produits modulaires calcule un quotient partiel qui permet l'élimination d'un certain nombre de mots de poids fort ou de poids faible du produit partiel. Cela signifie en particulier que pour un même poids $d_{i,j}$, plusieurs quotients partiels et multiplication qp vont être calculés pour réduire la même partie du calcul.

En additionnant les produits de même poids avant leur réduction, on diminue le nombre de quotients à calculer, qui passe de $l + 1$ à 1. De même, il n'y a plus $l + 1$ mais une seule multiplication qp à réaliser, comme illustré à la figure 4.4.

Nous avons vu que la caractérisation des produits bas est $0 \leq i + j \leq \lceil k/2 \rceil - 1$. Or $d_{i,j} = d_{i',j'} \implies i + j = i' + j'$. Si l'on regroupe les réductions de même poids, le nombre d'appel à l'algorithme de réduction partielle de Montgomery est donc $\lceil k/2 \rceil$. De même, les produits hauts sont caractérisés par $\lfloor 3k/2 \rfloor - 1 \leq i + j \leq 2k - 2$ soit $\lceil k/2 \rceil$ appels à l'algorithme de réduction partielle de Barrett.

P.	Calcul	Complexité
0	$a_0 b_0 \beta^{-n/2} \bmod p$	$M(n/5) + M(n/2, 2n/5) + M(n, n/2)$
1	$a_1 b_0 \beta^{-3n/10} \bmod p$	$M(n/5) + M(3n/10) + M(n, 3n/10)$
2	$a_0 b_1 \beta^{-3n/10} \bmod p$	$M(n/5) + M(3n/10) + M(n, 3n/10)$
3	$a_0 b_2 \beta^{-n/10} \bmod p$	$M(n/5) + M(n/10) + M(n, n/10)$
4	$a_2 b_0 \beta^{-n/10} \bmod p$	$M(n/5) + M(n/10) + M(n, n/10)$
5	$a_1 b_1 \beta^{-n/10} \bmod p$	$M(n/5) + M(n/10) + M(n, n/10)$
6	$(a_0 b_3 + a_0 b_4 \beta^{n/5} + a_1 b_4 \beta^{2n/5}) \beta^{n/10} \bmod p$	$3M(n/5)$
7	$(a_3 b_0 + a_4 b_0 \beta^{n/5} + a_4 b_1 \beta^{2n/5}) \beta^{n/10} \bmod p$	$3M(n/5)$
8	$(a_1 b_2 + a_1 b_3 \beta^{n/5} + a_2 b_3 \beta^{2n/5}) \beta^{n/10} \bmod p$	$3M(n/5)$
9	$(a_2 b_1 + a_3 b_1 \beta^{n/5} + a_3 b_2 \beta^{2n/5}) \beta^{n/10} \bmod p$	$3M(n/5)$
10	$a_2 b_2 \beta^{3n/10} \bmod p$	$M(n/5)$
11	$a_3 b_3 \beta^{7n/10} \bmod p$	$M(n/5) + M(n/10) + M(n, n/10)$
12	$a_4 b_2 \beta^{7n/10} \bmod p$	$M(n/5) + M(n/10) + M(n, n/10)$
13	$a_2 b_4 \beta^{7n/10} \bmod p$	$M(n/5) + M(n/10) + M(n, n/10)$
14	$a_4 b_3 \beta^{9n/10} \bmod p$	$M(n/5) + M(3n/10) + M(n, 3n/10)$
15	$a_3 b_4 \beta^{9n/10} \bmod p$	$M(n/5) + M(3n/10) + M(n, 3n/10)$
16	$a_4 b_4 \beta^{11n/10} \bmod p$	$M(n/5) + M(n/2, 2n/k) + M(n, n/2)$

TABLE 4.1 – Exemple de groupement de produits médians sur un même processeur pour la multiplication multipartite $k = 5$

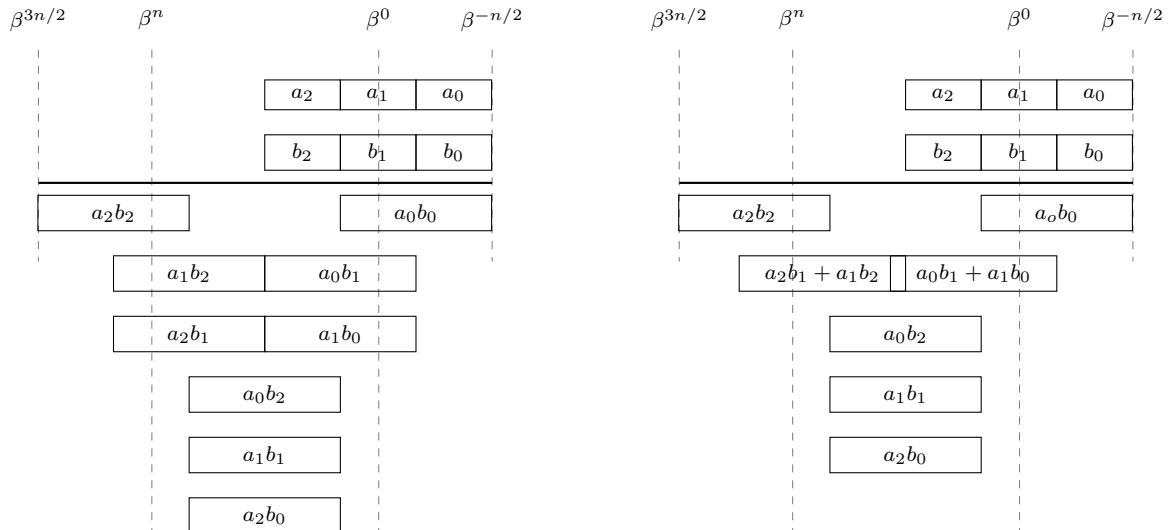


FIGURE 4.4 – Diminution du nombre de réductions par regroupement des calculs de même poids pour $k = 3$

Cette méthode permet de réduire la complexité totale de la multiplication multipartite à :

$$k^2 M\left(\frac{n}{k}\right) + 2 \sum_{l=\lceil \frac{k}{2} \rceil - 2}^{\lceil \frac{k}{2} \rceil - 1} \left[M\left(\frac{n}{2} - \frac{nl}{k}\right) + M\left(n, \frac{n}{2} - \frac{nl}{k}\right) \right] \\ + 2 \sum_{l=0}^{\lceil \frac{k}{2} \rceil - 3} \left[M\left(\frac{n}{2} - \frac{nl}{k}, \frac{2n}{k}\right) + M\left(n, \frac{n}{2} - \frac{nl}{k}\right) \right].$$

Si cette méthode réduit la complexité séquentielle de la méthode, elle introduit cependant une dépendance dans les calculs, car la réduction devra attendre la fin du calcul de tous les produits de même poids. Cependant, le théorème suivant montre que calculer tous les produits puis leur réduction sur un même processeur n'augmente pas la complexité parallèle de la multiplication multipartite.

Théorème 15. *Le nombre de processeurs nécessaire pour garantir la complexité donnée au théorème 13 est égal à :*

$$\begin{cases} \left\lceil \frac{k(3k+10)}{12} \right\rceil & \text{si } k \text{ est pair} \\ \left\lceil \frac{3k^2+8k+6}{12} \right\rceil & \text{si } k \text{ est impair.} \end{cases}$$

Démonstration. Considérons le cas où une seule réduction est réalisée pour chaque poids. Si l'on séquentialise l'ensemble des produits de même poids qui devront être réduits, on doit utiliser $2\lceil k/2 \rceil$ processeurs pour le calcul des produits modulaires. On soustrait donc au nombre de processeurs du théorème 14 le nombre de processeurs devenus inutiles pour le calcul des produits modulaires. Prouvons maintenant que calculer séquentiellement tous les produits de même poids puis leur réduction n'augmente pas la complexité parallèle de la multiplication multipartite.

Soit r_l la complexité de la réduction d'un produit modulaire de rang l . Par le lemme 10 on a : $r_l \geq r_{l+1}$. De plus, par le lemme 1 il y a exactement un produit de plus entre deux rangs l et $l+1$. Comme la complexité parallèle est égale à la complexité du calcul des termes extrémaux ($l=0$), on veut prouver :

$$M(n/k) + r_0 \geq 2M(n/k) + r_1 \geq 3M(n/k) + r_2 \geq \dots \geq \lceil k/2 \rceil M(n/k) + r_{\lceil k/2 \rceil - 1}$$

ce qui revient à prouver

$$r_l \geq r_{l+1} + M(n/k) \tag{4.8}$$

Il y a trois cas à considérer :

- $r_l = M(n/2 - nl/k, 2n/k) + M(n, n/2 - nl/k)$ et $r_{l+1} = M(n/2 - n(l+1)/k, 2n/k) + M(n, n/2 - n(l+1)/k)$;
- $r_l = M(n/2 - nl/k, 2n/k) + M(n, n/2 - nl/k)$ et $r_{l+1} = M(n/2 - n(l+1)/k) + M(n, n/2 - n(l+1)/k)$;
- $r_l = M(n/2 - nl/k) + M(n, n/2 - nl/k)$ et $r_{l+1} = M(n/2 - n(l+1)/k) + M(n, n/2 - n(l+1)/k)$.

Dans chacun de ces cas, l'équation 4.8 est vraie, quel que soit l'algorithme de multiplication entière utilisé. \square

Exemple 10 (Ordonnancement des calculs de la multiplication multipartite avec $k=5$ (figure 4.3)). Selon le théorème précédent, pour garantir une complexité parallèle égale à $M(n/5) + M(n/2, 2n/5) + M(n, n/2)$, il suffit de 11 processeurs. L'ordonnancement des calculs est donné par le tableau suivant.

P.	Calcul	Complexité
0	$a_0 b_0 \beta^{-n/2} \bmod p$	$M(n/5) + M(n/2, 2n/5) + M(n, n/2)$
1	$(a_1 b_0 + a_0 b_1) \beta^{-3n/10} \bmod p$	$2M(n/5) + M(3n/10) + M(n, 3n/10)$
2	$(a_0 b_2 + a_2 b_0 + a_1 b_1) \beta^{-n/10} \bmod p$	$3M(n/5) + M(n/10) + M(n, n/10)$
3	$(a_0 b_3 + a_0 b_4 \beta^{n/5} + a_1 b_4 \beta^{2n/5}) \beta^{n/10} \bmod p$	$3M(n/5)$
4	$(a_3 b_0 + a_4 b_0 \beta^{n/5} + a_4 b_1 \beta^{2n/5}) \beta^{n/10} \bmod p$	$3M(n/5)$
5	$(a_1 b_2 + a_1 b_3 \beta^{n/5} + a_2 b_3 \beta^{2n/5}) \beta^{n/10} \bmod p$	$3M(n/5)$
6	$(a_2 b_1 + a_3 b_1 \beta^{n/5} + a_3 b_2 \beta^{2n/5}) \beta^{n/10} \bmod p$	$3M(n/5)$
7	$a_2 b_2 \beta^{3n/10} \bmod p$	$M(n/5)$
8	$(a_2 b_4 + a_4 b_2 + a_3 b_3) \beta^{7n/10} \bmod p$	$3M(n/5) + M(n/10) + M(n, n/10)$
9	$(a_3 b_4 + a_4 b_3) \beta^{9n/10} \bmod p$	$2M(n/5) + M(3n/10) + M(n, 3n/10)$
10	$a_4 b_4 \beta^{11n/10} \bmod p$	$M(n/5) + M(n/2, 2n/k) + M(n, n/2)$

4.4.3 Méthodes heuristiques pour l'ordonnement des calculs

Le nombre de processeurs donné par le théorème 15 garantit la complexité 13, mais il n'est cependant pas optimal. En effet, nous avons vu qu'au moins trois produits médians peuvent être calculés dans le même temps que le produits modulaire le plus coûteux. De même, les produits modulaires ont des coûts différents : peut-on calculer un ou plusieurs produits médians à la suite d'un produit modulaire sans augmenter la complexité parallèle du calcul ? Il est difficile de répondre à ces deux questions car elles dépendent de la complexité de la multiplication entière ($M(n, m)$) qui peut être différente suivant l'algorithme utilisé.

Nous avons choisi de considérer deux cas : le modèle purement quadratique, dans lequel $M(n, m) \approx 2nm$ opérations de mots et un modèle théorique où la multiplication serait *linéaire* avec $M(n, m) \approx n + m$. On suppose $k > 3$, ce qui entraîne une complexité parallèle égale à $M(n/k) + M(n/2, 2n/k) + M(n, n/2)$.

4.4.3.1 Ordonnement dans le modèle quadratique

On pose $M(n, m) = 2nm$ opérations. La complexité d'un produit modulaire de rang l est égale à

$$(l + 1)M(n/k) + M(n/2 - ln/k, 2n/k) + M(n, n/2 - ln/k),$$

c'est-à-dire un nombre d'opérations égal à :

$$2(l + 1) \frac{n^2}{k^2} + 2n^2 \left[\frac{k - 2l}{k^2} \right] + n^2 \left[\frac{k - 2l}{k} \right] = \frac{n^2}{k^2} [k^2 + 2k(1 - l) + 2]. \quad (4.9)$$

La complexité du produit modulaire de rang suivant $l + 1$ est égale à $(l + 2)M(n/k) + M(n/2 - (l + 1)n/k, 2n/k) + M(n, n/2 - (l + 1)n/k)$ soit un nombre d'opérations égal à :

$$4(l + 1) \frac{n^2}{k^2} + 2n^2 \left[\frac{k - 2(l + 1)}{k^2} \right] + n^2 \left[\frac{k - 2(l + 1)}{k} \right] = \frac{n^2}{k^2} [k^2 + 2k(1 - (l + 1)) + 2]. \quad (4.10)$$

La différence entre les équation 4.9 et 4.10 est égale à :

$$2k \left[\frac{n^2}{k^2} \right] = kM(n/k). \quad (4.11)$$

Selon cette équation, on peut donc calculer un produit modulaire de rang $l + 1$ plus k produits médians de complexité $M(n/k)$ dans le même temps que le calcul d'un produit modulaire de rang l .

Or, k (ou $k + 1$ processeurs si k est impair) suffisent pour que chaque produit modulaire soit calculé sur une unité différente. On peut donc répartir l'ensemble des produits médians restants à calculer sur $k - 2$ processeurs (on ne souhaite pas augmenter la complexité parallèle de la multiplication, donc on n'ajoute pas de calcul aux unités qui calculeront les produits extrémaux). Comme on vient de le voir,

on peut calculer *au minimum* k produits médians sans augmenter la complexité sur chacun des $k - 2$ processeurs, soit au total $k(k - 2)$ produits médians, ce qui est très supérieur au nombre de produits médians restant donné par le corollaire 1. On démontre ainsi le théorème suivant.

Théorème 16. *La complexité parallèle donnée par le théorème 13 est garantie dans le modèle quadratique avec un nombre de processeurs égal à :*

$$\begin{cases} k & \text{si } k \text{ est pair} \\ k + 1 & \text{sinon} \end{cases}$$

Exemple 11 (Ordonnancement des calculs de la multiplication multipartite avec $k = 5$ (figure 4.3)). Selon le théorème précédent, pour garantir une complexité parallèle égale à $M(n/5) + M(n/2, 2n/5) + M(n, n/2)$, il suffit de 6 processeurs. L'ordonnancement des calculs est donné par le tableau suivant. Les résultats de chaque calcul sont réduits modulo p .

P.	Calcul	Complexité
0	$a_0 b_0 \beta^{-n/2}$	$M(\frac{n}{5}) + M(\frac{n}{2}, \frac{2n}{5}) + M(n, \frac{n}{2})$
1	$(a_0 b_3 + a_0 b_4 \beta^{n/5} + a_1 b_4 \beta^{2n/5}) \beta^{n/10} + (a_1 b_0 + a_0 b_1) \beta^{-3n/10}$	$5M(\frac{n}{5}) + M(\frac{3n}{10}) + M(n, \frac{3n}{10})$
2	$(a_3 b_0 + a_4 b_0 \beta^{n/5} + a_4 b_1 \beta^{2n/5}) \beta^{n/10} + (a_0 b_2 + a_2 b_0 + a_1 b_1) \beta^{-n/10}$	$3M(\frac{n}{5}) + M(\frac{n}{10}) + M(n, \frac{n}{10})$
3	$(a_1 b_2 + a_1 b_3 \beta^{n/5} + a_2 b_3 \beta^{2n/5}) \beta^{n/10} + (a_2 b_4 + a_4 b_2 + a_3 b_3) \beta^{7n/10}$	$6M(\frac{n}{5}) + M(\frac{n}{10}) + M(n, \frac{n}{10})$
4	$(a_2 b_1 + (a_2 b_2 + a_3 b_1) \beta^{n/5} + a_3 b_2 \beta^{2n/5}) \beta^{n/10} + (a_3 b_4 + a_4 b_3) \beta^{9n/10}$	$6M(\frac{n}{5}) + M(\frac{3n}{10}) + M(n, \frac{3n}{10})$
5	$a_4 b_4 \beta^{11n/10}$	$M(\frac{n}{5}) + M(\frac{n}{2}, \frac{2n}{5}) + M(n, \frac{n}{2})$

4.4.3.2 Ordonnancement dans le modèle linéaire

On pose $M(n, m) = n + m$. Le nombre d'opérations du produit de rang l de complexité

$$(l + 1)M(n/k) + M(n/2 - nl/k, 2n/k) + M(n, n/2 - nl/k)$$

est égal à :

$$2(l + 1)n/k + n/2 - nl/k + 2n/k + n + n/2 - nl/k = 2n + 4n/k. \quad (4.12)$$

Ce nombre d'opérations ne dépend pas de l , il est constant à n et k fixés. Cela signifie deux choses : on a une multiplication multipartite lisse, c'est-à-dire que toutes les tâches ont la même complexité ; et on ne peut pas, sans augmenter la complexité parallèle de la multiplication, calculer des produits médians à la suite des produits modulaires. On en déduit les deux théorèmes suivants.

Théorème 17. *La complexité parallèle donnée par le théorème 13 est garantie dans le modèle linéaire avec un nombre de processeurs égal à :*

$$\begin{cases} k + \left\lceil \frac{k(3k - 2)}{4(k + 2)} \right\rceil & \text{si } k \text{ est pair} \\ k + 1 + \left\lceil \frac{3k^2 - 4k - 3}{4(k + 2)} \right\rceil & \text{sinon} \end{cases}$$

Démonstration. Dans ce modèle, le nombre d'opérations est $2n + 4n/k$ pour le calcul des produits modulaires et $2n/k$ pour le calcul d'un produit médian. On peut donc calculer $k + 2$ produits médians séquentiellement sur un processeur sans augmenter la complexité parallèle. Au final, il faut donc k (ou $k + 1$) processeurs pour les produits modulaires et le nombre de produits médians (corollaire 1) divisé par $k + 2$ processeurs pour les produits médians. \square

Théorème 18. *En utilisant k processeurs si k est pair ou $k + 1$ processeurs si k est impair, la complexité de la multiplication multipartite dans le modèle linéaire pour $k > 3$ est égale à :*

$$\begin{cases} \left\lceil \frac{3k + 2}{4} \right\rceil M(n/k) + M(n/2, 2n/k) + M(n, n/2) & \text{si } k \text{ est pair} \\ \left\lceil \frac{3k}{4} \right\rceil M(n/k) + M(n/2, 2n/k) + M(n, n/2) & \text{si } k \text{ est impair} \end{cases}$$

Théorème	#P	k	Complexité
15	4	2	$M(n/2) + M(n/2) + M(n, n/2)$
	6	3	$M(n/3) + M(n/2) + M(n, n/2)$
	8	4	$M(n/4) + M(n/2) + M(n, n/2)$
	12	5	$M(n/5) + M(n/2, 2n/5) + M(n, n/2)$
16	4	4	$M(n/4) + M(n/2) + M(n, n/2)$
	6	6	$M(n/6) + M(n/2, n/3) + M(n, n/2)$
	8	8	$M(n/8) + M(n/2, n/4) + M(n, n/2)$
	12	12	$M(n/12) + M(n/2, n/6) + M(n, n/2)$
17	4	3	$M(n/3) + M(n/2) + M(n, n/2)$
	6	4	$M(n/4) + M(n/2) + M(n, n/2)$
	8	5	$M(n/5) + M(n/2, 2n/5) + M(n, n/2)$
	12	7	$M(n/7) + M(n/2, 2n/7) + M(n, n/2)$

TABLE 4.2 – Complexité parallèle de la multiplication multipartite en fonction du nombre de cœurs utilisés et du modèle choisi

Démonstration. On répartit les produits médians sur les k ou $k+1$ processeurs. La complexité obtenue est donc la complexité parallèle plus le coût des produits médians. \square

Exemple 12 (Ordonnancement des calculs de la multiplication multipartite avec $k = 5$ (figure 4.3) en utilisant le théorème 17). Selon le théorème précédent, pour garantir une complexité parallèle égale à $M(n/5) + M(n/2, 2n/5) + M(n, n/2)$, il suffit de 8 processeurs. L'ordonnancement des calculs est donné par le tableau suivant.

P. Calcul	Complexité
0 $a_0b_0\beta^{-n/2} \bmod p$	$M(n/5) + M(n/2, 2n/5) + M(n, n/2)$
1 $(a_1b_0 + a_0b_1)\beta^{-3n/10} \bmod p$	$2M(n/5) + M(3n/10) + M(n, 3n/10)$
2 $(a_0b_2 + a_2b_0 + a_1b_1)\beta^{-n/10} \bmod p$	$3M(n/5) + M(n/10) + M(n, n/10)$
3 $(a_0b_3 + a_3b_0 + (a_4b_0 + a_0b_4)\beta^{n/5} + (a_1b_4 + a_4b_1)\beta^{2n/5})\beta^{n/10}$	$6M(n/5)$
4 $(a_1b_2 + a_2b_1 + (a_1b_3 + a_3b_1 + a_2b_2)\beta^{n/5} + (a_2b_3 + a_3b_2)\beta^{2n/5})\beta^{n/10}$	$7M(n/5)$
5 $(a_2b_4 + a_4b_2 + a_3b_3)\beta^{7n/10} \bmod p$	$3M(n/5) + M(n/10) + M(n, n/10)$
6 $(a_3b_4 + a_4b_3)\beta^{9n/10} \bmod p$	$2M(n/5) + M(3n/10) + M(n, 3n/10)$
7 $a_4b_4\beta^{11n/10} \bmod p$	$M(n/5) + M(n/2, 2n/5) + M(n, n/2)$

4.4.4 Conclusion

En considérant un parallélisme au seul niveau arithmétique modulaire, on peut donner un premier résumé des complexités de la multiplication multipartite en fonction du nombre de processeurs disponibles, de k et du modèle utilisé : générique (§4.4.2 et théorème 15), quadratique (§4.4.3.1 et théorème 16) ou linéaire (§4.4.3.2 et théorème 17).

La méthode à utiliser en pratique sera déterminée par la taille des entiers à multiplier. Cependant, notre préférence va à l'approche quadratique et ce pour deux raisons : l'heuristique proposée est proche de la réalité en termes d'opérations sur les mots ; et pour les tailles d'entiers cryptographiques (plusieurs milliers de bits) les algorithmes à complexité quasi-linéaire sont généralement peu efficaces (voir le tableau 2.3 qui donne les seuils de changement de multiplication pour une architecture).

Avec un parallélisme à ce niveau, la seule comparaison possible est avec la multiplication bipartite, qui a une complexité parallèle égale à $M(n/2) + 2M(n, n/2)$ mais qui nécessite seulement deux

cœurs. Nous avons vu au chapitre 3 que cette multiplication, tout comme les versions classiques de Montgomery et de Barrett, peuvent bénéficier d'un parallélisme au niveau de l'arithmétique entière.

Il est donc évident qu'un modèle plus réaliste consiste à considérer un parallélisme à deux niveau : on découpe les opérations modulaires en un (Montgomery et Barrett), deux (bipartite), quatre (quadripartite) ou plus (multipartite) calculs indépendants, puis on utilise les opérateurs d'arithmétique entière parallèle pour accélérer les calculs entiers.

4.5 Modèle réaliste : un parallélisme à deux niveaux

Les multiplications de Barrett, de Montgomery et la méthode bipartite tirent donc avantage d'un parallélisme de l'arithmétique entière. Nous allons décrire dans cette section une méthode qui permet un double parallélisme pour la multiplication multipartite sans augmenter le nombre d'unités de calcul nécessaires. Il est possible de faire baisser la complexité parallèle de notre multiplication en utilisant un double niveau de parallélisme.

Le résultat de chacun des produits modulaires de la multiplication multipartite est obtenu par le calcul de trois multiplications entières qui doivent être exécutées séquentiellement. Nous pourrions paralléliser ces multiplications à l'aide des opérateurs parallèles définis dans le chapitre 3, c'est-à-dire utiliser plusieurs tâches pour calculer chaque produit modulaire, mais cela ajouterait une dimension au nombre de processeurs nécessaires pour le calcul.

Il est possible d'utiliser le parallélisme au niveau de l'arithmétique entière sans augmenter le nombre de processeurs tout en faisant diminuer la complexité parallèle du calcul. Cependant, ces améliorations se font au prix de synchronisation(s) qui n'apparaissent pas dans le modèle modulaire.

La complexité parallèle de la multiplication multipartite est dominée par $M(n, n/2)$, invariant quelque soit k , et correspondant au coût de la multiplication qp dans les produits modulaires extrêmes. La seule façon de baisser la complexité de cette multiplication est d'utiliser une multiplication entière parallèle. Cependant, cela nécessite soit un nombre plus important de processeurs, soit une diminution de k pour un nombre de processeurs donné. En outre, si cette multiplication est parallélisée, la complexité de la multiplication multipartite est alors dominée par $M(n, n/2 - n/k)$, correspondant aux calculs des multiplications qp des produits modulaires de rang 1.

Une méthode efficace pour diminuer le coût de ce calcul est de réunir les $2\lceil k/2 \rceil$ quotients résultants des $2\lceil k/2 \rceil$ produits modulaires et de calculer qp une seule fois. En utilisant les processeurs ainsi libérés, cette opération pourra être parallélisée. Pour ce faire, on peut remarquer que la multiplication multipartite s'écrit comme :

$$ab\beta^{-n/2} \bmod p = \left(a_{k-1}b_{k-1}\beta^{2n\frac{k-1}{k}} + \dots + a_0b_0 \right) \beta^{-n/2} - \left(\underbrace{\left[\overbrace{(q_{2k-2}\beta^{n\frac{k-2}{k}} + \dots + q_{i+j})\beta^{n/2}}^{\text{Quotients hauts}} - \overbrace{(q_{\lceil \frac{k}{2} \rceil - 1}\beta^{n\frac{k-2}{k}} + \dots + q_0)}^{\text{Quotients bas}} \right]}_{qp} \right) p \beta^{-n/2}. \quad (4.13)$$

En additionnant l'ensemble des quotients partiels, on obtient un quotient de taille n . La multiplication qp est alors une multiplication de complexité $M(n)$. Dans le modèle quadratique, en utilisant $k = k_1k_2$ processeurs, la complexité parallèle de la multiplication multipartite est égale à :

$$M\left(\frac{n}{k}\right) + M\left(\frac{n}{2}, \frac{2n}{k}\right) + M\left(\frac{n}{k_1}, \frac{n}{k_2}\right).$$

Dans ce modèle, l'ordonnancement classique défini au paragraphe 4.4.3.1 s'applique sans transformation pour garantir la complexité précédente.

L'ordonnement est plus compliqué dans le modèle linéaire. En effet, nous avons vu que le nombre d'opérations dans ce modèle pour un parallélisme modulaire est constant. Or, si l'on considère uniquement les complexités du calcul des produits suivis du calcul du quotient partiel, le nombre d'opérations dans le modèle linéaire est égal à :

$$\frac{n}{2} + \frac{n}{k}(l + 4).$$

Cela signifie que la complexité parallèle de la multiplication n'est plus donnée par les produits de rang 0, mais par les produits de rang maximal, soit $l = \lceil k/2 \rceil - 1$. Dans ce cas, la complexité parallèle théorique du calcul est égale à $\lceil k/2 \rceil M(n/k) + M(n/k) = (\lceil k/2 \rceil + 1)M(n/k)$ plus le coût de la multiplication qp sur un nombre de processeurs égal à celui donné au théorème 17.

En pratique, nous considérons uniquement le modèle quadratique, plus fidèle à la réalité.

4.6 Résumé des complexités parallèles de la multiplication modulaire

Dans le tableau suivant nous présentons les complexités parallèles des multiplications de Barrett, de Montgomery, bipartite et multipartite. Les complexités des trois premiers algorithmes ont été démontrées au chapitre précédent (équation 3.1 pour la complexité des algorithmes parallèles de Barrett et de Montgomery et équation 3.2 pour la multiplication modulaire bipartite). La complexité de la multiplication multipartite est celle donnée en utilisant le modèle quadratique et la parallélisation de la multiplication qp .

#P.	Barrett/Montg.	Bipartite	Multipartite
2	$3M(n, n/2)$	$2M(n, n/2) + M(n/2)$	-
4	$3M(n/2)$	$2M(n/2) + M(n/2, n/4)$	$3M(n/4) + 2M(n/2)$
6	$3M(n/2, n/3)$	$2M(n/2, n/3) + M(n/2, n/6)$	$M(n/4) + M(n/2) + M(n/2, n/3)$
8	$3M(n/2, n/4)$	$2M(n/2, n/4) + M(n/4)$	$13M(n/8) + M(n/2, n/4)$
12	$3M(n/3, n/4)$	$2M(n/3, n/4) + M(n/4, n/6)$	$M(n/6) + M(n/2, 2n/6) + M(n/2, n/4)$

TABLE 4.3 – Complexité parallèle des méthodes de multiplication modulaire en fonction du nombre de cœurs

Il est clair que la multiplication bipartite a la meilleure complexité théorique quelque soit le nombre de processeurs considéré. Nous verrons cependant qu'en pratique la multiplication multipartite offre une bonne alternative à la méthode bipartite, car elle diminue la dépendance entre les différentes tâches parallèles et réduit ainsi la synchronisation nécessaire. Ce sera en particulier critique pour des tailles d'entiers cryptographiques.

Dans le tableau suivant, nous présentons le nombre total d'opérations nécessaires aux multiplications modulaires précédentes dans le modèle quadratique. Ces quantités sont obtenues en sommant l'ensemble des opérations réalisées par les tâches parallèles.

#P.	Barrett/Montg.	Bipartite	Multipartite
2	$6n^2$	$5n^2$	-
4	$6n^2$	$5n^2$	$5.25n^2$
6	$6n^2$	$5n^2$	$5.25n^2$
8	$6n^2$	$5n^2$	$5.2n^2$
12	$6n^2$	$5n^2$	$5.33n^2$

TABLE 4.4 – Complexité parallèle des méthodes de multiplication modulaire en fonction du nombre de cœurs

Le nombre d'opérations séquentielles à réaliser pour calculer les multiplications parallèles de Barrett, de Montgomery et bipartite ne varient pas, car les découpages sont réalisés pour effectuer la

même quantité de travail. En revanche, le nombre d'opérations nécessaires pour réaliser la multiplication multipartite dépend du découpage. Ce nombre est proche de la multiplication bipartite mais plus important car cette multiplication ajoute de la redondance dans le calcul des quotients partiels. Elle reste cependant meilleure à ce niveau là que les versions parallèles des algorithmes de Barrett et de Montgomery car elle bénéficie au niveau théorique du gain de complexité de la multiplication bipartite. Séquentiellement, si l'algorithme de multiplication entière a complexité quadratique est utilisé, les temps de calcul des multiplications bipartite et multipartite seront ainsi proches et meilleurs que les temps de calcul pour les versions parallèles des algorithmes de Montgomery et de Barrett.

4.7 Résultats expérimentaux

Nous avons ajouté à la bibliothèque PACE plusieurs versions des algorithmes que nous avons présenté dans les parties précédentes. Comme pour la multiplication entière, la parallélisation est réalisée en utilisant l'interface OpenMP. Notre implémentation vise à être générique en termes de nombre de tâches et de taille des opérandes. Cependant, pour optimiser les temps de calcul, nous avons préféré du code spécialisé pour certaines versions de la multiplication multipartite présentées dans la section précédente. Ainsi, pour 4, 6, 8 et 12 processeurs, chaque tâche possède son propre code. Nous donnons dans l'annexe B un exemple de code pour la multiplication multipartite $k = 4$ sur 4 cœurs.

Pour analyser l'efficacité de notre multiplication multipartite, nous avons testé nos implémentations sur un nœud de la plateforme HPC@LR utilisant deux processeurs Intel Xeon X5650 Westmere contenant six cœurs cadencés à 2.66GHz chacun avec un cache de type L3 de 12Mo. Nos fonctions sont basées sur la version 5.0.2 de la bibliothèque GMP. Nous avons réalisé dans un premier temps la comparaison des multiplications modulaires parallèles pour des entiers de taille 2^{10} à 2^{17} bits. Quatre méthodes différentes sont ici considérées : les algorithmes de Montgomery et de Barrett utilisant l'arithmétique entière parallèle définie dans le chapitre précédent ; et les méthodes bipartite et multipartite utilisant le parallélisme aux deux niveaux arithmétiques. Pour confirmer les résultats obtenus par la multiplication, nous avons également réalisé une série de mesures de temps pour l'exponentiation modulaire utilisant ces différentes multiplications. Comme nous le verrons, nos résultats mettent en avant le potentiel de la multiplication multipartite pour des tailles d'entiers réduites (entre 2^{11} et 2^{15} bits), tailles qui sont fréquemment utilisées en cryptographie asymétrique.

4.7.1 Multiplication modulaire parallèle

Les versions parallèles des algorithmes de Barrett et de Montgomery ainsi que de la méthode bipartite que nous avons testées sont celles décrites au chapitre précédent, qui utilisent un code spécialisé pour chaque nombre de processeurs différent, réduisant donc l'*overhead* lié au parallélisme. Pour la multiplication multipartite, il existe plusieurs découpages et ordonnancements possibles pour un même nombre de processeurs. Le tableau 4.5 présente les ordonnancements utilisés pour nos tests ainsi que leur complexité ; où P représente le calcul d'un produit partiel $a_i b_j$, B le calcul d'un quotient partiel pour les produits bas et H le calcul d'un quotient partiel pour les produits hauts. Toutes les tâches participent au calcul de la multiplication qp en même proportion. En pratique, sur l'ensemble des expérimentations réalisées, ce sont ces découpages et ordonnancements qui donnent les meilleurs temps de calcul.

Nous avons décidé de confronter les performances des différentes multiplications modulaires considérées en comparant leur efficacité par rapport à la multiplication modulaire de la bibliothèque GMP. La figure 4.5 présente ces résultats pour un parallélisme sur 4, 6, 8 et 12 cœurs. De manière similaire aux résultats obtenus pour la multiplication entière (voir chapitre précédent), l'*overhead* introduit par le parallélisme domine le coût total du calcul pour les petites tailles, ici inférieures à 2048 bits. Pour des tailles intermédiaires, entre 2048 et 16384 bits, la multiplication multipartite est toujours plus performante que les autres méthodes. En pratique, cela s'explique par le fait que l'*overhead* reste important dans le coût total du calcul. Or la multiplication multipartite permet des calculs bien moins dépendants les uns des autres que les autres méthodes, donc nécessite moins de synchronisation. Pour des tailles supérieures à 16384 bits, la multiplication bipartite devient meilleure, car l'*overhead* est maintenant totalement amorti et la complexité théorique de cette méthode est meilleure que les complexités des autres multiplications. Un cas particulier est remarquable : lorsque 6 cœurs sont utilisés,

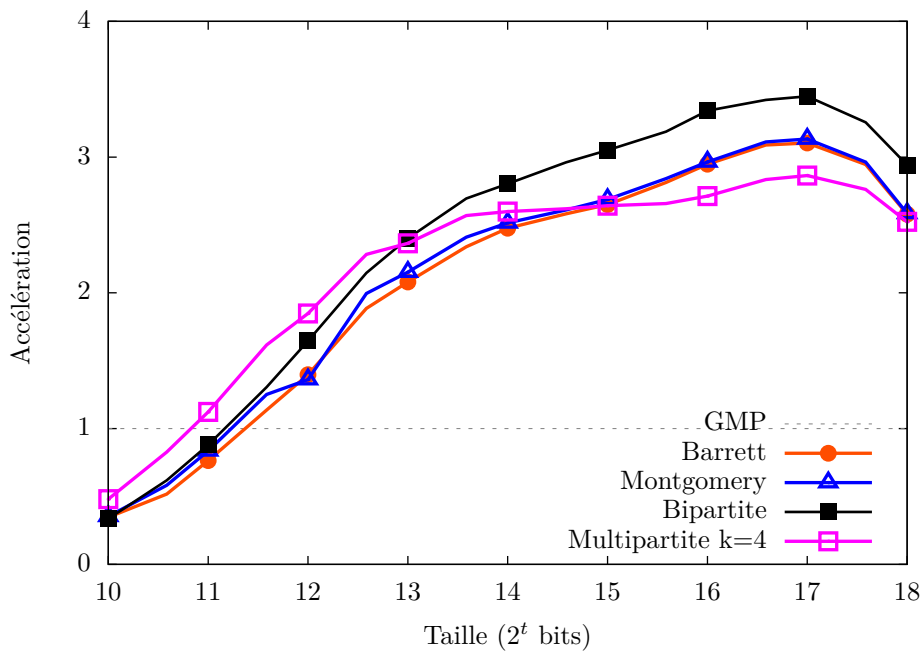
#P.	k	P.	Ordonnancement	Complexité
4	4	0	3P+B	$3M(n/4) + 2M(n/2)$
		1	5P+B	$6M(n/4) + M(n/2)$
		2	5P+H	$6M(n/4) + M(n/2)$
		3	3P+H	$3M(n/4) + 2M(n/2)$
6	4	0	P+B	$M(n/4) + M(n/2) + M(n/2, n/3)$
		1	3P+B	$3M(n/4) + M(n/4) + M(n/2, n/3)$
		2	4P	$4M(n/4) + M(n/2, n/3)$
		3	4P	$4M(n/4) + M(n/2, n/3)$
		4	3P+H	$3M(n/4) + M(n/4) + M(n/2, n/3)$
		5	P+H	$M(n/4) + M(n/2) + M(n/2, n/3)$
8	8	0	4P+B	$4M(n/8) + M(n/2, 2n/8) + M(n/2, n/4)$
		1	7P+B	$7M(n/8) + M(3n/8, 2n/8) + M(n/2, n/4)$
		2	9P+B	$9M(n/8) + M(2n/8) + M(n/2, n/4)$
		3	12P+B	$13M(n/8) + M(n/2, n/4)$
		4	12P+H	$13M(n/8) + M(n/2, n/4)$
		5	9P+H	$9M(n/8) + M(2n/8) + M(n/2, n/4)$
		6	7P+H	$7M(n/8) + M(3n/8, 2n/8) + M(n/2, n/4)$
		7	4P+H	$4M(n/8) + M(n/2, 2n/8) + M(n/2, n/4)$
12	6	0	P+B	$M(n/6) + M(n/2, 2n/6) + M(n/2, n/4)$
		1	2P+B	$2M(n/6) + M(2n/6) + M(n/2, n/4)$
		2	3P+B	$3M(n/6) + M(n/6) + M(n/2, n/4)$
		3	4 P	$4M(n/6) + M(n/2, n/4)$
		4	4 P	$4M(n/6) + M(n/2, n/4)$
		5	4 P	$4M(n/6) + M(n/2, n/4)$
		6	4 P	$4M(n/6) + M(n/2, n/4)$
		7	4 P	$4M(n/6) + M(n/2, n/4)$
		8	4 P	$4M(n/6) + M(n/2, n/4)$
		9	3P+H	$3M(n/6) + M(n/6) + M(n/2, n/4)$
		10	2P+H	$2M(n/6) + M(2n/6) + M(n/2, n/4)$
		11	P+H	$M(n/6) + M(n/2, 2n/6) + M(n/2, n/4)$

TABLE 4.5 – Ordonnements utilisés sur 4, 6, 8, et 12 cœurs pour la multiplication multipartite
P : Produit $a_i b_j$; B : calcul d'un quotient partiel de produits bas et H : calcul d'un quotient partiel de produits hauts.

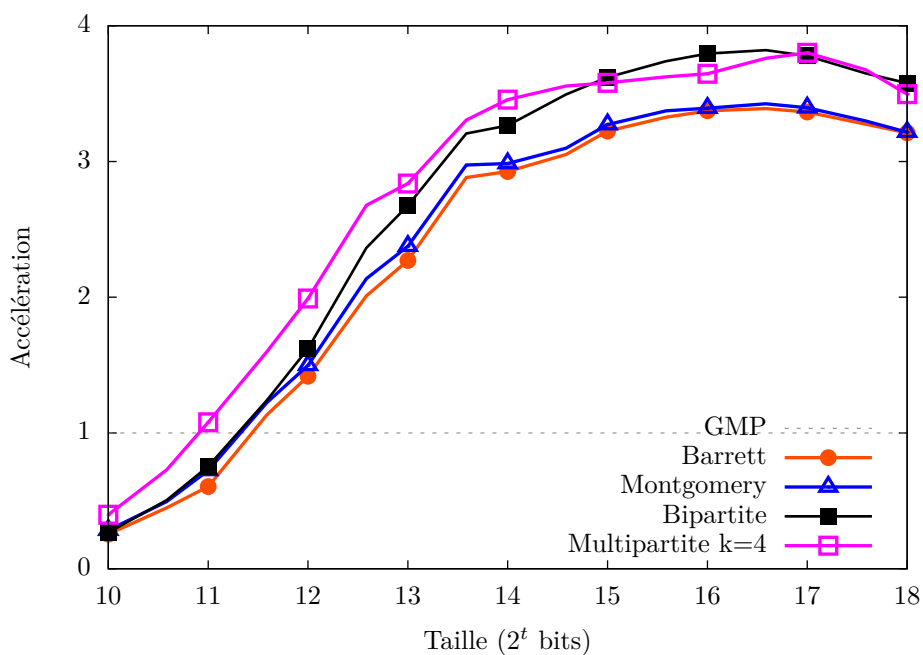
les performances de la multiplication multipartite sont très proches des performances de multiplication bipartite, ce qui n'est pas le cas pour les autres nombres de processeurs. Il faut se rappeler que nous avons basé notre stratégie sur l'hypothèse d'une complexité quadratique de la multiplication entière, alors qu'en pratique GMP utilisera les meilleurs algorithmes pour les tailles considérées. Or la stratégie choisie dans le cas où 6 processeurs sont utilisées reste valide lorsque l'algorithme de multiplication FFT est utilisé. En construisant des stratégies basées sur les hypothèses de multiplications en temps sous-quadratique ou quasi-linéaire, nous pourrions donc augmenter l'efficacité de notre multiplication multipartite. Nos résultats montrent également que les algorithmes de Barrett et de Montgomery ne sont pas les mieux adaptés au parallélisme logiciel.

4.7.2 Exponentiation parallèle modulaire

L'exponentiation modulaire est un des exemples classiques d'utilisation de la multiplication modulaire. Cette opération est au cœur de nombreux cryptosystèmes, comme RSA [80] ou El Gamal [29]. Pour évaluer les performances de notre multiplication multipartite nous avons implémenté l'algorithme d'exponentiation *Square-and-multiply* (la version multiplicative de l'algorithme 7) avec plusieurs multiplications modulaires différentes. Cet algorithme calcule $g^e \bmod p$ avec $g, e < p < 2^n$ en évaluant



(a) 4 Cœurs



(b) 6 Coeurs

FIGURE 4.5 – Comparaison des performances des multiplications modulaires de Barrett, de Montgomery, bipartite et multipartite

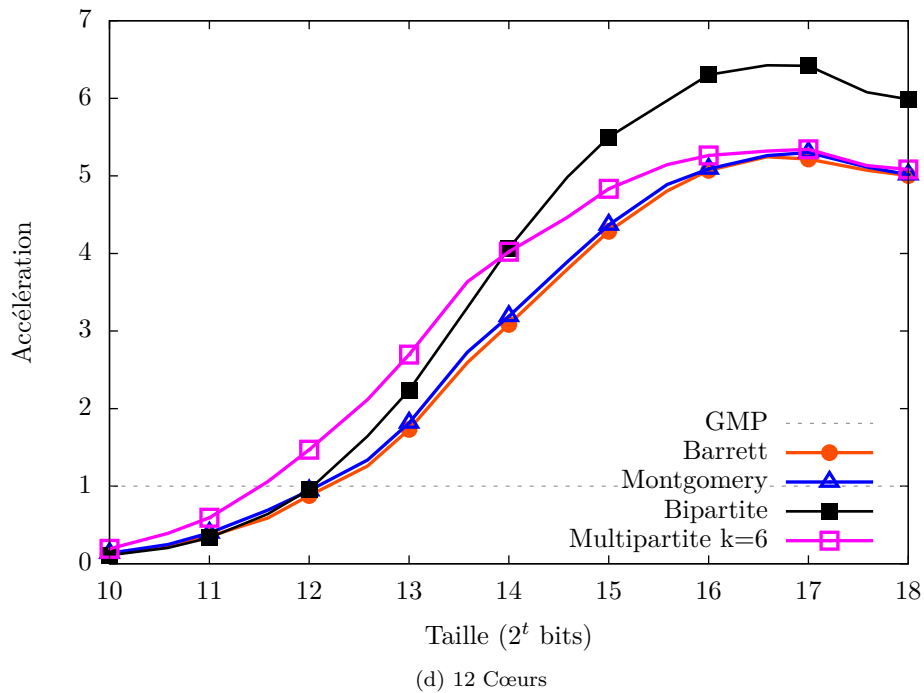
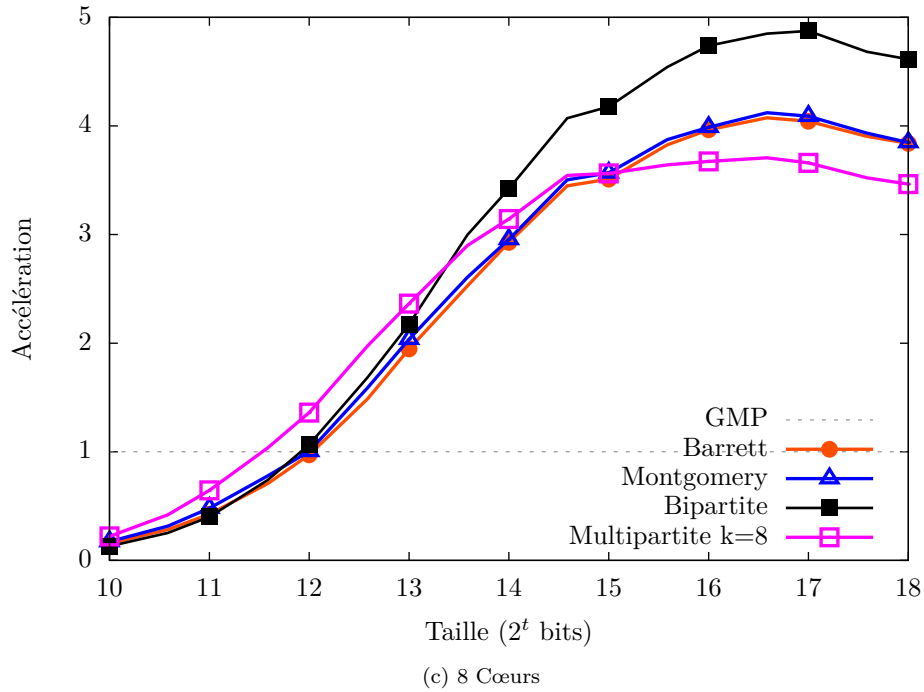


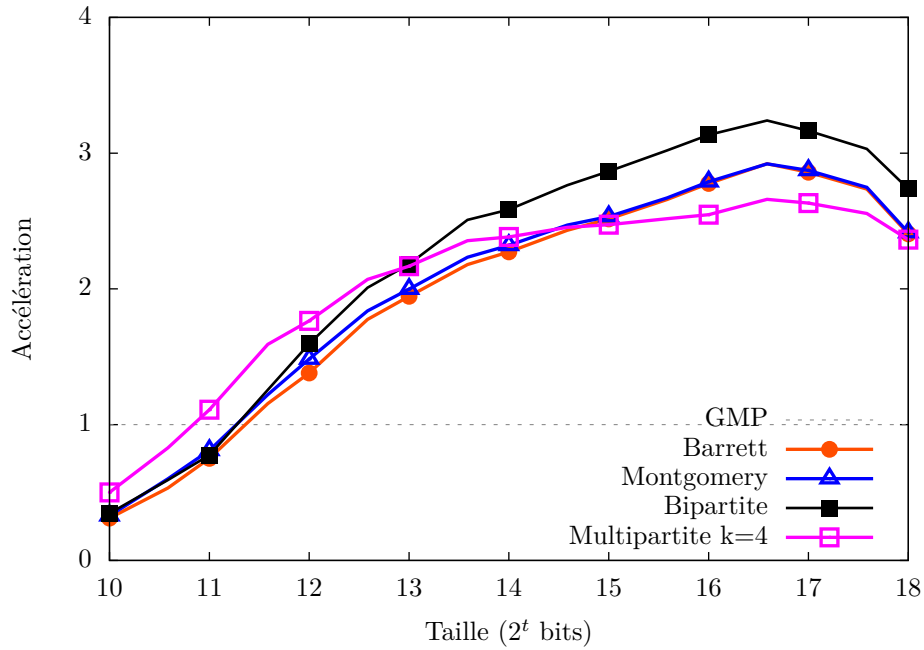
FIGURE 4.5 – Comparaison des performances des multiplications modulaires de Barrett, de Montgomery, bipartite et multipartite

n carrés et en moyenne $n/2$ multiplications modulaires. Pour simplifier notre étude, nous utilisons néanmoins uniquement l'opérateur de multiplication modulaire. L'algorithme *Square-and-Multiply* est générique dans le sens où nous pouvons modifier uniquement la multiplication modulaire utilisée. On garantit ainsi que les différences des performances proviennent essentiellement des différences entre les différents opérateurs utilisés. Théoriquement, le temps pour calculer cette exponentiation est donc le temps pour calculer $3n/2$ multiplications modulaires. Dans chacun des cas, nous lançons une seule fois les tâches parallèles qui se synchronisent uniquement durant la multiplication modulaire.

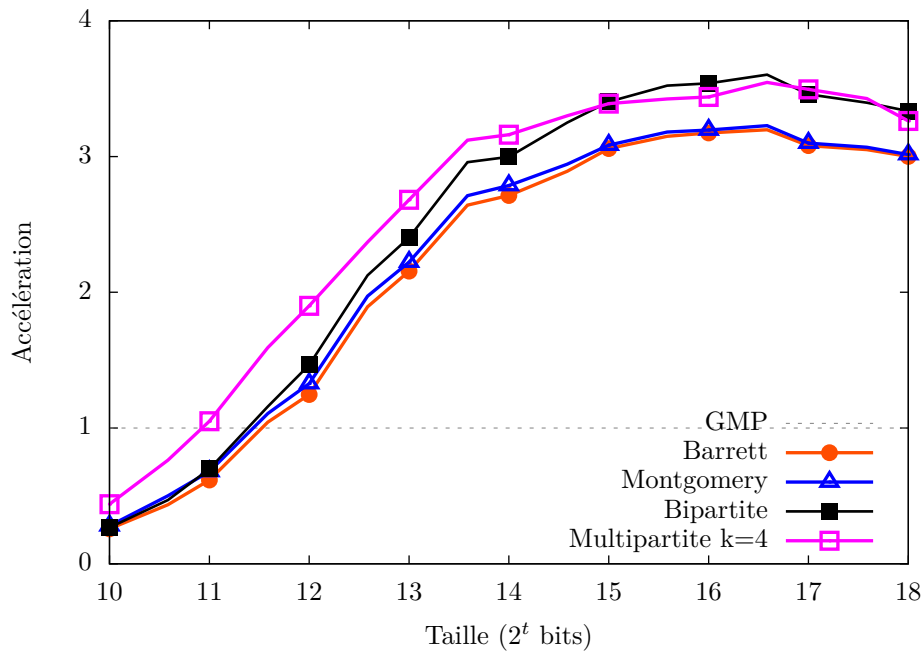
Les performances de nos implémentations sont comparées par rapport à une version de l'algorithme basée sur la multiplication modulaire de GMP composée d'un appel à la fonction `mpz_mul` suivi d'un appel à `mpz_tdiv_r`. Elles sont présentées à la figure 4.6.

Logiquement, les courbes sont similaires à celles de la multiplication modulaire présentées à la figure 4.5. Les exponentiations les plus performantes sont obtenues avec les multiplications les plus performantes. On peut remarquer que les accélérations sont cependant inférieures à celles de la multiplication, mais cela s'explique par le fait que l'opérateur de multiplication de GMP utilise nativement l'opération de mise au carré quand il le peut, ce que nous ne faisons pas.

Pour des tailles cryptographiques (entre 2^{11} et jusqu'à 2^{15} sur 6 cœurs), notre multiplication multipartite donne les meilleures performances pour l'exponentiation.

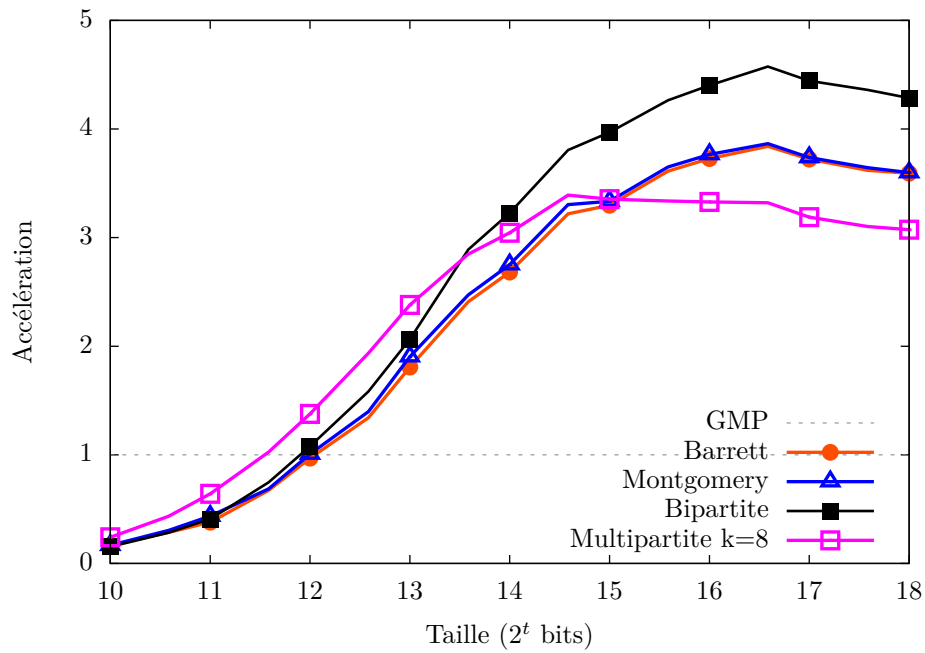


(a) 4 Coeurs

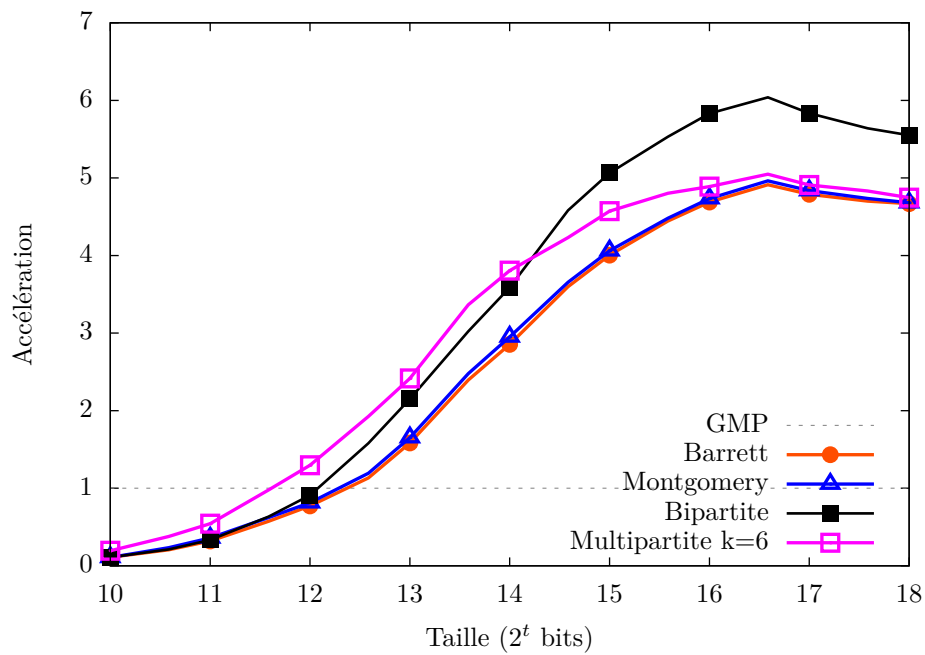


(b) 6 Coeurs

FIGURE 4.6 – Comparaison des performances d’une exponentiation modulaire *Square-and-multiply* basée sur nos implémentations parallèles des multiplications de Barrett, de Montgomery, bipartite et multipartite



(c) 8 Cœurs



(d) 12 Cœurs

FIGURE 4.6 – Comparaison des performances d’une exponentiation modulaire *Square-and-multiply* basée sur nos implémentations parallèles des multiplications de Barrett, de Montgomery, bipartite et multipartite

4.8 Conclusion et perspectives

La méthode de multiplication modulaire multipartite que nous proposons généralise l'algorithme bipartite. Elle permet un parallélisme au niveau de l'arithmétique modulaire et peut bénéficier d'un parallélisme au niveau de l'arithmétique multiprécision. Nous avons réalisé un grand nombre d'expérimentations pour comparer cette nouvelle méthode aux algorithmes de Barrett, de Montgomery avec un parallélisme de l'arithmétique entière, et à la méthode bipartite qui utilise un parallélisme aux deux niveaux arithmétiques. Les performances obtenues montrent l'intérêt de la méthode pour des tailles d'entiers utilisées en cryptographie. L'ordonnancement *optimal* des calculs a été fait à la main en posant des hypothèses sur les complexités de la multiplication entière. Une méthode générale pour déterminer ces optimisations est souhaitable avec des architectures qui évoluent en permanence et disposent d'un nombre croissant d'unités de calcul, mais semble difficile à mettre en place en pratique. En effet, comparer les complexités de chaque partie du calcul hors du cadre quadratique est compliqué, car ces complexités ne sont pas homogènes et les algorithmes utilisés pour la multiplication entière peuvent changer d'un produit à l'autre en fonction des tailles des opérandes.

Cette méthode de multiplication ouvre de nouvelles perspectives pour le calcul modulaire parallèle. La première question qui se pose est l'intérêt d'utiliser un schéma sous-quadratique pour le découpage de nos produits. Cette idée est particulièrement intéressante, car un tel découpage permettra une réduction du nombre d'unités de calcul et donnera ainsi une meilleure flexibilité, pour un nombre de processeur fixé, qu'un découpage quadratique. En pratique cependant, il faut prendre en considération que ces découpages particuliers nécessitent des synchronisations entre les différentes tâches, car les calculs ne seront plus totalement indépendants. Il n'est pas certain au vu du coût faramineux de la synchronisation par rapport aux calculs arithmétiques que les gains soient conséquents pour des entiers de petites tailles.

Nous détaillons dans la suite deux perspectives majeures que nous avons commencées à étudier : le carré multipartite $a^2 \bmod p$ et surtout la réduction multipartite où les $n/2$ mots de poids fort et les $n/2$ mots de poids faibles d'un entier c de taille $2n$ seront réduits par un certains nombre de calculs indépendants.

4.8.1 Un carré multipartite

Le carré entier est considéré comme moins coûteux que la multiplication car certains calculs redondants peuvent être calculés une seule fois. Par exemple, si $a = a_1\beta^{n/2} + a_0$ et $b = b_1\beta^{n/2} + b_0$, $ab = a_1b_1\beta^n + (a_0b_1 + a_1b_0)\beta^{n/2} + a_0a_0$. Or, si $b = a$, $a_0b_1 = a_1b_0 = a_0a_1$, soit trois produits au lieu de quatre. Le même principe est applicable pour dériver un carré multipartite à partir de la multiplication modulaire, qui permettra de diminuer le nombre de produits partiels à calculer. Ainsi, on baissera le nombre d'unités de calcul à allouer au parallélisme au niveau modulaire.

Le nombre de produits $a_i b_j$ dans la multiplication multipartite tels que $i \neq j$ est $k(k-1)$. Le nombre de produits $a_i b_j$ différents dans le carré est donc $k(k-1)/2$, car pour chaque produit $a_i a_j$ avec $i \neq j$ il existe un produit croisé $a_j a_i$. Le nombre de produits différents à calculer dans le carré multipartite est donc $k(k+1)/2$, et le nombre de réductions des produits haut (ou des produits bas) est donné par le lemme suivant.

Lemme 4. Le nombre de produits modulaire dans le carré multipartite est égal à :

$$\begin{cases} \frac{k(k+4)}{8} & \text{si } k \equiv 0 \pmod{4} \\ \frac{(k+3)^2}{8} & \text{si } k \equiv 1 \pmod{4} \\ \frac{k(k+4)+4}{8} & \text{si } k \equiv 2 \pmod{4} \\ \frac{(k+1)(k+5)}{8} & \text{si } k \equiv 3 \pmod{4} \end{cases}$$

Démonstration. D'après le lemme 2, il y a $\lceil k/2 \rceil (\lceil k/2 \rceil + 1)$ produits modulaires à calculer. Comme $0 \leq i+j \leq \lceil k/2 \rceil - 1$, il a $2\lceil k/4 \rceil$ produits tels que $i = j$. Il y a donc $\lceil k/2 \rceil (\lceil k/2 \rceil + 1) - 2\lceil k/4 \rceil$ produits tels que $i \neq j$, et on peut en calculer seulement la moitié. Le nombre total de produits modulaires dans le carré multipartite est donc égal à :

$$\lceil k/2 \rceil (\lceil k/2 \rceil + 1) - 1/2 [\lceil k/2 \rceil (\lceil k/2 \rceil + 1) - 2\lceil k/4 \rceil].$$

□

Un carré multipartite peut donc être intéressant en pratique, notamment pour les algorithmes d'exponentiation modulaire.

4.8.2 Une réduction multipartite

Pour un découpage k , notre méthode de multiplication multipartite doit calculer k^2 produits dont certains doivent être réduits. L'avantage de cette approche est de permettre des calculs totalement parallèles avec une seule synchronisation des tâches à la fin des calculs. Si en théorie elle donne de bon résultat, en pratique elle peut s'avérer délicate en mettre en œuvre car les architectures actuelles proposent un nombre d'unités de calcul limité. Pour pallier ce problème, une approche multipartite de la réduction semble intéressante. Dans ce cas, la multiplication $c = ab$ pourra être réalisée en parallèle grâce aux opérateurs décrits dans le chapitre précédent. Ensuite, la réduction sera également parallélisée par k appels à l'algorithme de réduction partielle de Barrett sur les $n/2$ mots de poids fort de c divisés en k parties; et k appels à l'algorithme de réduction partielle de Montgomery sur les $n/2$ mots de poids faible de c divisés en k parties. La figure 4.7 présente ce principe avec $k = 3$.

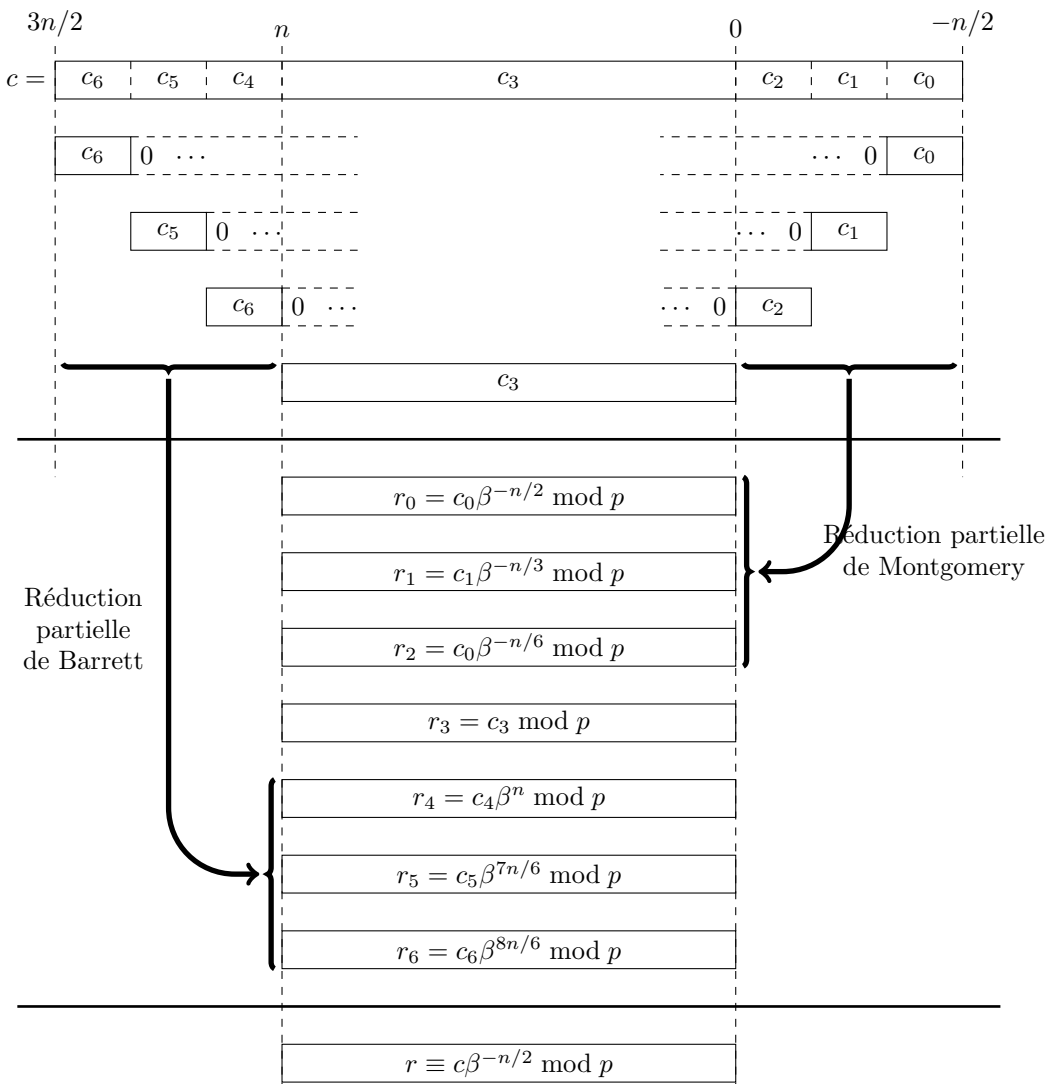
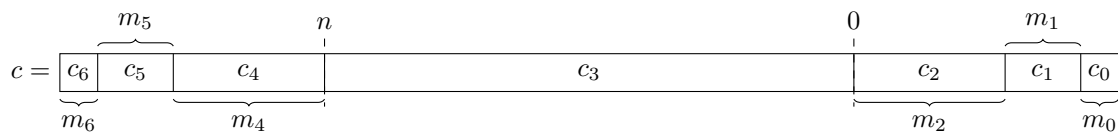


FIGURE 4.7 – Réduction multipartite avec $k = 3$: trois réductions pour les $n/2$ mots de poids fort de c , trois réductions pour les $n/2$ mots de poids faible

Appliqué naïvement, la complexité parallèle de cette réduction sera égale à la complexité maximale des réductions partielles, c'est-à-dire la complexité de la réduction des parties extrémales, c_0 et c_6 dans l'exemple. Cette complexité est égale à $M(n/2, n/2k) + M(n, n/2)$.

Cependant, des méthodes plus astucieuses sont désormais possibles grâce à la flexibilité introduite par cette réduction. En effet, le parallélisme intrinsèque au niveau de l'arithmétique modulaire nécessite moins d'unités de calcul que la multiplication multipartite. Dès lors, on pourra utiliser les unités de calcul ainsi libérées pour ajouter un parallélisme au niveau de l'arithmétique entière sous-jacente aux calculs. Par exemple, on peut imaginer attribuer deux fois plus de processeurs à la réduction des parties extrémales et ainsi baisser la complexité parallèle de la réduction totale. Pour baisser un peu plus la complexité de cette réduction, on peut également imaginer un découpage non uniforme produisant des parties de taille m_0, \dots, m_l telles que si m_0 est la taille des parties extrémales et m_l la taille des parties dont la réduction est normalement la moins coûteuse on ait : $m_0 \leq m_1 \leq \dots \leq m_l$. Ce découpage hétérogène est illustré par l'exemple suivant où $m_2 = 2m_1 = 4m_0$ et $m_4 = 2m_5 = 4m_6$ soit $m_0 = m_6 = n/14$, $m_1 = m_5 = 2n/14$ et $m_2 = m_4 = 4n/14$.



Les complexités de ces réductions sont les suivantes :

- pour la réduction de c_0 et c_6 la complexité est $M(n/2, n/14) + M(n, n/2)$;
- pour la réduction de c_1 et c_5 la complexité est $M(6n/14, 2n/14) + M(n, 6n/14)$;
- pour la réduction de c_2 et c_4 la complexité est $M(4n/14) + M(n, 4n/14)$.

Comme nous l'avons vu, il est difficile de comparer ces complexités non homogènes sans poser d'hypothèse sur l'algorithme de multiplication utilisé. Il est cependant clair que ces découpages hétérogènes permettront de lisser les complexités. En jouant sur les paramètres m_i , nous pourrions proposer des ordonnancements optimaux en fonction des différentes hypothèses de complexité.

Enfin, ces deux techniques peuvent également se combiner : un découpage hétérogène pour diminuer le coût de la partie calcul du quotient des termes extrémaux puis utilisation des opérateurs de multiplication parallèle pour réduire la complexité du calcul qp .

CHAPITRE 5

OPTIMISATION DES FORMULES POUR LA MULTIPLICATION SCALAIRE

Sommaire

5.1	Formules	92
5.1.1	Définition	92
5.1.2	Représentation des formules	94
5.1.3	Le problème de l'optimisation	94
5.2	Une approche pour l'optimisation automatique	96
5.2.1	Élimination des sous-expressions communes	96
5.2.2	Transformations arithmétiques	98
5.2.3	Straight-Line programs	98
5.3	Une fonction Hash heuristique	99
5.3.1	Le problème <i>Ensemble Computation</i>	102
5.4	Implémentation et résultats	102
5.4.1	Implémentation	102
5.4.2	Formule de quintuplement	102
5.4.3	Cas spéciaux	105
5.4.4	Production de code	105
5.5	Conclusion et perspectives	107

Nous avons présenté au chapitre 2 les algorithmes classiques de multiplication scalaire qui nécessitent la définition d'au moins deux opérations dans le groupe $E(\mathbb{F}_p)$: l'addition et le doublement. Certains algorithmes doivent en outre pré-calculer des valeurs $[s]P$ pour un s petit (quelques bits). D'autres peuvent tirer parti d'un calcul $[s]P'$ à la volée, c'est-à-dire une multiplication par un petit scalaire d'un point P' précédemment calculé. Ces multiplications par de petits scalaires (ainsi que l'opération d'addition) consistent en une série d'opérations, nommée *formule*, dans le corps de définition de la courbe elliptique (\mathbb{F}_p dans notre cas).

Une formule pour calculer $[s]P$ n'est pas unique : on peut définir un ensemble de formules équivalentes modulo un ensemble de transformations arithmétiques. Optimiser une formule revient donc à trouver une formule équivalente réalisant le même calcul $[s]P$ avec une série moins coûteuse d'opérations arithmétiques.

Dans ce chapitre, notre objectif est de définir une méthodologie pour permettre l'optimisation du calcul de $[s]P$ de manière automatique. Ce travail a été publié dans [39]. Ce problème est difficile au vu de l'accroissement exponentiel à la fois de la taille des formules et du nombre de transformations arithmétiques possibles.

Après avoir défini formellement le problème d'optimisation de formules, nous présentons les deux grandes étapes de notre méthode : l'élimination des sous-expressions communes et la réduction du coût arithmétique grâce à des transformations arithmétiques. L'élimination des sous-expressions communes implique de pouvoir évaluer l'équivalence de deux sous-expressions. Si en théorie cette évaluation est simple, elle est cependant trop coûteuse en temps de calcul et surtout en utilisation de la mémoire. Nous présentons donc une heuristique permettant une implémentation efficace. Enfin, nous donnons les résultats obtenus par notre programme : une nouvelle formule de quintuplement ainsi que le coût arithmétique pour un ensemble de cas spéciaux ($s = 2^i + 2^j$ par exemple).

Toutes les courbes elliptiques considérées sont définies sur \mathbb{F}_p et le système de coordonnées jacobiniennes est utilisé pour les exemples, c'est-à-dire que le point P est représenté par le triplet $(X_P : Y_P : Z_P)$. Le scalaire s représente un petit entier (entre 2 et 2^7) et k un élément quelconque de \mathbb{N}^* .

5.1 Formules

Nous commençons par définir formellement une formule ainsi que la structure de données que nous utilisons par la suite. Nous pourrions alors définir le problème d'optimisation de formules.

5.1.1 Définition

Dans la suite, nous notons M le nombre de variables nécessaires à la représentation d'un point dans le système de coordonnées considéré. En coordonnées affines, deux variables (x et y) sont nécessaires pour coder toute l'information relative aux points de la courbe. Trois variables (X, Y, Z) sont nécessaires en coordonnées projectives. Pour simplifier les calculs, on peut ajouter de la redondance en introduisant de nouvelles coordonnées calculées en fonctions des précédentes. C'est le cas par exemple des coordonnées de Chudnovsky où les points sont représentés par un ensemble de cinq variables (X, Y, Z, Z^2, Z^3) qui représentent le point affine $(X/Z^2, Y/Z^3)$.

Définition 5. Soit un point $P = (V_0, \dots, V_{M-1}) \in E(\mathbb{F}_p)$ représenté dans un système de coordonnées à M variables. On appelle *formule* l'application :

$$\begin{aligned} \mathcal{F} : E(\mathbb{F}_p) &\rightarrow E(\mathbb{F}_p) \\ P = (V_0, \dots, V_{M-1}) &\mapsto Q = [s]P = (W_0, \dots, W_{M-1}) \end{aligned}$$

c'est-à-dire l'ensemble des opérations qui permettent le calcul de $[s]P$ à partir de P .

Les termes des formules sont catégorisés en trois grands types : les variables (V_0, \dots, V_{M-1}) , les constantes (paramètres de la courbe et petits entiers) et les opérateurs arithmétiques.

Définition 6. Deux formules \mathcal{F} et \mathcal{G} sont équivalentes (noté $\mathcal{F} \sim \mathcal{G}$) si et seulement si pour tous les tuples (V_0, \dots, V_{M-1}) :

$$\mathcal{F}(V_0, \dots, V_{M-1}) = \mathcal{G}(V_0, \dots, V_{M-1}).$$

Exemple 13 (Formules de doublement du point $P = (X_P : Y_P : Z_P)$).

Formule \mathcal{F} :

$$\begin{aligned}
XX &= X_P^2 \\
YY &= Y_P^2 \\
ZZ &= Z_P^2 \\
S &= 4 \times X_P \times YY \\
M &= 3 \times XX + a \times ZZ^2 \\
T &= M^2 - 2 \times S \\
X_{2P} &= T \\
Y_{2P} &= M \times (S - T) - 8 \times YY^2 \\
Z_{2P} &= 2 \times Y_P \times Z_P
\end{aligned}$$

Formule \mathcal{G} :

$$\begin{aligned}
XX &= X_P^2 \\
YY &= Y_P^2 \\
YYYY &= YY^2 \\
ZZ &= Z_P^2 \\
S &= 2 \times ((X_P + YY)^2 - XX - YYYY) \\
M &= 3 \times XX + a \times ZZ^2 \\
T &= M^2 - 2 \times S \\
X_{2P} &= T \\
Y_{2P} &= M \times (S - T) - 8 \times YYYY \\
Z_{2P} &= (Y_P + Z_P)^2 - YY - ZZ
\end{aligned}$$

Les formules \mathcal{F} et \mathcal{G} sont équivalentes. Elles calculent toutes deux $[2]P$.

Remarque 27. La formule \mathcal{F} est la formule la plus efficace connue à ce jour pour calculer le doublement en système de coordonnées jacobiennes. Cette formule est déjà optimisée.

Pour comparer l'efficacité de deux formules équivalentes, nous introduisons la notion de coût arithmétique.

Définition 7. Le coût arithmétique d'une formule, noté $C(\mathcal{F})$, est la somme des coûts de chaque opération qui la compose.

Nous utilisons les notations suivantes pour définir le coût des opérations réalisées dans \mathbb{F}_p avec p de taille n :

Notation	Opération
I	Inversion
M	Multiplication entre éléments de taille n
S	Carré d'un élément de taille n
A	Addition et soustraction d'éléments de taille n
m	Multiplication par un élément du corps \mathbb{F}_p (exemple : paramètre a de la courbe)
c	Multiplication par de petites constantes (quelques bits)

Il est possible de définir une relation d'ordre entre ces coûts : $M \geq S \gg A$ avec $c \approx A$. Placer m est plus difficile car la constante peut être un petit élément du corps (auquel cas $m \approx c$) ou un grand élément du corps ($m \approx M$).

Exemple 14 (Coût des formules de l'exemple 13).

$$\begin{aligned}
C(\mathcal{F}) &= 6S + 3M + 1m + 5c + 4A \\
C(\mathcal{G}) &= 8S + 1M + 1m + 4c + 10A
\end{aligned}$$

On considère en général uniquement les multiplications de coût M et les carrés S qui sont les deux opérations les plus coûteuses. Le tableau 5.1 donne le coût de trois opérations sur une courbe elliptique E pour quelques systèmes de coordonnées. Ils sont extraits de la base de données *Explicit-Formulas Database*¹ qui compile les formules connues pour un large ensemble de courbes et de systèmes de représentation.

Pour pouvoir traiter nos formules (appliquer des algorithmes, appliquer des transformations arithmétiques, etc.) nous définissons une structure de données classique dans laquelle une formule est représentée par un arbre binaire.

1. <http://www.hyperelliptic.org/EFD/index.html>

Système	Doublement	Addition	Addition Mixte
Affine	$1I, 2M, 2S$	$1I, 2M, 1S$	-
Projectif ($c = 1, d = 1$)	$5M, 6S$	$12M, 2S$	$9M, 2S$
Jacobien	$1M, 8S$	$11M, 5S$	$7M, 4S$
Edwards [28]	$3M, 4S$	$10M, 1S$	$9M$
Edwards inversé [10]	$3M, 4S$	$9M, 1S$	$8M, 1S$

TABLE 5.1 – Tableau des coûts de l’addition et du doublement de points d’une courbe elliptique définie sur \mathbb{F}_p dans différents systèmes de coordonnées (I : inversion dans le corps).

5.1.2 Représentation des formules

Nous utilisons la structure classique d’un arbre : un ensemble de nœuds X et un ensemble d’arêtes E .

Définition 8 (Arbre de formule). L’arbre $\mathcal{T} = (X, E)$ représentant la formule \mathcal{F} est un arbre binaire dans lequel :

- $X = \{ \text{termes de } \mathcal{F} \}$;
- $E = \{ \text{relations entre les opérateurs et leur(s) opérande(s)} \}$.

Les feuilles de l’arbre correspondent aux variables et aux constantes de la formule. Les nœuds internes correspondent aux opérateurs arithmétiques.

Nous définissons également les fonctions classiques $\mathbf{fg}(x)$ et $\mathbf{fd}(x)$ qui renvoient respectivement les nœuds fils gauche et fils droit du nœud $x \in X$ s’ils existent, \emptyset sinon. Enfin, nous définissons une fonction $\mathit{Cal}(x)$ qui représente de manière générique le calcul du sous-arbre enraciné en x .

La figure 5.1 présente l’arbre de la formule de doublement en coordonnées jacobienne (équation 2.7). Pour pouvoir travailler sur l’arbre (et non sur une forêt), nous enracinons virtuellement chacune des composantes connexes de l’arbre.

Nous pouvons maintenant introduire le problème de l’optimisation de formule.

5.1.3 Le problème de l’optimisation

Définition 9. Étant donnée une formule \mathcal{F} pour calculer $[s]P$, le Problème d’optimisation de Formule consiste à trouver une formule $\mathcal{G} \sim \mathcal{F}$ pour calculer $[s]P$ telle que $\forall \mathcal{F}_i \sim \mathcal{F}, C(\mathcal{G}) \leq C(\mathcal{F}_i)$. En particulier, $C(\mathcal{G}) \leq C(\mathcal{F})$.

Exemple 15. Dans l’exemple précédent, la formule \mathcal{G} est moins coûteuse que la formule \mathcal{F} . C’est aujourd’hui la formule la plus rapide pour calculer le doublement dans le système de coordonnées jacobienne. On ne peut cependant pas prétendre qu’il s’agit de la formule optimale pour ce calcul.

De nombreux papiers présentent des optimisations dans différents systèmes de coordonnées. On notera en particulier [9, 8, 20, 61, 62, 60, 59, 81, 45]. Ces optimisations sont principalement basées sur des réarrangements de formules dans lesquels les calculs redondants ont été éliminés, ou sur l’utilisation de propriétés mathématiques.

Il existe deux difficultés majeures à l’optimisation de formules. Premièrement, l’explosion exponentielle de leur taille lorsque s augmente, comme illustré dans la table 5.2 qui donne le nombre de termes des formules pour calculer certaines opérations. Ces formules ont été obtenues avec l’algorithme classique et l’algorithme NAF.

Le second problème à une optimisation efficace est posé par la combinatoire des transformations arithmétiques que l’on peut appliquer à la formule. Cette combinatoire est elle aussi exponentielle. Il faut donc trouver des transformations arithmétiques performantes et qui peuvent être appliquées de manière systématique.

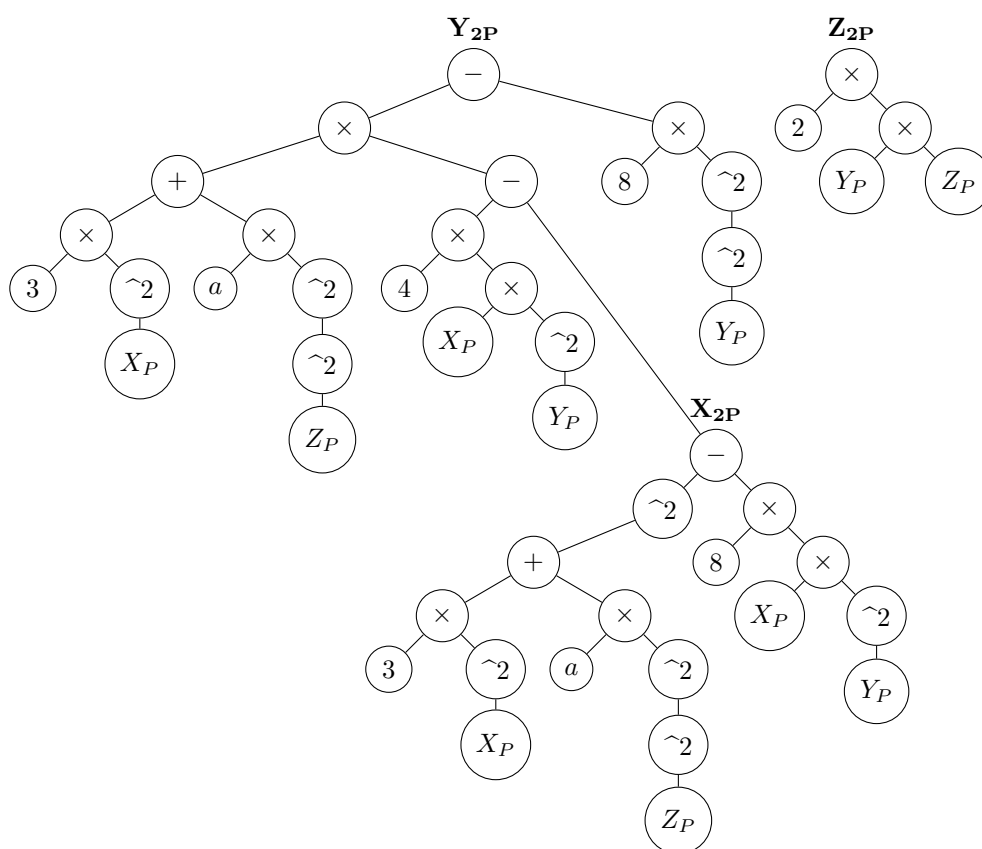


FIGURE 5.1 – Arbre de la formule de doublement en coordonnées jacobiennes

Algorithm	[2]P	P + Q	[3]P	[7]P	[13]P	[17]P
Doublement-Addition	98	204	953	71898	611318	564585
NAF	98	204	8005	67811	5066371	564585

TABLE 5.2 – Nombre de termes des formules pour différents calculs (coordonnées jacobiennes)

L'exemple suivant présente le calcul de la coordonnée Y dans le calcul de $[3]P$ en système jacobien sur les courbes de Weierstrass. Comment factoriser à la main un tel calcul ? Plus important, comment l'optimiser ?

Exemple 16. Calcul de Y_{3P} dans le calcul $[3]P = [2]P + P$

$$Y_{3P} = ((2 \times (((2 \times (((Y1+Z1)^2 - (Y1^2+Z1^2))^2 \times X1) - (((3 \times X1^2) + (a \times Z1^4))^2 - (4 \times ((X1+Y1^2)^2 - (X1^2+Y1^4))) \times Z1^2)))^2 \times (((3 \times X1^2) + (a \times Z1^4))^2 - (4 \times ((X1+Y1^2)^2 - (X1^2+Y1^4))) \times Z1^2)) - ((2 \times ((Y1 \times (((Y1+Z1)^2 - (Y1^2+Z1^2)) \times ((Y1+Z1)^2 - (Y1^2+Z1^2)^2)) - Z1^2 \times (((2 \times ((X1+Y1^2)^2 - (X1^2+Y1^4))) - ((3 \times X1^2) + (a \times Z1^4))^2 - (4 \times ((X1+Y1^2)^2 - (X1^2+Y1^4))) \times Z1^2)))^2 - (((Y1+Z1)^2 - (Y1^2+Z1^2))^2 \times X1) - (((3 \times X1^2) + (a \times Z1^4))^2 - (4 \times ((X1+Y1^2)^2 - (X1^2+Y1^4))) \times Z1^2)) \times (2 \times (((Y1+Z1)^2 - (Y1^2+Z1^2))^2 \times X1) - (((3 \times X1^2) + (a \times Z1^4))^2 - (4 \times ((X1+Y1^2)^2 - (X1^2+Y1^4))) \times Z1^2)))^2 - (2 \times ((2 \times (((Y1+Z1)^2 - (Y1^2+Z1^2))^2 \times X1) - (((3 \times X1^2) + (a \times Z1^4))^2 - (4 \times ((X1+Y1^2)^2 - (X1^2+Y1^4))) \times Z1^2)))^2 \times (((3 \times X1^2) + (a \times Z1^4))^2 - (4 \times ((X1+Y1^2)^2 - (X1^2+Y1^4))) \times Z1^2))) \times ((Y1 \times (((Y1+Z1)^2 - (Y1^2+Z1^2)) \times ((Y1+Z1)^2 - (Y1^2+Z1^2)^2)) - Z1^2 \times (((2 \times ((X1+Y1^2)^2 - (X1^2+Y1^4))) - ((3 \times X1^2) + (a \times Z1^4))^2 - (4 \times ((X1+Y1^2)^2 - (X1^2+Y1^4))) \times Z1^2))) \times ((3 \times X1^2) + (a \times Z1^4)) - (8 \times Y1^4) \times Z1^2))) - (2 \times (Z1^2 \times (Z1 \times (((2 \times ((X1+Y1^2)^2 - (X1^2+Y1^4))) - ((3 \times X1^2) + (a \times Z1^4))^2 - (4 \times ((X1+Y1^2)^2 - (X1^2+Y1^4))) \times Z1^2))) - (2 \times (Z1^2 \times (Z1 \times (((2 \times ((X1+Y1^2)^2 - (X1^2+Y1^4))) - ((3 \times X1^2) + (a \times Z1^4))^2 - (4 \times ((X1+Y1^2)^2 - (X1^2+Y1^4))) \times Z1^2))) \times ((3 \times X1^2) + (a \times Z1^4)) - (8 \times Y1^4) \times Z1^2))) \times (((Y1+Z1)^2 - (Y1^2+Z1^2))^2 \times X1) - (((3 \times X1^2) + (a \times Z1^4))^2 - (4 \times ((X1+Y1^2)^2 - (X1^2+Y1^4))) \times Z1^2)) \times (2 \times (((Y1+Z1)^2 - (Y1^2+Z1^2))^2 \times X1) - (((3 \times X1^2) + (a \times Z1^4))^2 - (4 \times ((X1+Y1^2)^2 - (X1^2+Y1^4))) \times Z1^2))))))$$

Remarque 28. Une première difficulté pour cette optimisation est de choisir judicieusement la bonne formule à optimiser pour le calcul de $[s]P$. Ces formules peuvent provenir de plusieurs sources : application symbolique d'un algorithme de multiplication scalaire, choix à la main d'une chaîne d'additions, prise en compte de formules déjà connues pour les calculs de $[3]P$ et de $[5]P$, etc. Chacune de ces méthodes présente des avantages et des inconvénients, mais la qualité des formules produites en tant que matière première à l'optimisation est difficile à déterminer. En effet, nous pourrions choisir la formule la plus courte, c'est-à-dire celle qui sera la plus factorisée. D'un point de vue pratique, ce choix permet de minimiser les ressources de calcul et de mémoire nécessaires, mais les factorisations effectuées a priori rendront impossibles de potentielles factorisations plus intéressantes. À l'inverse, choisir la formule la plus développée (généralement celle résultant de l'algorithme *Doublement-et-addition* avec des formules non-simplifiées pour l'addition et le doublement), permet de considérer un plus grand nombre de transformations arithmétiques. En théorie, cette dernière solution est donc à privilégier pour optimisation complète. Nous verrons cependant qu'il est impossible en pratique de prendre en compte toutes les transformations possibles, certaines optimisations reposant sur la résolution de problèmes difficiles. Dans la suite, nous considérons en théorie des formules peu factorisées. En pratique, plusieurs formules pour le calcul d'un même $[s]P$ seront utilisées pour trouver la meilleure optimisation possible.

5.2 Une approche pour l'optimisation automatique

Notre approche d'optimisation d'une formule \mathcal{F} se divise en deux étapes :

- éliminer les sous-expressions communes de \mathcal{F} à l'aide d'une transformation de l'arbre en une structure où chaque sous-arbre est unique ;
- réduction de son coût à l'aide de transformations arithmétiques.

Nous considérons dans la suite une formule \mathcal{F} quelconque représentée par l'arbre $T = (X, E)$.

5.2.1 Élimination des sous-expressions communes

Définition 10. Une sous-expression de \mathcal{F} est un sous-arbre connexe de T enraciné en x . Deux sous-expressions enracinées en x_1 et x_2 sont communes si et seulement si $Cal(x_1) = Cal(x_2)$.

Éliminer les sous-expressions communes revient à factoriser les calculs de manière à réutiliser des résultats obtenus précédemment. Un outil utilisé en compilation pour cette élimination est la structure de DAG.

Définition 11 (DAG). Un arbre orienté sans cycle orienté, appelé DAG (Directed Acyclic Graph) est un arbre dans lequel chaque sous-arbre est unique. Dans notre cas, cela signifie :

$$\forall x_i, x_{i'} \in X, \text{Cal}(x_i) = \text{Cal}(x_{i'}) \implies x_i = x_{i'}$$

La figure 5.2 présente le DAG associé à l'arbre 5.1.

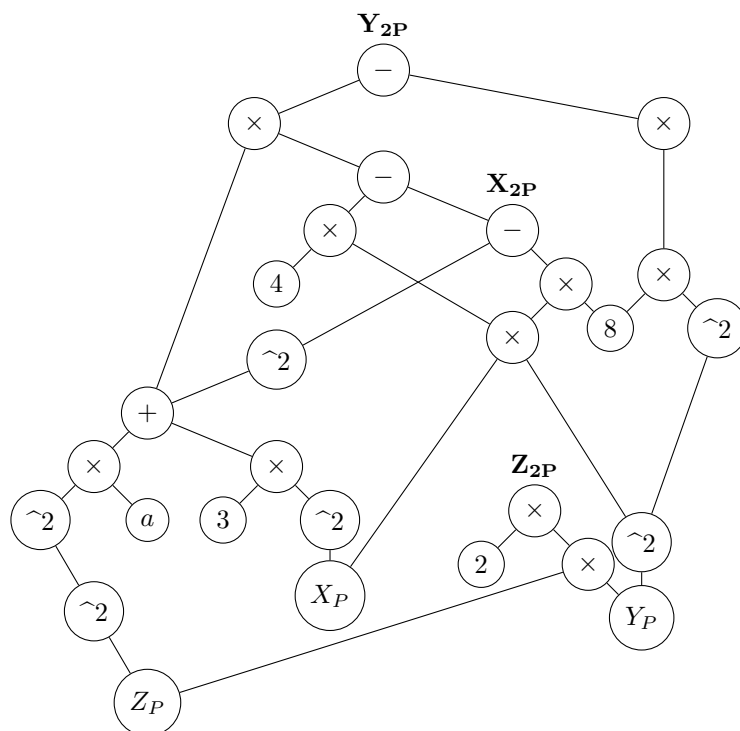


FIGURE 5.2 – DAG de l'arbre 5.1

Le DAG est construit par l'algorithme 14. L'arbre est parcouru en profondeur. Pour chaque nœud x , on vérifie si un sous-arbre équivalent enraciné en x' n'a pas déjà été découvert. Un sous-arbre équivalent est un sous-arbre enraciné en $x' \neq x$ tel que $\text{Cal}(x) = \text{Cal}(x')$. Si x' a été découvert, on supprime x et son parent pointe sur x' . Cela signifie en particulier qu'un nœud peut avoir zéro, un ou deux enfants comme dans la structure classique, mais il peut avoir plusieurs parents. De la racine vers les feuilles, notre DAG reste donc un arbre binaire.

Pour tester si $\text{Cal}(x) = \text{Cal}(x')$, l'algorithme prend en paramètre une fonction de hachage `Hash` qui produit une représentation unique du sous-arbre. Grâce à une table de hachage L dans laquelle les paires clé-valeurs sont $(\text{Hash}(x), x)$, nous identifions facilement les sous-arbres communs. La fonction `Get()` renvoie x si $x \in L$, \emptyset sinon.

Théorème 19. *L'algorithme 14 calcule correctement le DAG de l'arbre en $\mathcal{O}(\#X \log \#X)$.*

Démonstration. L'algorithme effectue un parcours en profondeur. En supposant la fonction `Hash` sans erreurs, tous les sous-arbres communs sont retrouvés. Dans le pire des cas, l'arbre est déjà un DAG. L'algorithme parcourt donc l'ensemble des nœuds de X avec pour chacun une recherche dans la table de hachage. En utilisant une structure ordonnée, cette recherche est réalisée en $\mathcal{O}(\log \#X)$. \square

Le principal problème de notre approche est de trouver une fonction de hachage efficace. Nous abordons cette question dans la section 5.3.

Algorithme 14: DAG

Données : $T = (X, E)$, $n \in X$, L table de hachage, $\text{Hash}()$
Résultat : Le DAG de l'arbre de racine n

- 1 $t \leftarrow \text{Get}(\text{Hash}(n), L)$
- 2 **si** $t \neq \emptyset$ **alors retourner** t
- 3 **si** $fg(n) \neq \emptyset$ **alors** $fg(n) \leftarrow \text{DAG}(fg(n), L, \text{Hash})$
- 4 **si** $fd(n) \neq \emptyset$ **alors** $fd(n) \leftarrow \text{DAG}(fd(n), L, \text{Hash})$
- 5 $L \leftarrow L \cup (\text{Hash}(n), n)$
- 6 **retourner** n

5.2.2 Transformations arithmétiques

Une fois les sous-expressions communes éliminées, on peut appliquer au DAG obtenu des transformations permettant de diminuer le coût arithmétique de la formule. Nous avons établi la relation $M \geq S$. On peut transformer une multiplication en une somme de carrés grâce aux relations suivantes :

$$ab = \frac{1}{2}((a+b)^2 - a^2 - b^2) \quad (5.1)$$

$$ab = \frac{1}{2}(a^2 + b^2 - (a-b)^2) \quad (5.2)$$

$$ab = \frac{1}{4}((a+b)^2 - (a-b)^2) \quad (5.3)$$

On considère généralement $0,67M \leq S \leq 0,8M$. Utiliser ces opérations pour transformer une multiplication quelconque n'est donc pas efficace. En effet, en utilisant 5.3, on transformerait un produit de coût M et deux carrés de coût total $1,54M \leq 2S \leq 1,6M$. Cette méthode sera efficace si un carré (pour l'équation 5.3), ou 2 carrés (pour les équations 5.1 et 5.2) sont nécessaires dans une autre partie du calcul.

Exemple 17. Dans l'exemple 13 qui présente le calcul du doublement en système de coordonnées jacobiennes, le calcul de la troisième coordonnée dans \mathcal{F} est réalisé par : $Z_2 = 2 \times Y_1 \times Z_1$. Or, les quantités Y_1^2 et Z_1^2 sont nécessaires pour le calcul de X_2 et Y_2 . On peut effectuer la transformation

$$Z_2 = 2 \times Y_1 \times Z_1 = 2\left(\frac{1}{2}((Y_1 + Z_1)^2) - Y_1^2 - Z_1^2\right) = (Y_1 + Z_1)^2 - Y_1^2 - Z_1^2$$

qui implique le calcul d'un nouveau carré mais qui élimine la multiplication. De la même manière, le calcul $4X_1Y_1^2$ peut être remplacé par $2((X_1 + Y_1^2)^2 - X_1^2 - Y_1^4)$. Ces deux transformations expliquent le coût réduit de la formule \mathcal{G} .

Remarque 29. Deux carrés étant plus coûteux qu'une multiplication, on transforme $a^2 - b^2$ en $(a-b)(a+b)$ si a^2 et b^2 n'apparaissent nulle part ailleurs dans le calcul.

Remarque 30. Les coefficients rationnels $1/2$ et $1/4$ introduits par les transformations arithmétiques introduisent une inversion dans le corps de définition de la courbe. Cependant, en système de coordonnées projectives, un point d'une courbe est une classe d'équivalence : $(X : Y : Z) = \{(\lambda^c X, \lambda^d Y, \lambda Z) : \lambda \in \mathbb{F}_p\}$. Les coefficients rationnels sont éliminés en multipliant chacune des variables résultats par un λ à la bonne puissance bien choisi.

Lors de la construction du DAG, nous identifions chaque multiplication et chaque carré calculés par la formule. Nous appliquons ensuite les transformations arithmétiques lorsqu'elles sont possibles.

5.2.3 Straight-Line programs

Les formules que nous traitons peuvent s'écrire sous forme de *straight-line programs* qui sont exécutés par des circuits logiques [101, 83]. Un *straight-line program* (SLP) définit un ensemble d'opérations arithmétiques sur un ensemble de variables x_i en un nombre finie d'étapes s . Une étape d'un SLP peut être :

- la lecture d’une variable (**s** READ x_i);
- l’écriture d’un résultat (**s** OUTPUT i) où i représente la i -ème étape du programme;
- une opération arithmétique **OP** entre les résultats de deux étapes précédentes i et j (**s** OP i, j).

Étape	Opération	Variable(s)
1	READ	4
2	READ	X_P
3	READ	Y_P
4	MULT	3, 3
5	MULT	2, 4
6	MULT	1, 5
7	OUTPUT	6

Exemple 18 (SLP du calcul de S dans la formule \mathcal{F} de l’exemple 13).

Le graphe produit par le SLP et représenté sous forme de circuit est un DAG dans le sens où chaque étape agit sur le résultat d’une étape précédente (excepté pour les feuilles). Les polynômes multivariés peuvent être représentés sous forme de SLP. Sous cette forme, on pourra par exemple leur appliquer des algorithmes de factorisation [51] ou de recherche de PGCD [50].

La difficulté de construire un SLP minimal en terme d’opérations arithmétiques à partir d’une formule est cependant la même que celle de construire un DAG minimal : l’élimination des sous-expressions communes nécessite des outils pour comparer deux SLP. L’équivalence de deux SLP est abordée par Ibarra et Moran dans [46], où ils prouvent que le problème d’équivalence de deux SLP pour des variables définies dans un corps fini est NP-Difficile.

En revanche, une fois le DAG minimal calculé, on pourra utiliser le SLP pour appliquer les algorithmes connus de factorisation, ce qui offre une perspective intéressante à nos travaux.

5.3 Une fonction Hash heuristique

Pour produire un DAG, la fonction **Hash** utilisée doit tenir compte des propriétés des opérateurs arithmétiques (associativité, commutativité, distributivité). Il est en effet nécessaire d’avoir $\mathbf{Hash}((ab)c)=\mathbf{Hash}(a(bc))$, $\mathbf{Hash}(ab)=\mathbf{Hash}(ba)$ et $\mathbf{Hash}(a(b+c))=\mathbf{Hash}(ab+ac)$. On définit donc une fonction optimale comme une fonction qui :

- garantit l’unicité de chaque sous-arbre du DAG au sens calculatoire, c’est-à-dire qu’il n’existe pas dans le DAG deux nœuds différents x_1 et x_2 tels que $\mathbf{Cal}(x_1) = \mathbf{Cal}(x_2)$;
- ne produit pas de faux positifs : $\mathbf{Hash}(x_1)=\mathbf{Hash}(x_2) \Leftrightarrow \mathbf{Cal}(x_1) = \mathbf{Cal}(x_2)$.

Grâce à la propriété suivante, nous avons identifié une fonction permettant de produire un DAG optimal.

Propriété 1. Deux polynômes sont égaux si, et seulement si, ils sont de même degré et que les coefficients de leurs monômes de même degré sont égaux.

Or, chaque sous-expression enracinée en x s’écrit comme un polynôme à m variables. Pour garantir les deux conditions définies ci-dessus, il suffit de développer totalement chaque sous-expression et de stocker dans un vecteur de taille suffisamment grande l’ensemble des coefficients de ses monômes.

Remarque 31. Cette méthode tient compte des propriétés arithmétiques des expressions, mais pas des propriétés de la courbe elliptique. En particulier, pour identifier deux représentants d’une même classe d’équivalence (voir §2.4.3), il est nécessaire de transformer les coordonnées projectives en coordonnées affines, où chaque point est représenté par un unique couple (x, y) .

Si cette méthode permet en théorie d’obtenir un DAG minimal, elle pose en pratique un problème de stockage exponentiel dû à la croissance des tailles des formules. Il faudra en effet pour chaque

nœud garder en mémoire les coefficients et les puissances des monômes du polynôme correspondant à la sous-expression. Or, le nombre de nœuds de l'arbre croît exponentiellement comme nous l'avons illustré au tableau 5.2. De plus, la taille des polynômes croît également rapidement. Si l'on calcule $[2]P + P = [3]P$ par exemple, le degré du polynôme est égal à 45. Il dépasse 200 pour le calcul de $[5]P = [3]P + [2]P$.

Nous allons donc présenter une fonction Hash heuristique qui évalue chacune des sous-expressions en un certain nombre de points différents. Le risque de cette approche est l'apparition de faux-positifs : des polynômes différents ayant la même évaluation. La probabilité d'avoir deux évaluations identiques peut être calculée grâce au lemme de Schwartz-Zippel [86, 103].

Définition 12. Le degré d d'un polynôme multivarié est égal au degré de son monôme de plus haut degré, soit celui dont la somme des degrés des variables est maximale.

Lemme 5. (Schwartz-Zippel) Soit \mathcal{P} un polynôme multivarié non nul à m variables et de degré d défini sur un corps fini (ou un anneau intègre) \mathbb{K} . Soit un ensemble $S \subseteq \mathbb{K}$ et c_1, \dots, c_m des valeurs tirées aléatoirement dans S . On a :

$$Pr[\mathcal{P}(c_1, \dots, c_m) = 0] \leq \frac{d}{\#S}.$$

Lorsque l'on veut comparer deux polynômes \mathcal{P}_1 et \mathcal{P}_2 ($\mathcal{P}_1 \neq \mathcal{P}_2$) de degré d , on pose $\mathcal{P} = \mathcal{P}_1 - \mathcal{P}_2$. On a donc

$$Pr[\mathcal{P}(c_1, \dots, c_m) = 0] \leq \frac{d}{\#S}$$

soit :

$$Pr[\mathcal{P}_1(c_1, \dots, c_m) = \mathcal{P}_2(c_1, \dots, c_m)] \leq \frac{d}{\#S}.$$

Ce lemme signifie que l'on peut obtenir une probabilité d'obtention de faux-positifs en fonction du degré maximal du polynôme et de l'ensemble dans lequel nous allons tirer nos valeurs.

Chacune des sous-expressions communes de la formule a potentiellement un degré différent. Étant donné qu'il n'y a pas de division dans nos équations, le degré des polynômes croît des feuilles vers la racine. L'exemple suivant donne la probabilité que les évaluations de deux polynômes soient identiques dans le calcul de $[3]P$.

Exemple 19. Pour la meilleure formule de triplement en coordonnées jacobiniennes connue, le degré du polynôme est égal à 25. En choisissant un ensemble S tel que $\#S \approx 2^{32}$, la probabilité d'une collision est égale à $5,8 \times 10^{-9}$. Si l'on calcule $[3]P = [2]P + P$ en utilisant les opérations d'additions et de doublement optimisées, le degré du polynôme est égal à 45. La probabilité d'une collision est inférieure ou égale à 1×10^{-8} .

Nous avons donc utilisé un ensemble S d'environ 2^{32} éléments (pour éviter des calculs multiprécisions). Il est difficile de poser un résultat général sachant que le degré maximal dépend de la formule utilisée, comme illustré à l'exemple précédent.

En pratique, nous avons choisi d'évaluer chacun de polynômes en 1000 tuples différents. En effet, le degré des polynômes croît rapidement, donc la probabilité $d/\#S$ de générer des faux positifs également. L'évaluation pour 1000 tuples est rapide : il suffit pour chaque nœud interne de l'arbre d'appliquer l'opération arithmétique correspondante entre chaque membre des listes représentant les évaluations de ses fils et de stocker les résultats dans une nouvelle liste. Ainsi, si le stockage a une croissance exponentielle du fait du nombre de termes de la formule, il reste néanmoins constant pour chacun des termes et peut être évalué *a priori*.

Dans tous les cas, une fois la formule optimisée générée, nous produisons un code de validation symbolique permettant de vérifier l'exactitude de son calcul (voir §5.4.4).

L'exemple suivant décrit notre heuristique en utilisant 5 tuples et un ensemble S à 101 éléments. Nous donnons également des valeurs aux constantes non connues des formules. Dans notre cas, le paramètre a de la courbe apparaissant dans la formule de doublement en coordonnées jacobienes.

Exemple 20. Soit l'expression $\mathcal{E} = (3X_1^2 + aZ_1^4) * ([...] - (aZ_1^4 + 3X_1^2)^2) \dots$, qui compte deux variables X_1 et Z_1 et une constante a . La table 5.3 donne les valeurs aléatoires choisies pour a , X_1 et Z_1 , modulo un nombre premier $p = 101$.

a	5	15	34	23	12	10
X_1	34	56	23	72	39	87
Z_1	20	45	98	47	14	68

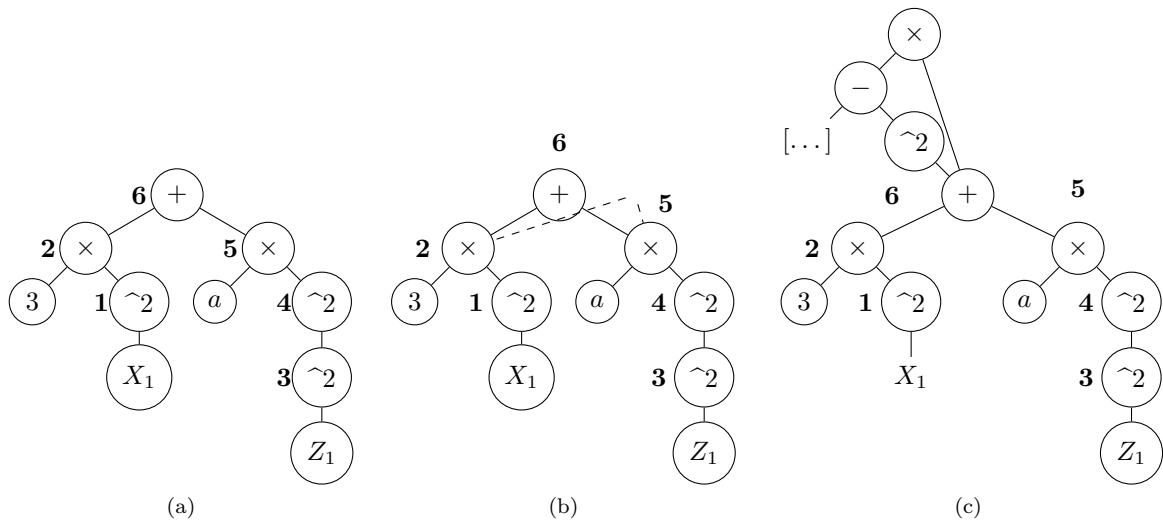
TABLE 5.3 – Valeurs aléatoires pour a , X_1 et Z_1

La formule est parcourue de gauche à droite, en commençant par le terme gauche le plus bas. Ainsi, nous parcourons le nœud 3, puis X_1 , puis X_1^2 , etc. La table 5.4 donne les évaluations de chaque nœud de la sous-expression $(3X_1^2 + aZ_1^4)$. (Les constantes et les variables ne sont pas évaluées car leurs valeurs sont déjà connues).

1	$X_1^2 \bmod p$	45	5	24	33	6	95
2	$3X_1^2 \bmod p$	34	15	72	99	18	83
3	$Z_1^2 \bmod p$	97	5	9	88	95	79
4	$Z_1^4 \bmod p$	16	25	81	68	36	80
5	$aZ_1^4 \bmod p$	80	72	27	49	28	93
6	$(3X_1^2 + aZ_1^4) \bmod p$	13	87	99	47	46	75

TABLE 5.4 – Valeurs des premiers nœuds du DAG

Aucune de ces expressions étant connues, elles sont toutes ajoutées au DAG (figure 5.3a).



Le parcours de la formule continue avec l'évaluation de l'expressions aZ_1^4 , qui est déjà présent dans le DAG, tout comme $3X_1^2$. Comme illustré à la figure 5.3b, ces deux nœuds présents seront partagés. Enfin, nous évaluons l'expression $(aZ_1^4 + 3X_1^2)$ qui donne $(13, 87, 99, 47, 46, 75)$. Cette liste est déjà présente dans le DAG et correspond au nœud 6. La figure 5.3c présente le DAG de l'expression E .

Notre heuristique ne permet cependant pas de calculer les arrangements optimaux.

5.3.1 Le problème *Ensemble Computation*

Si l'heuristique précédente permet en pratique de construire un DAG, celui-ci n'est pas minimal car les règles d'associativité et de commutativité ne sont pas toujours prises en compte. Une exemple simple : si deux sous-expressions sont de la forme $a(bc)$ et $(ab)c$, nous trouvons qu'elles sont communes car leurs évaluations seront égales. Cependant, la formule calculera bc puis $a(bc)$ d'un côté et ab puis $(ab)c$ de l'autre, soit quatre multiplications au lieu de deux. Nous n'avons pas les moyens de réordonner efficacement ces calculs.

Ce problème est appelé *Ensemble Computation* et est NP-complet (voir *Computer and Intractability* [36], p. 66). Nous n'avons pas encore trouvé d'heuristique efficace pour le traiter dans le cas de la simplification de formule, ce qui reste une perspective majeure dans le cadre de ces travaux (même si la situation décrite apparaît peu souvent dans nos formules).

Remarque 32. Le problème *Ensemble Computation* s'applique également sur les chaînes d'additions.

5.4 Implémentation et résultats

Dans cette section, nous présentons l'implémentation des méthodes présentées précédemment ainsi que les résultats obtenus par notre programme en système de coordonnées jacobiennes. Le premier résultat est une formule de quintuplement de coût arithmétique moindre que celles de la littérature. Nous donnons ensuite les coûts arithmétiques de certains cas spéciaux où s est une somme ou un produit de petites puissances.

5.4.1 Implémentation

Nous avons implémenté les méthodes décrites en C++. Le programme prend comme paramètre l'entier s pour lequel on veut calculer une formule $[s]P$ optimisée, ainsi que les fichiers contenant les formules d'addition et de doublement dans le système de coordonnées considéré. Si des formules pour d'autres calculs sont connues, par exemple pour le triplement ou le quintuplement, le programme les prendra en compte. L'utilisateur doit également communiquer au programme comment générer la formule \mathcal{F} qui sera utilisée comme base pour l'optimisation. Il a le choix entre plusieurs types d'algorithmes (*Doublement-et-Addition*, *NAF_w*) ou il peut définir sa propre chaîne d'additions nécessaire au calcul.

À l'aide de la bibliothèque *Boost.Spirit*², le programme construit les arbres de formule pour chacune des opérations qui lui sont passées en paramètre. Ensuite, il construit l'arbre total correspondant au calcul de $[s]P$ et produit le DAG en utilisant la fonction `Hash` heuristique. Enfin, il applique les transformations arithmétiques possibles sur celui-ci.

Le programme produit la nouvelle formule sous forme de fichier texte en pseudo-code où les calculs sont regroupés et factorisés de manière à être facilement lisibles, en prenant comme modèle les formules présentes sur l'EFD³. Il donne également le coût arithmétique de la formule et peut générer du code d'implémentation ou de validation.

5.4.2 Formule de quintuplement

Grâce à notre programme d'optimisation automatique nous obtenons des formules optimisées à partir de la formule de quintuplement de Dimitrov et Mishra [25]. Notre programme a généré la formule

2. <http://boost-spirit.com/home/>

3. Explicit Formula Database, <http://www.hyperelliptic.org/EFD/index.html>

suivante :

$$\begin{aligned}
 XX &= X_P^2; \\
 YY &= Y_P^2; \\
 YYYY &= YY^2; \\
 ZZ &= Z_P^2; \\
 ZZZZ &= ZZ^2; \\
 A &= ((3 \times XX) + (a \times ZZZZ)); \\
 B &= 6 \times ((X_P + YY)^2 - (XX + YYYY)) - A^2; \\
 C &= \frac{1}{2} \times ((B + A)^2 - (B^2 + A^2)) - 8 \times YYYY; \\
 D &= 32 \times C \times YYYY - B \times B^2; \\
 E &= \frac{1}{2}(D - 4 \times C^2) \times ((C + B)^2 - (C^2 + B^2)); \\
 F &= 6 \times ((C^2 + D)^2 - (C^4 + D^2)) - D^2 - 16 \times C^4; \\
 G &= F \times B \times B^2 - 512 \times C^4 \times C \times YYYY; \\
 X_5 &= D^2 \times X_P - 8 \times YY \times E; \\
 Y_5 &= G \times Y_P; \\
 Z_5 &= \frac{1}{2} \times ((D + Z_P)^2 - (D^2 + ZZ));
 \end{aligned}$$

Cette formule a un coût de 8 multiplications et 15 carrés, ce qui est meilleur que celle de Dimitrov et Mishra [25] et de Longa et Gebotys [59] (9 multiplications et 15 carrés), qui ont été produites à la main.

Comme nous l'avons déjà vu, il existe en coordonnées projectives une classe d'équivalence entre un point $P = (X_P : Y_P : Z_P)$ et l'ensemble des points $Q = (X_Q : Y_Q : Z_Q)$ tels que $X_Q = \lambda^c X_P$, $Y_Q = \lambda^d Y_P$ et $Z_Q = \lambda Z_P$. En système jacobien, $c = 2$ et $d = 3$. La multiplication par le rationnel $\frac{1}{2}$ peut ainsi être évitée en multipliant X_5 par 4, Y_5 par 8 et Z_5 par 2 (ce qui correspond à $\lambda = 2$). La formule ainsi transformée devient :

$$\begin{aligned}
 XX &= X_P^2; \\
 YY &= Y_P^2; \\
 YYYY &= YY^2; \\
 ZZ &= Z_P^2; \\
 ZZZZ &= ZZ^2; \\
 A &= ((3 \times XX) + (a \times ZZZZ)); \\
 B &= 6 \times ((X_P + YY)^2 - (XX + YYYY)) - A^2; \\
 C &= (B + A)^2 - (B^2 + A^2) - 16 \times YYYY; \\
 D &= 16 \times C \times YYYY - B \times B^2; \\
 E &= (D - C^2) \times ((C + 2 \times B)^2 - (C^2 + 4 \times B^2)); \tag{5.4} \\
 F &= 3 \times ((C^2 + 4 \times D)^2 - (C^4 + 16 \times D^2)) - 8 \times D^2 - 8 \times C^4; \tag{5.5} \\
 G &= F \times B \times B^2 - 128 \times C^4 \times C \times YYYY;
 \end{aligned}$$

$$\begin{aligned}
 X_5 &= 4 \times (D^2 \times X_P - YY \times E); \tag{5.6} \\
 Y_5 &= G \times Y_P; \\
 Z_5 &= (D + Z_P)^2 - (D^2 + ZZ);
 \end{aligned}$$

Il reste une dernière optimisation, que nous devons faire à la main, et qui montre les limites de l'automatisation des optimisations. Nous avons choisi des heuristiques qui nous permettent d'automatiser certaines simplifications, mais pour être complet, il faudrait tenir compte de toutes les transformations ce qui est difficile au vu de la combinatoire exponentielle des possibilités. Pour effectuer cette

simplification, nous devons revenir un peu en arrière. L'équation 5.4 non transformée est celle-ci :

$$E = 4 \times (D - C^2) \times C \times B; \quad (5.7)$$

La variable E sert au calcul de X_5 dans l'équation 5.6, si on la remplace dans l'équation, on trouve :

$$X_5 = 4 \times (D^2 \times X_P - YY \times \underbrace{4 \times (D - C^2) \times C \times B}_E);$$

Nous réorganisons la chaîne de multiplications $YY \times E$ en $4 \times YY \times C \times B \times (D - C^2)$. On peut transformer $2 \times YY \times C$ en $(YY + C)^2 - YYY - CC$. La multiplication $2 \times B \times (D - C^2)$ peut être transformée en $(B + (D - C^2))^2 - B^2 - (D - C^2)^2$, mais il nous manque deux carrés : $(B + (D - C^2))^2$ et $(D - C^2)^2$.

Le calcul de F à l'équation 5.5 peut être transformé. Si l'on revient à la multiplication qui a été simplifiée, l'équation 5.5 devient :

$$F = 8 \times [3 \times (C^2 \times D) - C^4 - D^2]$$

or,

$$\begin{aligned} (D - C^2)^2 = D^2 + C^4 - 2 \times D \times C^2 &\Leftrightarrow (D - C^2)^2 + 2 \times D \times C^2 = D^2 + C^4 \\ &\Leftrightarrow -(D - C^2)^2 - 2 \times D \times C^2 = -D^2 - C^4 \end{aligned}$$

donc,

$$\begin{aligned} F &= 8 \times [3 \times (C^2 \times D) - (D - C^2)^2 - 2DC^2] \\ &= 8 \times [D \times C^2 - (D - C^2)^2] \\ &= 8 \times \left[\frac{1}{2} \times ((-1) \times (D - C^2)^2 + D^2 + C^4) - (D - C^2)^2 \right] \\ &= 4 \times [D^2 + C^4 - 3 \times (D - C^2)^2] \end{aligned}$$

On calcule donc $(D - C^2)^2$, qui servira alors pour deux multiplications.

Avec cette optimisation, nous pouvons alors écrire la meilleure formule connue à ce jour pour calculer $[5]P$.

$$\begin{aligned} A &= 3 \times X_P^2 + a \times Z_P^4 \\ B &= 6 \times ((X_P + Y_P^2)^2 - X_P^2 - Y_P^4) - A^2 \\ C &= (A + B)^2 - A^2 - B^2 - 16 \times Y_P^4 \\ D &= 4 \times Y_P^4 \times C \\ E &= 4 \times D - B \times B^2 \\ F &= (E - C^2) \\ G &= ((B + F)^2 - B^2 - F^2) \\ X_5 &= 4 \times (X_P \times E^2 - ((Y_P^2 + C)^2 - Y_P^4 - C^2) \times G) \\ Y_5 &= 4 \times Y_P \times (B \times B^2 \times (E^2 + C^4 - 3 \times F^2) - 8 \times D \times C^4) \\ Z_5 &= (Z_P + E)^2 - Z_P^2 - E^2 \end{aligned}$$

Sa validité a été testée avec Magma. Elle nécessite 7 multiplications, 16 carrés, 1 multiplication par la constante a de l'équation de Weierstrass, 24 additions ou soustractions, 2 multiplications par 16, 2 multiplications par 3, 1 multiplication par 32, 1 multiplication par 4 et 1 multiplication par 6.

Notre programme retourne des formules optimisées, mais en utilisant des heuristiques. Pour effectuer notre dernière optimisation à la main, nous nous sommes servis de plusieurs propriétés mathématiques. Nous pourrions automatiser ces simplifications, mais il faudrait les appliquer à toutes les expressions, sans savoir si cela pourra être utile ou non. La taille de la structure explosera exponentiellement, car il faudra garder dans des classes d'équivalences les possibilités pour chaque opération puis choisir le meilleur chemin.

5.4.3 Cas spéciaux

Avoir des formules optimisées pour certains s *spéciaux* peut s'avérer intéressant : si $s = p^i q^j$ par exemple, les formules pourront être utilisées pour la multiplication scalaire utilisant la représentation double base.

Le tableau 5.5 présente les résultats expérimentaux pour certains cas spéciaux en coordonnées jacobiniennes. Les formules pour calculer $[2]P$ et $[3]P$ sont les meilleures formules connues, extraites de l'*Explicit Formula Database*⁴, et $[5]P$ est notre formule de quintuplement.

$[s]P$	Littérature	Notre programme
$2P$	$1M + 8S$	–
$3P$	$5M + 10S$	–
$5P$	$15M + 10S$ [25] $9M + 15S$ [59]	$7M+16S$
$2^i P \pm 2^j P(*)$	$(i + 11)M + (8i + 4)S$	$(i + 9)M + (8i + 6)S$
$3^i P \pm 3^j P(*)$	$(5i + 11)M + (10i + 4)S$	$(5i + 9)M + (10i + 6)S$
$5^i P \pm 5^j P(*)$	$(15i + 11)M + (10i + 4)S$	$(7i + 9)M + (16i + 6)S$
$2^i 3^j P \pm 1$	–	$(i + 5j + 9)M + (8i + 10j + 6)S$
$2^i 5^j$	–	$(i + 7j)M + (8i + 16j)$
$2^i 5^j \pm 1$	–	$(i + 7j + 9)M + (8i + 16j + 6)$
$3^i 5^j$	–	$(5i + 7j)M + (10i + 16j)$
$3^i 5^j \pm 1$	–	$(5i + 7j + 9)M + (10i + 16j + 6)$

TABLE 5.5 – Coût arithmétique pour le calcul de $[s]P$ en coordonnées Jacobiennes (*) avec $0 \leq j < i$

5.4.4 Production de code

Pour garantir la fiabilité du résultat, le programme produit un code de vérification Magma qui permet une certification de la formule obtenue par un calcul symbolique. La première partie des scripts calcule $[s]P$ en utilisant l'algorithme *Doublement-et-Addition*. La seconde partie des scripts contient la nouvelle formule de calcul. L'exemple 21 présente le code de validation produit par le programme pour notre nouvelle formule de quintuplement.

Exemple 21 (Code de vérification Magma pour le quintuplement).

```
K<a,b,X1,Y1>:=FieldOfFractions(PolynomialRing(Rationals(),4));
R<Z1>:=PolynomialRing(K,1);
S:=quo<R|Y1^2-X1^3-a*X1*Z1^4-b*Z1^6>;
x1:=X1/Z1^2; y1:=Y1/Z1^3;
S!(y1^2-x1^3-a*x1-b);
l:=(3*x1^2+a)/(2*y1);
x2:=\1^2-x1-x1; y2:=1*(x1-x2)-y1;
S!(y2^2-x2^3-a*x2-b);
l:=(3*x2^2+a)/(2*y2);
x4:=1^2-x2-x2; y4:=1*(x2-x4)-y2;
S!(y4^2-x4^3-a*x4-b);
l:=(y4-y1)/(x4-x1);
x5:=1^2-x1-x4; y5:=1*(x1-x5)-y1;
S!(y5^2-x5^3-a*x5-b);
AB := (16*((Y1)^2)^2);
AC := (a*((Z1)^2)^2);
```

4. <http://www.hyperelliptic.org/EFD/>

```

AD := ((3*(X1)^2)+AC);
AE := (X1+(Y1)^2)^2;
AF := ((AE-(X1)^2)-((Y1)^2)^2);
AG := ((6*AF)-(AD)^2);
AH := ((AD+AG)^2-(AD)^2);
AI := ((AH-(AG)^2)-AB);
AJ := (((Y1)^2)^2*AI);
AK := ((4*(4*AJ))-(AG*(AG)^2));
AL := (AK-(AI)^2)^2;
AM := (AG+(AK-(AI)^2));
AN := ((AM)^2-(AG)^2);
AO := ((Y1)^2+AI)^2;
AP := (AO-((Y1)^2)^2);
AQ := ((AP-(AI)^2)*(AN-AL));
AR := (X1*(AK)^2)-AQ;
AS := (((AI)^2)^2*(4*AJ));
AT := ((AK)^2+((AI)^2)^2);
AU := ((AG*(AG)^2)*(AT-(3*AL)));
AV := (Y1*(AU-(8*AS)));
AW := ((Z1+AK)^2-(Z1)^2);
X5 := (4*AR);
Y5 := (4*AV);
Z5 := (AW-(AK)^2);
S!(x5-X5/Z5^2); S!(y5-Y5/Z5^3);

```

Le premier script calcule l'ensemble symbolique des points $[5]P$ pour une courbe arbitraire en système de coordonnées jacobiniennes. Il calcule ensuite les points $[5]P$ avec notre nouvelle formule et les deux ensembles sont comparés par la dernière instruction. Si le résultat du script est $(0,0)$, les deux ensembles sont égaux et la formule est correcte. Dans le cas contraire, la formule est fautive.

Nous produisons également du code C++ pour la bibliothèque PACE (voir annexe A) et nous pouvons facilement générer du code pour d'autres langages ou programmes mathématiques. L'avantage de cette production de code est de faciliter l'implémentation *en dur* des multiplications scalaires sur les courbes elliptiques pour de petits scalaires. Pour l'instant, notre programme optimise uniquement les calculs. Nous verrons dans les perspectives que des optimisations à d'autres niveaux sont possibles et souhaitables : optimisation de l'utilisation de la mémoire en minimisant les variables temporaires nécessaires (ce qui peut avoir pour effet d'augmenter les calculs), ou optimisation pour produire du code parallèle.

5.5 Conclusion et perspectives

Le problème de l'optimisation de formules est de réduire le nombre d'opérations coûteuses dans le corps de définition de la courbe elliptique pour le calcul de $[s]P$.

À la main, cette optimisation est quasiment impossible pour $s > 3$ sans pré-traitement. Notre méthode automatise cette optimisation en utilisant une heuristique simple et efficace, qui permet de réduire le nombre de multiplications modulaires en éliminant les sous-expressions communes puis en transformant certaines de ces multiplications en expressions arithmétiques moins coûteuses. Grâce à cette méthode, nous sommes capables de produire des formules optimisées pour calculer $[s]P$ pour une large gamme de petits scalaires s . Nous avons présenté ici l'optimisation de formules en coordonnées jacobiniennes, mais la méthode fonctionne pour tous les systèmes de coordonnées car elle s'abstrait totalement de la structure considérée, à savoir la courbe elliptique.

Un des avantages de cette automatisation de l'optimisation est de produire à la fois du code de vérification symbolique et du code efficace pour des bibliothèques mathématiques (en particulier pour PACE). Plusieurs perspectives possibles sont à considérer pour ces travaux.

Tout d'abord dans la forme. Notre méthode se divise en deux grandes étapes qui sont l'élimination de sous-expressions communes et la réduction du coût à l'aide de certaines transformations classiques. La partie élimination de sous-expressions communes a été traitée à l'aide d'un outil classique de la compilation et d'une heuristique que nous avons définie. Trouver une heuristique pour résoudre de manière fiable le problème *Ensemble Computation* est nécessaire pour éliminer les multiplications redondantes dans le calcul.

La réduction du coût arithmétique de la formule est réalisée à l'aide de transformations arithmétiques simples. Or nous avons vu au paragraphe 5.4.2 que d'autres transformations, moins évidentes à identifier, permettent d'optimiser un peu plus les formules. Dans cette optique, une étude des motifs récurrents transformables dans les formules permettrait de définir de nouvelles transformations ciblées. Il semble en effet difficile de considérer l'ensemble des transformations arithmétiques, sans explosion exponentielle de la mémoire (pour garder les équivalences de chaque sous-expression) ou du temps de calcul (recalculer toutes les équivalences à la volée). Il existe cependant une transformation simple à prendre en compte : considérer l'équation de la courbe dans les transformations. Avec l'équation simplifiée de Weierstrass, on peut par exemple transformer x^3 en $y^2 - ax - b$ et gagner ainsi sur le degré du polynôme.

On pourra également utiliser les outils de factorisation utilisés pour la simplification des programmes *straight-line*, en gardant en mémoire que la complexité des algorithmes croît en fonction de la taille des programmes.

Sur le fond, nous distinguons plusieurs perspectives directement liées à ces travaux. Tout d'abord, nous avons uniquement considéré l'aspect arithmétique du problème. Or ces formules ont pour vocation à servir dans le calcul de la multiplication scalaire dans le cadre de la cryptographie basée sur les courbes elliptiques. Pour contrer les attaques dites par canaux cachés, dans lesquelles un attaquant peut récupérer des informations sur la clé secrète en utilisant des mesures physiques (consommation de courant, émissions magnétiques, etc.), on utilise généralement des formules unifiées pour le doublement et l'addition. On pourrait lisser les coûts des formules de calcul de $[s_i]P$ pour un ensemble de scalaires s_i proches.

Dans ces travaux, nous avons considéré uniquement l'aspect temps de calcul et négligé l'aspect mémoire, c'est-à-dire que l'on utilise autant de registres pour stocker les variables intermédiaires que l'on veut. Ce choix est largement justifié pour les architectures grand public et les serveurs, qui disposent de zones mémoires suffisamment importantes. Cependant, l'aspect mémoire est important sur les architectures embarquées ou encore les cartes graphiques. Une extension possible de ces travaux et de focaliser l'optimisation sur la mémoire, quitte à augmenter le nombre de calculs.

Nous avons principalement travaillé sur les courbes elliptiques définies sur un corps premier. Une étude similaire pour les courbes définies sur une extension de corps binaire, généralement utilisées pour les coprocesseurs dédiés à la cryptographie ou les circuits intégrés programmables, permettrait de fournir des formules optimisées qui offriraient des gains de temps non-négligeables.

Ces travaux ont été réalisés en début de thèse. Avec le recul, proposer un outil pour produire des formules parallèles pour le calcul de $[s]P$ aurait un intérêt majeur. En effet, nous avons présenté au chapitre 3 une parallélisation possible de la multiplication scalaire dans le cas où le point P n'est pas connu en amont du calcul et qui permet des gains de performances bornés. Paralléliser les opérations dans le corps fini ou les opérations multiprécision pour les tailles considérées pour la cryptographie sur les courbes elliptiques est inefficace dans le modèle de parallélisme synchrone en raison de l'*overhead* lié au parallélisme. En se plaçant entre ces deux niveaux, paralléliser les opérations dans le groupe des points de la courbe permettra sans doute des gains de performances honorables, en particulier en utilisant un parallélisme asynchrone dans lequel les tâches pourront être lancées et synchronisées localement, en utilisant en particulier les DAG créés pour les formules pour l'ordonnancement des tâches.

Nous nous sommes intéressés dans ce manuscrit aux performances de certains opérateurs de niveaux arithmétiques différents utilisés en cryptographie asymétrique. Pour chacun de ces opérateurs, nous avons défini une ou plusieurs méthodes permettant d'accélérer leur temps de calcul en utilisant le parallélisme.

Sur architecture SMP, nous avons proposé une première implémentation parallèle de la multiplication entière utilisant un schéma quadratique. Les performances obtenues par cet opérateur ont mis en lumière l'influence du surcoût lié au parallélisme logiciel pour les tailles visées. Elles ont également montré que la parallélisation à ce niveau arithmétique peut bénéficier à des calculs entre entiers de plus de 2000 bits. Grâce à cette multiplication, nous avons pu définir simplement des opérateurs de multiplication modulaire parallèle qui utilisent les algorithmes classiques de Barrett et de Montgomery. La méthode de multiplication bipartite, déjà parallèle, peut également tirer parti de cette implémentation.

Nous avons également proposé deux parallélisations pour la multiplication scalaire sur les courbes elliptiques. La première, qui donne des performances intéressantes nécessite les pré-calculs de multiplications scalaires d'un point de base par une puissance de 2 et sera donc utilisée lorsque le point est connu. La seconde parallélisation ne nécessite pas de pré-calcul. Les performances obtenues en pratique atteignent les performances théoriques qui estiment une multiplication par un scalaire de l bits dans le même temps que l doublements.

Nous avons également défini une méthode permettant de calculer la multiplication modulaire de manière totalement parallèle. Nous avons étudié les différents ordonnancements de ses calculs pour obtenir les meilleures performances. Nous avons remarqué que pour une large plage d'entiers de taille cryptographique, ce nouvel opérateur est plus performant que les versions parallèles des algorithmes de la littérature.

Les suites de ces travaux sont nombreuses, mais on peut insister sur quatre points en particulier. Au niveau de l'arithmétique entière, il est important d'implémenter une version parallèle des algorithmes de multiplication utilisant la FFT. On pourra notamment s'inspirer de la bibliothèque FFTW⁵ pour une implémentation parallèle du calcul de la DFT. Au niveau de l'arithmétique modulaire, la réduction multipartite devrait *a priori* donner des gains de performances intéressants. Couplée avec une multiplication entière parallèle efficace, il sera alors possible de concurrencer la méthode bipartite parallèle sur l'ensemble des tailles des opérandes. Enfin, au niveau de l'arithmétique des courbes, paralléliser les opérations du groupe des points de la courbe pourrait permettre de pallier le manque d'efficacité des algorithmes de multiplication scalaire sans pré-calcul. Pour ceux-ci, notre recherche devra s'orienter vers la découverte d'endomorphismes efficaces pour le parallélisme.

Sur les processeurs graphiques, nous avons proposé plusieurs méthodes pour la représentation des entiers multiprécision ainsi qu'une implémentation de la multiplication de Montgomery et des opérations sur les courbes elliptiques. Les performances obtenues sont inférieures aux performances théoriques en

5. <http://www.fftw.org/>

raison des problèmes posés par l'architecture : accès lents aux données, unités de calcul moins rapides, etc.

Pour pouvoir tenir compte de l'évolution rapide de ces architectures, une bibliothèque d'arithmétique multiprécision sur carte graphique devrait être écrite dans un langage qui s'abstrait de la plate-forme considérée, comme OpenCL. Cependant, cette abstraction se fera sans doute au détriment des performances.

Au vu de l'ensemble des résultats que nous avons obtenus, il est clair que l'arithmétique peut tirer parti des architectures parallèles. Il faut cependant prendre en compte le surcoût lié au parallélisme logiciel et définir les opérateurs en fonction du nombre d'unités de calcul disponibles. Il est plus difficile de répondre à la question du niveau arithmétique auquel il faut paralléliser les calculs. En réalité, il est clair que cela dépendra de l'application visée. En effet, un parallélisme de l'arithmétique entière pour les protocoles basés sur les courbes elliptiques sera par exemple totalement inefficace, car les tailles des paramètres sont trop petites. De manière générale, on peut néanmoins estimer que le parallélisme sera le plus efficace au niveau arithmétique le plus haut, ou en définissant soigneusement des opérateurs utilisant plusieurs niveaux de parallélisme. Par exemple, les protocoles utilisant l'exponentiation modulaire gagneront en performances en utilisant la version double parallélisme de notre multiplication multipartite.

Paralléliser au niveau arithmétique le plus haut est en particulier primordial pour les calculs parallèles sur carte graphique. En effet, nous avons obtenu d'excellentes performances pour la parallélisation de l'application requête par fredonnement. Ces performances ne sont pas retrouvées dans la parallélisation des opérations arithmétiques, car l'intensité des calculs ne masque pas les défauts majeurs des processeurs graphiques : coût élevé d'accès aux mémoires, coût des communications entre la carte et la mémoire centrale de la machine hôte et enfin unités de calcul moins performantes que celles présentes dans les processeurs classiques.

Dans le dernier chapitre, nous avons proposé une méthode pour optimiser automatiquement la multiplication par un petit scalaire sur les courbes elliptiques. Grâce à cette méthode, nous avons pu proposer de nouvelles formules pour un ensemble de scalaires différents en coordonnées jacobiennes. Les avantages de cette méthode sont la généralité de l'approche qui reste valide quelque soit le système de représentation et la production de code : les opérateurs obtenus peuvent s'intégrer directement dans les bibliothèques d'arithmétique des courbes elliptiques existantes.

La suite de ces travaux devra s'orienter vers la production automatique de formules parallèles.

ANNEXE A

PACE : PROTOTYPING ARITHMETIC FOR CRYPTOGRAPHY EASILY

La bibliothèque C++ PACE [47] a pour vocation le prototypage d'opérations arithmétiques pour la cryptographie. Elle comporte trois niveaux arithmétiques différents et hiérarchisés proposant chacun des objets mathématiques génériques. La figure A.1 présente son diagramme de classes simplifié.

Grâce à la généricité et au polymorphisme offerts par le langage C++, chaque objet mathématique pourra être spécialisé en fonction d'un ou plusieurs paramètres. On pourra en particulier définir facilement des opérateurs arithmétiques différents à la construction de l'objet pour tester leurs performances.

A.1 Entiers multiprécision

La classe `integer` est une interface permettant de manipuler facilement des entiers de taille N bits fixée. Cette classe peut utiliser différentes représentations pour stocker les entiers (à ce jour GMP pour des tailles quelconques et mpFq pour des « petites tailles »). Cette flexibilité permet d'utiliser les meilleures arithmétiques *bas niveau* disponibles, mais également d'intégrer facilement d'autres systèmes de représentation. On pourra par exemple intégrer les représentations RNS.

Si par défaut les opérations arithmétiques utilisent les algorithmes des bibliothèques utilisées (GMP, mpFq), le but final de cette bibliothèque est de permettre l'interchangeabilité du code. On va ainsi permettre d'instancier les objets de ce type en passant comme paramètre générique les opérations arithmétiques que l'on souhaite personnaliser.

Nous avons principalement utilisé la bibliothèque GMP dans nos implémentations. Cette bibliothèque représente classiquement les entiers sous forme d'un tableau d'entiers dont on fixe la taille. Les opérations de décalages ou de réduction modulo une puissance de la base sont réalisées par des déplacements de pointeurs ou des changements de taille de tableau, évitant ainsi des copies mémoire inutiles.

A.2 Arithmétique modulaire

La classe `gfp` permet la manipulation d'entiers modulo un `integer P` fixé comme paramètre générique. La classe surcharge les opérateurs arithmétiques pour intégrer les réductions modulaires. Passer en paramètre le type de multiplication utilisée permet une fois encore d'intégrer et d'utiliser simplement plusieurs opérations modulaires sans avoir à modifier le code. Il suffit que la classe de multiplication passée en paramètre propose une fonction de multiplication.

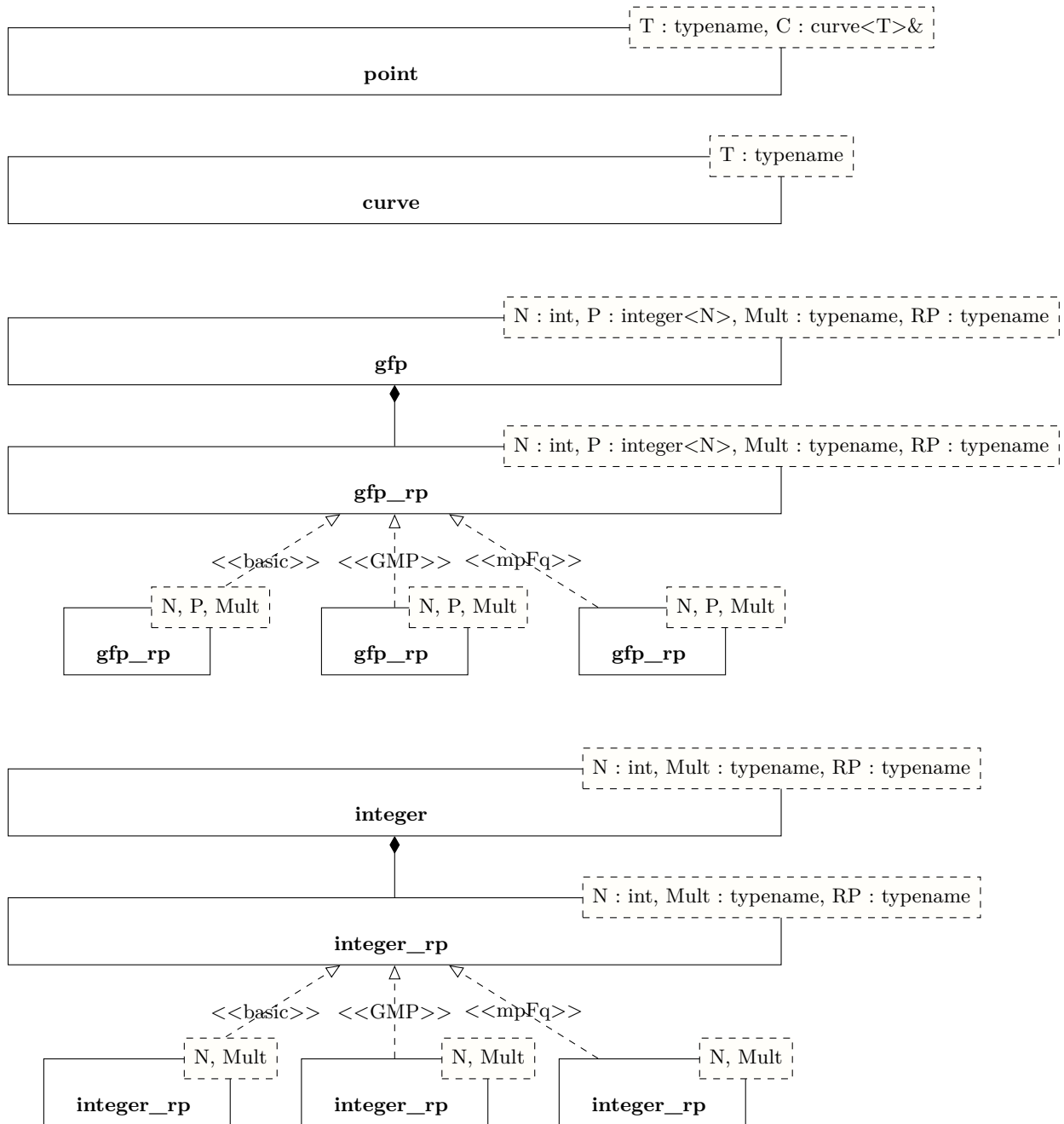


FIGURE A.1 – Hiérarchie de la bibliothèque PACE

Pour gérer au mieux la mémoire, chaque objet `gfp` contient une instance `integer` dans une représentation choisie. Chaque classe de multiplication calcule une et une seule fois les éventuelles constantes nécessitées par les algorithmes (μ et ν pour les algorithmes de Montgomery et de Barrett par exemple). Chaque classe embarque également une zone mémoire de taille prédéfinie permettant de stocker les variables temporaires nécessaires aux calculs. Dans le cas des algorithmes de Montgomery et de Barrett par exemple, il faudra une zone pour stocker le résultat de la multiplication ab , une seconde pour stocker q et une dernière pour stocker qp . L'objectif est de limiter autant que faire se peut les objets temporaires créés dans PACE.

A.3 Courbes elliptiques

Une courbe elliptique est un objet de type `curve`. Le type du corps de définition est défini comme un paramètre générique à la classe. Pour les corps fini \mathbb{F}_p , on utilisera un `gfp<N,p>`. Chaque objet de cette classe contient les paramètres a et b de la courbe. La classe `point` représente un point de la courbe elliptique passé comme paramètre générique. Chaque point contient un ensemble d'objets du même type que le corps de définition de la courbe représentant ses coordonnées dans le système utilisé.

A.4 Algorithmes de multiplication scalaire ou d'exponentiation modulaire

Les algorithmes d'exponentiation modulaire ou de multiplication scalaire sont génériques, dans le sens où seul le type des objets sur lesquels l'exponentiation va être réalisée est nécessaire, avec comme condition la définition de d'au moins deux opérations de bases : l'addition (ou la multiplication) et le doublement (ou le carré). De la même manière, les éventuels précalculs nécessaires sont passés en paramètre par des tables de hachages de type `map<int,T>` ou `T` désigne le type des objets sur lesquels l'exponentiation va être réalisée.

ANNEXE B

EXEMPLE DE CODE SPÉCIALISÉ : MULTIPARTITE $K = 4$ SUR 4 PROCESSEURS

Nous donnons ici un exemple de code pour la multiplication multipartite $k = 4$ sur 4 processeurs en utilisant la bibliothèque GMP. Chacune des quatre tâches est implémentée explicitement dans une classe dérivée de la classe suivante. Les types de données et les opérateurs utilisés sont ceux de la bibliothèque GMP.

```
class Thread_Multipartite_4_4 {
public:
    // _n : size in words of the operands
    unsigned int _n, _l, _s;
    // p : modulus, cst : constant of Montgomery or Barrett
    const mp_limb_t *_p, *_cst;
    Thread_Multipartite_4_4(unsigned int n,
                           const mp_limb_t* p,
                           const mp_limb_t* cst)
    : _n(n), _l(n >> 1), _s(n >> 2), _p(p), _cst(cst) {}

    virtual void multiplication(mp_limb_t* C,
                               const mp_limb_t* A,
                               const mp_limb_t* B) = 0;
    // Registers for the multipartite multiplication
    static mp_limb_t *_r00, *_r01, *_r10, *_r11, *_r12,
                  *_r20, *_r21, *_r22, *_r30, *_r31, *_qp;
};
```

Implémentation de la tâche 0 :

```
class Thread_Multipartite_4_4_0 : public Thread_Multipartite_4_4 {
public:
    Thread_Multipartite_4_4_0(unsigned int n,
                              const mp_limb_t* p,
                              const mp_limb_t* cst)
    : Thread_Multipartite_4_4(n, p, cst) {}

    void multiplication(mp_limb_t* C, const mp_limb_t* A,
                      const mp_limb_t* B) {
        // A0B0 A1B1 A3B1
        mpn_mul_n(_r00, A, B, _s);
        mpn_mul_n(&_r00[_l], &A[_s], &B[_s], _s);
    }
};
```

```

    mpn_mul_n(&_r00[_n], &A[_l+_s], &B[_s], _s);
    mpn_mul_n(_r01, _r00, _cst, _l);
#pragma omp barrier
    //Add the partial quotients
    mpn_add_n(&_r01[_s], &_r01[_s], _r12, _s);
#pragma omp barrier
    //Q0P0
    mpn_mul_n(_qp, _r01, _p, _l);
    mpn_neg(_qp, _qp, _n);
#pragma omp barrier
    mpn_sub(&_qp[_l], &_qp[_l], _n+_l, _r12, _n);
    mpn_add(&_qp[_l], &_qp[_l], _n+_l, _r22, _n);
}
};

```

Implémentation de la tâche 1 :

```

class Thread_Multipartite_4_4_1 : public Thread_Multipartite_4_4 {
public:
    Thread_Multipartite_4_4_1(unsigned int n,
                              const mp_limb_t* p,
                              const mp_limb_t* cst)
: Thread_Multipartite_4_4(n, p, cst) {}
    void multiplication(mp_limb_t* C,
                      const mp_limb_t* A,
                      const mp_limb_t* B) {
        //A1B0 A0B1 A2B1 A1B2 A0B2
        mpn_mul_n(_r10, &A[_s], B, _s);
        mpn_mul_n(&_r10[_l], &A[_s], &B[_l], _s);
        mpn_mul_n(_r11, A, &B[_s], _s);
        mpn_mul_n(&_r11[_l], &B[_s], &A[_l], _s);
        _r10[_n] = mpn_add_n(_r10, _r10, _r11, _n);
        mpn_mul_n(_r12, _cst, _r10, _s);
        mpn_mul_n(_r11, A, &B[_l], _s);
        mpn_add(&_r10[_s], &_r10[_s], _l+_s, _r11, _l);
#pragma omp barrier
        mpn_add(&_r00[_l], _r30, _n+1, &_r00[_l], _n);
#pragma omp barrier
        //Q0P1
        mpn_mul_n(_r12, _r01, _p, _l);
#pragma omp barrier
        mpn_add(&_r00[_s], &_r00[_s], _n+_s+( _r00[_n+_l] ? 1 : 0),
              _r10, _n+_s+1);
    }
};

```

Implémentation de la tâche 2 :

```

class Thread_Multipartite_4_4_2 : public Thread_Multipartite_4_4 {
public:
    Thread_Multipartite_4_4_2(unsigned int n,
                              const mp_limb_t* p,
                              const mp_limb_t* cst)
: Thread_Multipartite_4_4(n, p, cst) {}

    void multiplication(mp_limb_t* C,
                      const mp_limb_t* A,
                      const mp_limb_t* B) {

```

```

//A3B2 A2B3 A3B0 A0B3 A1B3
mpn_mul_n(&_r20[_l], &A[_l+_s], &B[_l], _s);
mpn_mul_n(&_r21[_l], &B[_l+_s], &A[_l], _s);
_r20[_n] = mpn_add_n(&_r20[_l], &_r20[_l], &_r21[_l], _l);
mpn_mul(_r22, _cst, _s+1, &_r20[_l+_s], _s+(_r20[_n] ? 1 : 0));
mpn_mul_n(_r20, A, &B[_l+_s], _s);
mpn_mul_n(_r21, &A[_l+_s], B, _s);
_r20[_n] += mpn_add(_r20, _r20, _n+(_r20[_n] ? 1 : 0), _r21, _l);
mpn_mul_n(_r21, &A[_s], &B[_l+_s], _s);
_r20[_n] += mpn_add(&_r20[_s], &_r20[_s], _l+_s+(_r20[_n] ? 1 : 0),
_r21, _l);
#pragma omp barrier
//Add the partial quotients
mpn_add(&_r10[_l], _r20, _n+(_r20[_n] ? 1 : 0),
&_r10[_l], _l+(_r10[_n] ? 1 : 0));
#pragma omp barrier
//Q1P0
mpn_mul_n(_r22, _r31, _p, _l);
#pragma omp barrier
}
};

```

Implémentation de la tâche 3 :

```

class Thread_Multipartite_4_4_3 : public Thread_Multipartite_4_4 {
public:
Thread_Multipartite_4_4_3(unsigned int n,
const mp_limb_t* p,
const mp_limb_t* cst)
: Thread_Multipartite_4_4(n, p, cst) {}

void multiplication(mp_limb_t* C,
const mp_limb_t* A,
const mp_limb_t* B) {
//A3B3 A2B2 A2B0
mpn_mul_n(_r30, &A[_l], B, _s);
mpn_mul_n(&_r30[_l], &A[_l], &B[_l], _s);
mpn_mul_n(&_r30[_n], &A[_l+_s], &B[_l+_s], _s);
mpn_mul(_r31, _cst, _l+1, &_r30[_n], _l);
#pragma omp barrier
mpn_add(_r31, &_r31[_l], _l+(_r31[_n+_l] ? 1 : 0),
&_r22[_s], _s+(_r22[_l] ? 1 : 0));
#pragma omp barrier
//Q1P1
mpn_mul_n(&_qp[_n], _r31, _p, _l);
mpn_sub_1(&_qp[_n], &_qp[_n], _n, 1);
#pragma omp barrier
}
};

```

Finalement, nous maintenons un ensemble de tâches dites actives. La classe suivante illustre l'implémentation de cet ensemble. Nous avons deux fonctions de multiplications : une avec lancement de processus et une seconde qui se greffe sur des processus déjà lancé.

```

template < unsigned int N >
class active_threads<N, Thread_Multipartite_4_4> {
public:
    unsigned int _n;
    Thread_Multipartite_4_4 *_threads[4];
    const mpz_t *_P;
    //P : modulus, I : Montgomery constant, B : Barrett constant
    active_threads(const mpz_t& P,const mpz_t& I,const mpz_t& B) {
        _n = N >> bits_per_limb_p;
        _P = &P;
        Thread_Multipartite_4_4::_r00 =
            (mp_limb_t*) malloc(sizeof(mp_limb_t)*(_n << 1));
        [...]
        _threads[0] =
            new Thread_Multipartite_4_4_0(_n,P[0]._mp_d,I[0]._mp_d);
        _threads[1] =
            new Thread_Multipartite_4_4_1(_n,&P[0]._mp_d[_n >> 1], I[0]._mp_d);
        _threads[2] =
            new Thread_Multipartite_4_4_2(_n,P[0]._mp_d,&B[0]._mp_d[_n >> 2]);
        _threads[3] =
            new Thread_Multipartite_4_4_3(_n,&P[0]._mp_d[_n >> 1], B[0]._mp_d);
    }
    void multiplication(mpz_t& C, const mpz_t& A, const mpz_t& B) {
#pragma omp parallel num_threads(4)
    {
        int idx = omp_get_thread_num();
        _threads[idx]->multiplication(C[0]._mp_d, A[0]._mp_d, B[0]._mp_d);
    }
    //Final subtractions done by only the master thread
    mpn_sub_n(C[0]._mp_d, &Thread_Multipartite_4_4::_r00[_n >> 1],
        &Thread_Multipartite_4_4::_qp[_n >> 1], _n+1);
    C[0]._mp_size = _n + (C[0]._mp_d[_n] ? 1 : 0);
    while(mpz_cmp(C, *_P) >= 0) {
        mpz_sub(C,C,*_P);
    }
}
void mult_in(mpz_t& C, const mpz_t& A, const mpz_t& B) {
    int idx = omp_get_thread_num();
    _threads[idx]->multiplication(C[0]._mp_d, A[0]._mp_d, B[0]._mp_d);
#pragma omp barrier
    if(!idx) { //Final Subtractions done by the thread 0
        mpn_sub_n(C[0]._mp_d, &Thread_Multipartite_4_4::_r00[_n >> 1],
            &Thread_Multipartite_4_4::_qp[_n >> 1], _n+1);
        C[0]._mp_size = _n + (C[0]._mp_d[_n] ? 1 : 0);
        while(mpz_cmp(C, *_P) >= 0) {
            mpz_sub(C,C,*_P);
        }
    }
    //Final synchronization needed to insure the correctness of the result
#pragma omp barrier
}
};

```

- [1] National Security Agency. NSA suite B cryptography. http://www.nsa.gov/ia/programs/suiteb_cryptography/index.shtml.
- [2] J. Allali, P. Ferraro, P. Hanna, and C. Iliopoulos. Local transpositions in alignment of polyphonic musical sequences. In *String Processing and Information Retrieval Symposium, 14th International Symposium, SPIRE 2007*, volume 4726 of *Lecture Notes in Computer Science*, pages 26–38. Springer, October 2007.
- [3] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), pages 483–485. ACM, 1967.
- [4] B. Arnaud, A. Gailhac, T. Iazard, and S. Padou. Étude de la division entière sur les processeurs modernes. Projet de recherche, 2006.
- [5] P. Barrett. Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In *Advances in Cryptology - CRYPTO 86*, volume 263/1987 of *Lecture Notes in Computer Science*, pages 311–323. Springer-Verlag, 1987.
- [6] D. J. Bernstein. Fast multiplication and its applications. *Algorithmic number theory*, 44:325–384, 2008.
- [7] D. J. Bernstein, T. Chen, C. Cheng, T. Lange, and B. Yang. ECM on graphics cards. Cryptology ePrint Archive, Report 2008/480, 2008. <http://eprint.iacr.org/>.
- [8] D. J. Bernstein and T. Lange. Analysis and optimization of elliptic-curve single-scalar multiplication. Cryptology ePrint Archive, Report 2007/455, 2007.
- [9] D. J. Bernstein and T. Lange. Faster addition and doubling on elliptic curves. In *Advances in Cryptology : ASIACRYPT 2007*, volume 4833 of *Lecture Notes in Computer Science*, pages 29–50. Springer, 2007.
- [10] D. J. Bernstein and T. Lange. Inverted Edwards Coordinates. In *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes*, volume 4851/2007 of *Lecture Notes in Computer Science*, pages 20–27. Springer-Verlag, 2007.
- [11] I. Blake, G. Seroussi, and N. Smart, editors. *Advances in Elliptic Curve Cryptography*. Number 317 in London Mathematical Society Lecture Note Series. Cambridge University Press, 2005.
- [12] G. R. Blakely. A computer algorithm for calculating the product AB modulo M . *IEEE Transactions on Computers*, 32(5):497–500, 1983.
- [13] Guy E. Blelloch. Nesl: A nested data-parallel language (version 3.1). Technical report, Carnegie Mellon University, Pittsburgh, PA, USA, 1995.
- [14] R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. In *Proceedings of the fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 207–216. ACM, 1995.
- [15] R. Blumofe and C. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46:720–748, September 1999.

-
- [16] M. Bodrato and A. Zanzi. Integer and polynomial multiplication : Towards optimal Toom-Cook matrices. In *Proceedings of the 2007 international symposium on Symbolic and algebraic computation*, ISSAC '07, pages 17–24. ACM, 2007.
- [17] A. Brauer. On addition chains. *Bulletin of American Mathematical Society*, 45:736–739, 1939.
- [18] R. P. Brent and P. Zimmerman. *Modern Computer Arithmetic*. Cambridge University Press, 2010.
- [19] Certicom Research. Standards for efficient cryptography SEC 1 : Elliptic curve cryptography. Technical report, Certicom Corp, 2000.
- [20] M. Ciet, M. Joye, K. Lauter, and P. Montgomery. Trading inversions for multiplications in elliptic curve cryptography. *Design, Codes and Cryptography*, 39(2):189–206, 2006.
- [21] H. Cohen, C. Doche, G. Frey, T. Lange, K. Nguyen, F. Vercauteren, and R. M. Avanzi. *Handbook of elliptic and hyperelliptic curve cryptography*, volume 34 of *Discrete Mathematics and Its Applications*. Chapman and Hall/CRC, 2005.
- [22] S. A. Cook. *On the minimum computation time of functions*. PhD thesis, Department of Mathematics, Harvard University, 1966.
- [23] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22:644–654, 1976.
- [24] V. Dimitrov, L. Imbert, and P. K. Mishra. The double-base number system and its application to elliptic curve cryptography. *Mathematics of Computation*, 77(262):1075–1104, 2008.
- [25] V. Dimitrov and P. K. Mishra. Efficient quintuple formulas for elliptic curves and efficient scalar multiplication using multibase number. In *Information Security 10th International Conference*, volume 4779 of *Lecture Notes in Computer Science*, pages 390–406. Springer-Verlag, 2007.
- [26] Vassili Dimitrov, Laurent Imbert, and Pradeep K. Mishra. Efficient and secure elliptic curve point multiplication using double-base chains. In *Advances in Cryptology, ASIACRYPT'05*, volume 3788 of *Lecture Notes in Computer Science*, pages 59–78. Springer, 2005.
- [27] ECRYPT II. Yearly report on algorithms and key sizes (2009-2010). Technical report, European Network of Excellence in Cryptology II, 2010.
- [28] H. Edwards. A normal form for elliptic curves. *Bulletin of the American Mathematical Society*, 44(3):393–422, July 2007.
- [29] T. El Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In *Proceedings of CRYPTO 84 on Advances in cryptology*, volume 196 of *Advances in cryptology*, pages 10–18. Springer-Verlag, 1985.
- [30] M. D. Ercegovac and T. Lang. *Digital Arithmetic*. Morgan Kaufmann Publishers, 2004.
- [31] P. Ferraro, P. Hanna, L. Imbert, and T. Izard. Accelerating query-by-humming on GPU. In *Proceedings of the 10th International Society for Music Information Retrieval Conference (ISMIR'09)*, pages 279–284, Kobe, Japan, October 2009.
- [32] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions On Computers*, C-21(9):948–960, 1972.
- [33] M. Fürer. Faster integer multiplication. In *STOC '07: Proceedings of the thirty-ninth annual ACM symposium on Theory of computing*, pages 57–66, New York, NY, USA, 2007. ACM.
- [34] S. Galbraith, X. Lin, and M. Scott. Endomorphisms for faster elliptic curve cryptography on a large class of curves. *Journal of Cryptology*, 24(3):446–469, 2010.
- [35] R. Gallant, R. Lambert, and S. Vanstone. Faster point multiplication on elliptic curves with efficient endomorphisms. In *Advances in Cryptology — CRYPTO 2001*, volume 2139/2001 of *Lecture Notes in Computer Science*, pages 190–200. Springer, 2001.
- [36] M. R. Garey and D.S. Johnson. *Computer and Intractability : a guide to the theory of NP-Completeness*. W. H. Freeman, 1979.
- [37] P. Gaudry, A. Kruppa, and P. Zimmermann. A GMP-based implementation of Schönhage-Strassen's large integer multiplication algorithm. In *ISSAC '07: Proceedings of the 2007 International Symposium on Symbolic and Algebraic Computation*, pages 167–174, New York, NY, USA, 2007. ACM.

-
- [38] T. Gautier, X. Besson, and L. Pigeon. Kaapi: A thread scheduling runtime system for data flow computations on cluster of multi-processors. In *Proceedings of the 2007 international workshop on Parallel symbolic computation*, PASCO '07, pages 15–23. ACM, 2007.
- [39] P. Giorgi, L. Imbert, and T. Izard. Optimizing elliptic curve scalar multiplication for small scalars. In *Mathematics for Signal and Information Processing, Proc. of SPIE*, volume 7444, page 74440N, San Diego, CA, USA, August 2009. SPIE.
- [40] P. Giorgi, L. Imbert, and T. Izard. Multipartite modular multiplication. Soumis, 2011.
- [41] P. Giorgi, T. Izard, and A. Tisserand. Comparison of modular arithmetic algorithms on GPU. In *Parallel Computing: From Multicores and GPU's to Petascale*, volume 19 of *Advances in Parallel Computing*, pages 315–322, Lyon, France, September 2010. IOS Press.
- [42] D. M. Gordon. A survey of fast exponentiation methods. *Journal of Algorithms*, 27(1):129–146, April 1998.
- [43] D. Hankerson, A. Menezes, and S. Vanstone. *Guide to Elliptic Curve Cryptography*. Springer-Verlag, 2004.
- [44] M. D. Hill and M. R. Marty. Amdahl's law in the multicore era. *IEEE Computer*, 2008.
- [45] H. Hisil, G. Carter, and E. Dawson. New formulae for efficient elliptic curve arithmetic. In *Proceedings of the cryptology 8th international conference on Progress in cryptology - INDOCRYPT'07*, volume 4859 of *Lecture notes in computer science*, pages 138–151. Springer-Verlag, 2007.
- [46] O. Ibarra and S. Moran. Probabilistic algorithms for deciding equivalence of straight-line programs. *Journal of the Association for Computing Machinery (JACM)*, 30(1):217–228, 1983.
- [47] L. Imbert, A. Peirera, and A. Tisserand. A library for prototyping the computer arithmetic level in elliptic curve cryptography. In *Proceedings of Advanced Signal Processing Algorithms, Architectures and Implementations XVII*, volume 6697, pages 1–9. SPIE, August 2007.
- [48] D. Johnson, A. Menezes, and S. Vanstone. The elliptic curve digital signature algorithm (ECDSA). *International Journal of Information Security*, 1(1):36–63, 2001.
- [49] M. E. Kaihara and N. Takagi. Bipartite modular multiplication method. *IEEE Transactions on Computers*, 57(2):157–164, 2008.
- [50] E. Kaltofen. Greatest common divisors of polynomials given by straight-line programs. *Journal of the Association for Computing Machinery (JACM)*, 35(1):231–264, 1988.
- [51] E. Kaltofen. Factorization of polynomials given by straight-line programs. In S. Micali, editor, *Randomness and Computation*, volume 5 of *Advances in Computing Research*, pages 375–412. JAI Press Inc., 1989.
- [52] A. Karatsuba and Y. Ofman. Multiplication of multidigit numbers on automata. *Soviet Physics-Doklady*, 7:595–596, 1963.
- [53] A. Kerckhoffs. La cryptographie militaire. *Journal des sciences militaires*, IX :5–38, 1883.
- [54] T. Kleinjung, K. Aoki, J. Franke, A. Lenstra, E. Thomé, J. Bos, P. Gaudry, A. Kruppa, P. Montgomery, D. A. Osvik, H. Te Riele, A. Timofeev, and P. Zimmermann. Factorization of a 768-bit RSA modulus. In *CRYPTO 2010 Advances in Cryptology - CRYPTO 2010*, volume 6223 of *Lecture Notes in Computer Science*, pages 333–350, Santa Barbara États-Unis, 2010. Springer Verlag.
- [55] T. Kleinjung, L. Nussbaum, and E. Thomé. Using a grid platform for solving large sparse linear systems over GF(2). In *11th ACM/IEEE International Conference on Grid Computing (Grid 2010)*, 2010.
- [56] D. E. Knuth. *The art of computer programming, volume 2 (3rd ed.): seminumerical algorithms*. Addison-Wesley Longman Publishing Co., Inc., 1997.
- [57] N. Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48(177):203–209, 1987.
- [58] C. K. Koc, B. S. Kaliski Jr., and T. Acar. Analyzing and comparing Montgomery multiplication algorithms. *IEEE Micro*, 16(3):26–33, June 1996.
- [59] P. Longa and C. Gebotys. Setting speed records with the (fractional) multibase non-adjacent form method for efficient elliptic curve scalar multiplication. Cryptology ePrint Archive, Report 2008/118, 2008.

-
- [60] P. Longa and A. Miri. Fast and flexible elliptic curve point arithmetic over prime fields. *IEEE Transactions on computers*, 57(3):289–302, 2007.
- [61] P. Longa and A. Miri. New composite operations and precomputation scheme for elliptic curve cryptosystems over prime fields. In *Public Key Cryptography - PKC 2008*, volume 4939 of *Lecture Notes in Computer Science*, pages 229–247. Springer, 2008.
- [62] P. Longa and A. Miri. New multibase non-adjacent form scalar multiplication and its application to elliptic curve cryptosystems. *Cryptology ePrint Archive*, Report 2008/052, 2008.
- [63] S. Manavski and G. Valle. CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment. *BMC Bioinformatics*, 9(2):10–, 2008.
- [64] N. Méloni. *Arithmétique pour la Cryptographie basée sur les Courbes Elliptiques*. PhD thesis, Université Montpellier 2, 2007.
- [65] A. J. Menezes, S. A. Vanstone, and P. C. Van Oorschot. *Handbook of applied cryptography*. CRC Press, 2001.
- [66] V. S. Miller. Use of elliptic curves in cryptography. In *Advances in cryptology - CRYPTO 85*, *Lecture Notes in Computer Science*, pages 417–426. Springer-Verlag, 1985.
- [67] P. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, Apr. 1985.
- [68] A. Moss, D. Page, and N. P. Smart. Toward acceleration of RSA using 3D graphics hardware. In *Cryptography and Coding*, volume 4887/2007 of *Lecture Notes in Computer Science*, pages 364–383. Springer-Verlag, 2007.
- [69] J.-M. Muller. *Arithmétique des ordinateurs*. Études et recherches en informatique. Masson, 1989.
- [70] Y. Munekawa, F. Ino, and K. Hagihara. Design and implementation of the Smith-Waterman algorithm on the CUDA-compatible GPU. In *BIBE*, pages 1–6, 2008.
- [71] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *Queue*, 6(2):40–53, 2008.
- [72] Nvidia. *NVIDIA CUDA Best Practices*, 2009. http://www.nvidia.com/object/cuda_home_new.html.
- [73] Nvidia. *NVIDIA CUDA Programming Guide*, 2009. http://www.nvidia.com/object/cuda_home_new.html.
- [74] Nvidia. *NVIDIA CUDA Reference Manual*, 2009. http://www.nvidia.com/object/cuda_home_new.html.
- [75] S. Olivier, A. Porterfield, K. Wheeler, and J. Prins. Scheduling task parallelism on multi-socket multicore systems. In *Proceedings of the 1st International Workshop on Runtime and Operating Systems for Supercomputers*, ROSS '11, pages 49–56. ACM, 2011.
- [76] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.
- [77] T. Plantard. *Arithmétique modulaire pour la cryptographie*. PhD thesis, Université Montpellier 2, 2005.
- [78] J. Pollard. A Monte-Carlo method for factorization. *BIT Numerical Mathematics*, 15:331–334, 1975.
- [79] M. Rinard, J. Scales, and M. Lam. Heterogeneous parallel programming in jade. In *Proceedings of the 1992 ACM/IEEE conference on Supercomputing*, Supercomputing '92, pages 245–256. IEEE Computer Society Press, 1992.
- [80] R. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, 1978.
- [81] Y. Sakai and K. Sakurai. Efficient scalar multiplication on elliptic curves with direct computations of several doublings. *IEICE TRANS. Fundamentals*, E84-A:120–129, 2001.
- [82] K. Sakiyama, M. Knezevic, J. Fan, B. Preneel, and I. Verbauwhede. Tripartite modular multiplication. *Integration, the VLSI Journal*, 2011.

-
- [83] J. Savage. *Models of Computation*. Brown University, 2008. <http://www.cs.brown.edu/people/jes/book/pdfs/ModelsOfComputation.pdf>.
- [84] A. Schönhage and V. Strassen. Schnelle Multiplikation großer Zahlen. *Computing*, 7:281–292, 1971.
- [85] R. Schoof. Elliptic curves over finite fields and the computation of square roots mod p . *Mathematics of Computation*, 44:483–494, 1985.
- [86] J. T. Schwartz. Fast probabilistic algorithms for verification of polynomial identities. *J. ACM*, 27:701–717, October 1980.
- [87] D. Shanks. Class number, a theory of factorization and genera. *Proceedings of Symposia in Pure Mathematics*, 20, 1971.
- [88] P. W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26(5):1484–1509, 1997.
- [89] V. Shoup. Lower bounds for discrete logarithms and related problems. In *EUROCRYPT'97 Proceedings of the 16th annual international conference on Theory and application of cryptographic techniques*, volume 1233 of *Lecture Notes in Computer Science*, pages 256–266. Springer-Verlag, 1997.
- [90] J. H. Silverman. *The Arithmetic of Elliptic Curves (Second Edition)*, volume 106 of *Graduate Texts In Mathematics*. Springer-Verlag, 2009.
- [91] S. Singh. *Histoire des codes secrets*. JC Lattès, 1999.
- [92] T. Smith and M. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147:195–197, 1981.
- [93] D. Stinson. *Cryptography: Theory and Practice, Third Edition (Discrete Mathematics and Its Applications)*. Chapman & Hall/CRC, 2005.
- [94] R. Szerwinski and T. Güneysu. Exploiting the power of GPUs for asymmetric cryptography. In *CHES '08: Proceeding sof the 10th international workshop on Cryptographic Hardware and Embedded Systems*, volume 5154/2008 of *Lecture Notes in Computer Science*, pages 79–99, Berlin, Heidelberg, 2008. Springer-Verlag.
- [95] R. Szerwinski and T. Güneysu. Exploiting the power of GPUs for asymmetric cryptography. In Springer, editor, *Proc. Cryptographic Hardware and Embedded Systems*, volume 5154, pages 79–99, 2008.
- [96] M. Süß and C. Leopold. Common mistakes in OpenMP and how to avoid them a collection of best practices. In *OpenMP Shared Memory Parallel Programming*, volume 4315 of *Lecture Notes in Computer Science*, pages 312–323. Springer-Verlag, 2008.
- [97] T. Takagi, S. Yen, and B. Wu. Radix- r non-adjacent form. In *Information Security*, volume 3225/2004 of *Lecture Notes in Computer Science*. Springer-Verlag, 2004.
- [98] D. Tarditi, S. Puri, and J. Oglesby. Accelerator: using data parallelism to program GPUs for general-purpose uses. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 325–335, New York, NY, USA, 2006. ACM.
- [99] A. L. Toom. The complexity of a scheme of functional elements realizing the multiplication of integers. *Soviet Mathematics Doklady*, 3:714–716, 1963.
- [100] L. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33:103–111, August 1990.
- [101] J. Von Zur Gathen and J. Gerhard. *Modern Computer Algebra*. Cambridge University Press, New York, NY, USA, 2 edition, 2003.
- [102] Cyril Zeller. Tutorial CUDA, 2008.
- [103] R. Zippel. Probabilistic algorithms for sparse polynomials. In *Symbolic and Algebraic Computation*, volume 72 of *Lecture Notes in Computer Science*, pages 216–226. Springer-Verlag, 1979.
- [104] J. A. Zverina. GPUs versus CPUs: Apples and oranges? <https://www.teragrid.org/web/news/gpuvscpu>, March 2011.

Résumé

Les protocoles de cryptographie asymétrique nécessitent des calculs arithmétiques dans différentes structures mathématiques. Un grand nombre de ces protocoles requièrent en particulier des calculs dans des structures finies, rendant indispensable une arithmétique modulaire efficace. Ces opérations modulaires sont composées d'opérations multiprécision entre opérandes de tailles suffisamment grandes pour garantir le niveau de sécurité requis (de plusieurs centaines à plusieurs milliers de bits). Enfin, certains protocoles nécessitent des opérations arithmétiques dans le groupe des points d'une courbe elliptique, opérations elles-mêmes composées d'opérations dans le corps de définition de la courbe. Les tailles de clés utilisées par les protocoles rendent ainsi les opérations arithmétiques coûteuses en temps de calcul. Par ailleurs, les architectures grand public actuelles embarquent plusieurs unités de calcul, réparties sur les processeurs et éventuellement sur les cartes graphiques. Ces ressources sont aujourd'hui facilement exploitables grâce à des interfaces de programmation parallèle comme OpenMP ou CUDA. Cette thèse s'articule autour de la définition d'opérateurs arithmétiques parallèles permettant de tirer parti de l'ensemble des ressources de calcul, en particulier sur des architectures multicœur à mémoire partagée. La parallélisation au niveau arithmétique le plus bas permet des gains modérés en termes temps de calcul, car les tailles des opérandes ne sont pas suffisamment importantes pour que l'intensité arithmétique des calculs masque les latences dues au parallélisme. Nous proposons donc des algorithmes permettant une parallélisation aux niveaux arithmétiques supérieurs : algorithmes parallèles pour la multiplication modulaire et pour la multiplication scalaire sur les courbes elliptiques. Pour la multiplication modulaire, nous étudions en particulier plusieurs ordonnancements des calculs au niveau de l'arithmétique modulaire et proposons également une parallélisation à deux niveaux : modulaire et multiprécision. Ce parallélisme à plus gros grain permet en pratique des gains plus conséquents en temps de calcul. Nous proposons également une parallélisation sur processeur graphique d'opérations modulaires et d'opérations dans le groupe des points d'une courbe elliptique. Enfin, nous présentons une méthode pour optimiser la multiplication scalaire sur les courbes elliptiques pour de petits scalaires.

Mots clefs : *Arithmétique multiprécision - Arithmétique modulaire - Arithmétique des courbes elliptiques - Parallélisme - Calcul haute-performance*

Abstract

Protocols for asymmetric cryptography require arithmetic computations in several mathematical structures. In particular, many of them need computations in finite structures, imposing an efficient modular arithmetic. These modular calculations are composed of multiprecision operations between operands of sizes large enough to insure the required level of security (between several hundred and several thousand of bits). Finally, some protocols need arithmetic operations in the group of points of an elliptic curve, which are themselves composed of modular computations. The sizes of the keys used by the protocols make the arithmetic computations expansive in terms of execution time. Nowadays, current architectures have several computing units, which are distributed over the processors and GPU. These resources are now easily programmable using dedicated languages such as OpenMP or CUDA. This thesis focuses on the definition of parallel algorithms to take advantage of all the computing resources of multi-core shared-memory architectures. Parallelism at the lowest arithmetic level gives a moderate speedup since the sizes of the operands are not large enough so that the arithmetic intensity hides the latencies induced by the parallelism. We propose algorithms for parallelization at higher arithmetic: parallel algorithms for modular multiplication and scalar multiplication on elliptic curves. For modular multiplication, we study in particular several schedulings of modular computations. We also propose a two-level parallelization, at modular and multiprecision levels. This “coarse-grained” parallelism allows in practice a more substantial speedup. We also present a parallelization of modular and elliptic curves operations on GPU. Finally, we introduce a method to optimize scalar multiplication on elliptic curves for small scalars.

Keywords: *Multiprecision arithmetic - Modular arithmetic - Elliptic curves arithmetic - Parallelism - High-performance Computing*