CORESA 2013 - 16ème édition du colloque COmpression et REprésentation des Signaux Audiovisuels
Le Creusot, FRANCE, 28 et 29 novembre 2013

Recompaction de données non-structurées nano-lithographiques

V. Baticle[1,2], F. Payan[2], G. Renault[1], M. Antonini[2] et P. Schiavone[1]

[1]ASELTA Nanographics, MINATEX - BHT
7, parvis Louis Néel, Grenoble, FRANCE
{virginie.baticle, guillaume.renault, patrick.schiavone}@aselta.com

[2]Laboratoire I3S, UMR7271 - Université de Nice Sophia Antipolis CNRS
2000 route des Lucioles, Les Algorithmes - bât. Euclide B, Sophia-Antipolis, FRANCE
{fpayan, am}@i3S.unice.fr

Résumé

Les données provenant de la nanolithographie à faisceau d'électrons définissent un ensemble de formes décrites dans un espace à 2 dimensions. Ces informations présentent de fortes redondances géométriques et spatiales. Certains formats propres à la nanolithographie, OASIS par exemple, proposent une structure spécifique, appelée "répéteurs". Ces répéteurs exploitent les redondances afin de rendre les fichiers plus compacts. Aujourd'hui, les logiciels spécialisés dans la création de ces designs *n'exploitent pas toujours cette spécificité, et produisent donc des fichiers extrêmement volumineux, rendant difficile leur manipulation, leur transfert et leur stockage. Nous présentons un algorithme permettant de recompacter ces données. L'objectif de notre algorithme sera de retrouver ces répétitions afin de réduire la taille de nos fichiers. La principale caractéristique de notre méthode est d'être capable de traiter des données* out of core *en utilisant un traitement au fil de l'eau. Les résultats expérimentaux montrent que notre algorithme permet de réduire la taille des fichiers de manière significative.*

Mots clefs

Compaction, Lithographie à faisceau d'électrons, Format OASIS, Traitement au fil de l'eau, Données $out\ of\ core$.

1 Introduction

Dans le cadre de la réalisation de circuits intégrés, la lithographie est un processus largement répandu dans le domaine de la nanotechnologie. Le principe est de dupliquer un motif sur un substrat (plaque de semi-conducteur ou masque de photo-lithographie) en utilisant des photons (lithographie optique) ou un faisceau d'électrons [1]. Conformément à la loi de Moore [2], le nombre de transistors des circuits intégrés double à chaque génération. Cette évolution de la performance implique une meilleure résolution des motifs utilisés dans la lithographie à faisceau d'électrons et une complexification des *designs*. Un *design* propre à cette lithographie, qui n'est rien d'autre qu'un ensemble de polygones dans un espace *2D*, peut contenir jusqu'à plusieurs centaines de milliards de formes dans les technologies les plus avancées. Décrites une à une, ces données représentent plusieurs Teraoctets d'informations, ce qui limite le transfert et le stockage.

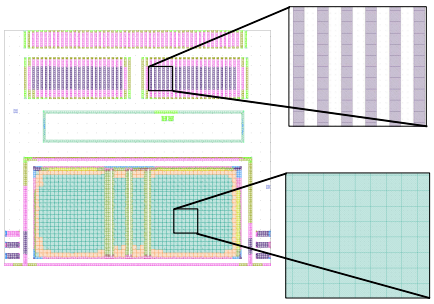

Figure 1 – *Exemple de données lithographiques, présentant de fortes redondances spatiales et géométriques.*

Heureusement, ces *designs* présentent de fortes redondances. En effet, de nombreux motifs possèdent des caractéristiques communes telles que la forme, la taille et/ou l'orientation... Par exemple, un *design* de quelques milliers de formes, peut contenir qu'une dizaine de polygones différents. De plus, un lot de formes identiques peut également présenter des redondances spatiales (voir figure 1) et être décrit comme un bloc de formes répétées. En utilisant une telle structure, il est possible de réduire considérablement la quantité d'information nécessaire pour décrire ces données.

Différents formats de fichiers (OASIS [3], JEOL [4], MEBES, VSB12...) prennent en compte la spécificité des données de la nanolithographie en proposant un moyen de structurer les données. Par exemple, le format OASIS [3] introduit la notion de *répéteurs* afin d'exploiter les redondances spatiales (Plus de détails sont donnés dans la section 2).

Par ailleurs, même si les données sont intrinséquement

structurées hiérarchiquement et spatialement, plusieurs traitements (tels que la "correction des effets de proximité") nécessitent de mettre à plat les données (c'est à dire "casser" la structure hiérarchique déjà présente). Autrement dit, même si les données d'entrées possèdent une certaine structure (comme les répéteurs), les fichiers de sortie de ces logiciels deviennent des données *out of core*, du fait que les répétitions ne sont plus utilisées.
Dans cet article, nous présentons une technique de recompaction de données lithographiques. L'idée est d'exploiter au mieux les redondances afin de décrire un maximum de données avec des répéteurs. Cette méthode présente les deux avantages suivants :

– Les données peuvent être traitées au fil de l'eau [5]. Ceci permet de commencer le processus sans avoir besoin de lire la totalité des données. En effet, la plupart des *designs* de la lithographie sont *out of core*, signifiant que l'on ne peut charger en mémoire la totalité du fichier à traiter.
– Le fichier de sortie est compatible avec le format OASIS, mais également avec tous les autres formats supportant la structure des répéteurs (JEOL, MEBES, VSB...).

2 Description du format OASIS

OASIS (*Open Artwork System Interchange Standard*) [3] est un format utilisé pour représenter les motifs des circuits intégrés. Ce format est devenu un standard dans l'industrie de la microélectronique.
OASIS permet de décrire différents types de formes dans un espace *2D* : carré, rectangle, polygones... En plus de ces différentes caractéristiques, le format définit les positions relatives de ces formes.

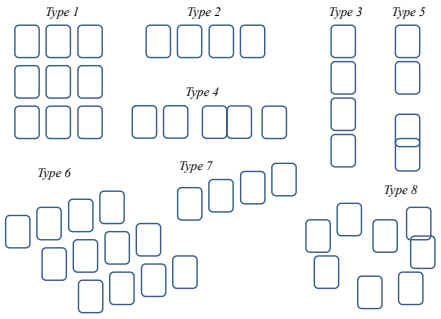

Figure 2 – *Les différents types de répétitions présent dans le format OASIS.*

En outre, ce format propose des opérateurs de compaction dans le but de minimiser la taille du fichier : les répéteurs. Un répéteur peut être définie comme étant une structure de formes géométriques. Il existe huit types de répéteurs, comme illustré sur la figure 2.
Suivant le type, un répéteur peut être défini selon les paramètres suivants :

– La forme géométrique de référence.
– La dimension (m, n) correspondant à la taille de la matrice de formes (m lignes, n colonnes).
– L'espacement entre deux formes.
– Le vecteur déplacement entre deux formes.

La syntaxe pour décrire chacun des répéteurs est donnée par le tableau 1.

3 Intérêt du traitement au fil de l'eau

Un algorithme au fil de l'eau [5] est un algorithme qui permet de traiter les données progressivement, au fur et à mesure de leur lecture. Ceci permet de ne pas traiter la globalité des données en une seule fois. En effet, après avoir réalisé quelques tests préliminaires, nous nous sommes rendus compte que pour $2Go$ de mémoire allouée, notre logiciel [6] dans sa forme la plus simple ne pouvait pas traiter plus de vingt millions de formes. La complexité étant linéaire, cela signifie que pour traiter un *design* contenant plus d'un milliard de formes, il faudrait pouvoir allouer $100Go$ de mémoire.

Afin de contourner ce problème, nous avons basé notre algorithme sur un système de *buffers*. Il s'agit d'un terme désignant une zone de mémoire virtuelle permettant de stocker temporairement des données. Dans notre cas, nous allons définir différents *buffers* contenant chacun un type de forme. Afin de contrôler la quantité d'informations à garder en mémoire, nous imposons le nombre ainsi que la taille des *buffers* à l'aide de deux paramètres, respectivement $nb\_buffers$ et $size\_buffers$.
Evidemment, le choix de ces paramètres influencera grandement les performances de compaction de notre algorithme. Plus notre algorithme sera capable de traiter un grand nombre de formes, plus les taux de compaction obtenus devraient êtres élevés. Dans ce papier, nous étudions donc aussi la relation entre valeurs des paramètres et performances de compaction.

4 Aperçu de l'algorithme

Nous rappelons que l'objectif premier de notre algorithme est de rendre plus compacte les données en retrouvant les répétitions présentes dans un ensemble donné de formes. Afin de réduire les coûts de calculs, la version actuelle de l'algorithme se concentre exclusivement sur les répétitions de types 1,2 et 3 (voir tableau 1). Autrement dit, notre algorithme détecte seulement les répétitions avec un espacement uniforme et orthogonal. De plus, étant donnée la quantité d'informations présente dans ce type de fichiers, nous avons choisi de développer un algorithme au fil de l'eau. Pour cela, nous proposons une implémentation basée sur le remplissage de *buffers*. Notre algorithme procède itérativement, selon six étapes :

1. Lecture des formes du fichier d'entrée une par une.
2. Tri des formes en fonction de leur type (carré, rectangle, polygone...) et de leurs tailles.
3. Remplissage des différents *buffers* avec les formes triées jusqu'à ce qu'un des deux paramètres utilisateurs soit atteint. Nous avons donc un *buffer* par type

Type	Format	Description
1	x-dimension, y-dimension x-space, y-space	Matrice ou vecteur avec espacement orthogonal et uniforme.
2	x-dimension x-space	
3	y-dimension y-space	
4	x-dimension $x - space_1$ ... $x - space_{N-1}$	Vecteur avec espacement orthogonal et non uniforme.
5	y-dimension $y - space_1$ ... $y - space_{N-1}$	
6	n-dimension, m-dimension n-displacement m-displacement	Répétition avec déplacements non orthogonaux et uniformes.
7	dimension displacement	
8	$displacementx_1$ ... $displacementx_{P-1}$	Répétition avec déplacements non orthogonaux et non uniformes.

Tableau 1 – *Syntaxe des 8 types de répéteurs OASIS.*

de forme.

4. Recherche des répétitions de types 1, 2 et 3 dans un *buffer* donné.
5. Ecriture dans le fichier de sortie des répétitions détectées en respectant la syntaxe du format désiré.
6. Si toutes les données du fichier ne sont pas traitées, nous retournons à l'étape 1. L'algorithme est alors réitéré jusqu'à la fin du traitement du fichier d'entrée.

5 Description de l'algorithme

Tri

Le tri des formes est simple puisque la syntaxe d'OASIS décrit explicitement chaque type de formes [3].

Remplissage des *buffers*

Pour traiter les données *out-of-core*, nous avons défini les deux paramètres $nb\_buffers$ (nombre maximal de *buffers*) et $size\_buffers$ (taille de chaque *buffer*) permettant de contrôler le nombre de formes mises en mémoire. Cette étape consiste donc à remplir les *buffers* en fonction des formes triées jusqu'à ce que l'un des *buffers* soit plein ou que le nombre maximal de *buffers* ait été atteint.

Recherche des répétitions dans un *buffer*

Ainsi, le *buffer* contenant le plus de formes est traité afin de regrouper les formes en répéteurs. Nous retrouvons dans l'état de l'art diverses techniques pour repérer des répétitions. Certains auteurs, comme Andrew B. Kahng et G. Robins [7] traitent de l'extraction de régularités spatiales dans une image vectorisée. Afin d'optimiser cette extraction, l'article de Veltman propose par la suite une méthode permettant de trouver la plus grande répétition dans un ensemble de formes [8].
Dans notre cas, notre algorithme ne produit pas forcément la meilleure compaction atteignable. En effet, nous avons préféré favoriser le compromis entre compaction et complexité. Nous avons donc limité les calculs en simplifiant la recherche des répétitions sur les formes du *buffer* selon leurs positions (autrement dit, dans un ordre lexicographique). En effet, dans une première phase, notre algorithme trie les formes selon leurs positions pour les transformer en répétition de type 2. Ensuite une deuxième phase est faite, selon l'axe des y afin de regrouper les répéteurs de types 2 en type 1.

Ecriture des données

A ce niveau de l'algorithme, nous avons une liste de répéteurs de type 2. Nous décrivons les données dans le fichier de sortie, au fur et à mesure de leur apparition. Par exemple, si le premier répéteur de notre liste peut être regroupé avec un deuxième répéteur de la liste, alors nous décrivons ces données comme étant un répéteur de type 1. A la fin du traitement, il se peut qu'il reste quelques formes isolées. Dans ce cas, s'il y en a au moins deux, nous les regroupons avec un répéteur de type 8 (voir schéma 2). Le *buffer* est alors vide : nous pouvons donc retourner à l'étape 1, et itérer les mêmes étapes jusqu'à ce que toutes les formes du *design* d'entrée soient traitées.

6 Analyse des paramètres utilisateurs

6.1 Validation de l'algorithme

Dans un premier temps, nous cherchons à valider notre algorithme en vérifiant que nos résultats suivent le comportement attendu. Dans ce but, nous avons créé plusieurs fichiers synthétiques qui ont la particularité d'être structurés, de manière à étudier les différentes répétitions.

La figure 3 montre l'évolution de la taille en sortie du premier fichier analysé (nommé *design1*) en fonction de la taille des *buffers*.

Nom	Nombre total de formes	Nombre de formes différentes	Fréquence Max	Description
design1	1000000	50	20000	Fichier synthétique, contenant une répétition de type 1.
design2	975623	3684	5445	Fichier semi synthétique, bloc d'un *design* réél répété selon un type 1.
design3	10771360	9006	125712	Fichier de données réélles.
design4	7563446	2092555	6367334	Fichier de données réélles.
design5	12185229	112385	2253001	Fichier de données réélles.

Tableau 2 – *Description des différents* designs *étudiés. La fréquence maximale d'un* design *correspond au nombre maximal d'apparitions d'une même forme.*

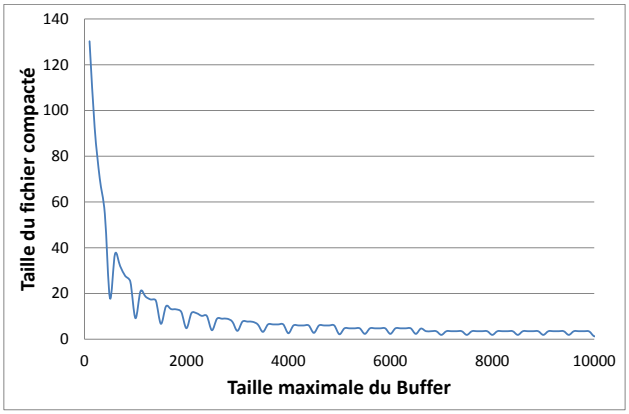

Figure 3 – *Taille (en Ko) du fichier* design1 *après compaction en fonction de la valeur de la taille des* buffers*. Le nombre maximum de* buffers *est fixé.*

Ce *design* est composé d'une seule forme, répétée selon le répéteur de type 1 (voir Tableau 2). Comme escompté, plus la taille du *buffer* est importante, plus l'algorithme traite de formes simultanément. Un nombre moins important de répéteurs est alors créé, mais ces derniers contiennent plus de formes. Le fichier sera donc finalement plus compact.
Nous observons également sur ce graphique des minimas locaux, régulièrements situés sur la courbe. Cela s'explique par le fait que dans le plus favorable des cas, seul un répéteur de type 1 sera créé ; et dans le pire des cas, des répéteurs de type 1 et 2. Imaginons, que notre *design* décrit une matrice de 10×10 formes. Si la taille de notre *buffer* est un multiple de 10, alors les données pourront être décrites uniquement avec un répéteur de type 1. Alors que si le *buffer* est de taille 35 par exemple, nous aurons inévitablement un répéteur de type 1 (de taille 30) et un répéteur de type 2 (de taille 5).

6.2 Analyse sur des fichiers réels

Nous avons ensuite étudié une partie d'un fichier réel de lithographie (nommé *design2*). Les figures 4 et 5 présentent les tailles du fichier de sortie obtenu en fonction du nombre de *buffers* et de leur taille. Sur le premier graphe (figure 4), nous avons fait varier le nombre de buffers. La taille des buffers a été fixé à 5500, ce qui correspond au nombre maximal d'apparitions d'une même forme pour ce *design*. Dans ce graphe, nous remarquons que la taille du fichier diminue réellement à partir d'environ 1900 *buffers*. C'est à ce niveau que l'algorithme devient vraiment intéressant en terme de performances de compaction.
Le deuxième graphe (figure 5) montre les performances de compaction en fonction de la taille maximale que peut prendre un *buffer*, pour une valeur de $nb\_buffers$ fixé à 3700. Nous avons choisi cette valeur pour pouvoir stocker toutes les formes possibles dans ce design puisqu'il ne contient que 3684 types de formes différentes (voir Tableau 2). Contrairement à la courbe précédente, la taille du fichier diminue fortement avec des tailles de *buffers* minimes et commence à devenir constant pour une taille d'environ 1600. Cela signifie qu'il n'est pas forcément utile de mettre des grandes tailles de *buffers*.

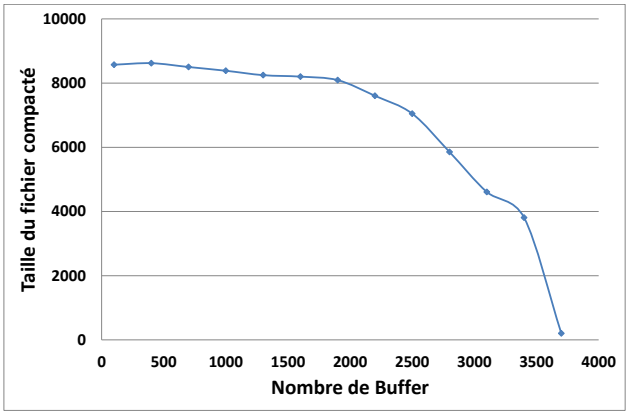

Figure 4 – *Evolution de la taille des données de sortie (en Ko) en fonction du nombre de* buffers *pour le fichier* design2.

6.3 Amélioration de la gestion des buffers

L'étude précédente nous permet de mettre en avant le fait que le nombre de *buffers* joue un rôle plus important que leur taille en terme de performances de compaction. Pour cette raison, nous avons préféré utiliser la mémoire en prio-

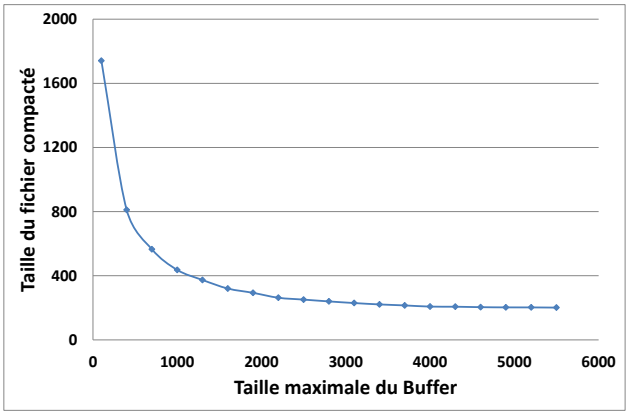

Figure 5 – *Evolution de la taille des données de sortie (en Ko) en fonction de la taille des* buffers *pour le fichier* design2.

rité pour augmenter le nombre de buffers plutôt que leur taille. C'est pour cette raison que nous avons choisi finalement de supprimer le paramètre utilisateur $nb\_buffers$. Ainsi, dans la nouvelle version de l'algorithme, un nouveau *buffer* est créé dès que l'on rencontre une nouvelle forme. Il n'y a plus de limitation quant au nombre de *buffers*. L'étape de Recherche des répétitions a lieu dès qu'un *buffer* atteint la taille maximale autorisée. De plus, ce choix est confirmé par le fait que, comme nous pouvons le voir dans le Tableau 2, cette valeur reste assez limité et ainsi, la mémoire utilisée ne risque pas d'exploser.

7 Performances de compaction

7.1 Résultats expérimentaux

Dans cette section, nous comparons nos résultats avec ceux obtenus par des logiciels libres. Il est inutile de comparer nos résultats avec des algorithmes de compression sans pertes d'images pixellisées, étant donnée la résolution très élevée des *designs* issus de la lithographie (jusqu'à 1nm par pixel). Par exemple, le fichier *design3* a une taille de 31,8Mo sans compaction (voir Tableau 3). Ce même *design* nécessiterait une image de 2,5 Tpixels pour le représenter, soit 300 Go de données sans compaction. Même avec un algorithme performant de compression d'images, il est impossible de concurrencer les approches "vectorielles".

Nous avons donc comparé nos résultas avec ceux obtenus avec WINZIP et KLAYOUT [9]. WINZIP est une méthode de compression par dictionnaire utilisant un codage à longueur variable. Cet outil est axé sur la compression de données binaires et ne prend pas en compte les redondances géométriques des *designs*. WINZIP n'est donc pas adapté aux fichiers nanolithographiques. KLAYOUT est un éditeur libre de fichiers lithographiques qui lit et traite les données dans leur globalité. KLAYOUT sera limité dans le traitement des données *out-of-core*. Cette comparaison est réalisée sur le fichier synthétique *design1* et sur 4 fichiers réels. Le tableau 3 présente le meilleur résultat pour chacun des algorithmes. Les logiciels KLAYOUT et WINZIP proposent différents niveaux de compaction, nous avons pris la meilleure. Pour notre algorithme, nous gardons la compaction la plus intéressante que l'on ait obtenu après l'étude des *buffers*. Nous observons que notre méthode réduit la taille des *designs* d'un facteur 3 à 43, selon la redondance présente dans les fichiers, et donc le nombre de répéteurs créés.

Notre algorithme est meilleur que WINZIP, excepté pour le *design3* et *design5*. Ceci peut s'expliquer par la grande variété de formes et à la faible redondance présente dans ces *designs*. En effet, une utilisation sous-optimale des répéteurs de types 1, 2 et 3 entraîne une diminution de l'efficacité de notre algorithme. WINZIP, quant à lui, n'est pas affecté par ces caractéristiques puisqu'il ne prend en compte que la redondance des symboles binaires.

En parallèle, notre méthode est meilleure que KLAYOUT pour le *design2* et le *design5* et donne des résultats similaires pour le *design4* et le *design1*. Ces résultats mettent en avant les limitations de notre algorithme. Dans l'implémentation actuelle, les répéteurs de ne sont pas optimisés, dans le sens où l'on ne cherche pas forcément à créer le plus grand répéteur. Inévitablement, nous obtenons des résultats sous-optimaux.

7.2 Amélioration

Au vu de ces résultats préliminaires, une première voie à explorer pour améliorer nos résultats est de faire une étude sur le coût binaire des répéteurs. La première question que l'on peut se poser est de savoir à partir de quelle dimension un répéteur est plus intéressant qu'un autre. En effet, après avoir étudié différents *designs* compactés, nous avons par exemple observé qu'un répéteur de type 8 pouvait être plus intéressant que plusieurs répéteurs de type 2.

Pour le format OASIS par exemple, nous pouvons représenter le coût des répéteurs en terme d'entiers non-signés (1 UI = 1 entier non-signé) :

- Une rectangle représente un coût de $6UI$
- Pour N formes répétées suivant le type 8, le coût sera de $(2 * N + 5) * UI$.
- Pour Nb_{rep} répétitions de type 2 , le coût sera de $Nb_{rep} * 8UI$. Un répéteur de type 2 présente un coût constant, quel que soit le nombre de formes qu'il contient.

A partir de ces formules, la figure 6 généralise l'évolution des coûts des répéteurs de type 2 et du coût d'un répéteur de type 8 en fonction du nombre de formes contenues. On peut donc voir sur cette courbe :

- La droite noire qui représente le coût du répéteur de type 8.
- La courbe bleu représente le coût d'*un* répéteur de type 2.
- La courbe rouge représente le coût de *deux* répéteurs de type 2.
- La courbe verte représente le coût de *trois* répéteurs de type 2.

		Méthode proposée (version 1)			KLAYOUT		WINZIP	
Fichier	Taille d'origine	Taille *Buffers*	Taille finale	Temps	Taille finale	Temps	Taille finale	Temps
design1	4.15	2×10^4	0.0009	1	0.0009	1	0.06	1
design2	8.33	5.5×10^3	0.2	2	1	1	5.9	1
design3	30.3	1.2×10^5	11.7	17	9.5	4	4.9	2
design4	77	1×10^6	2.06	6	1.7	3	4.7	6
design5	5.20	1×10^6	4.97	1	5.04	2	3.96	2

Tableau 3 – *Comparaison des résultats de compaction. La taille des fichiers est donné en mégaoctets et le temps de compaction en secondes.*

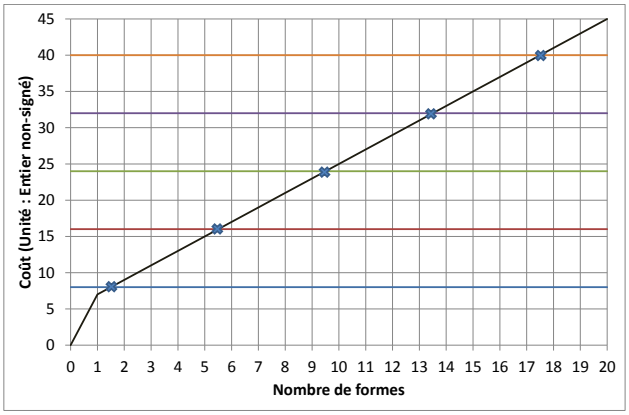

Figure 6 – *Comparaison du coût d'un répéteur de type 8 et de plusieurs répéteurs de types 2.*

Nom du Fichier	Version 1 (Mo)	Version 2 (Mo)
design1	0.0009	0.0008
design2	0.2	0.2
design3	11.7	11
design4	2.06	2.03
design5	4.97	4.97

Tableau 4 – *Privilégier la création de certains types de répéteurs par rapport à d'autres types est un moyen simple pour améliorer les taux de compaction.* Version 2 *correspond à l'amélioration proposée Section 8.*

– etc...

Une étude de ce graphique nous permet de voir que plusieurs répéteurs de type 2 contenant seulement 2 formes sont en fait moins intéressants à créer qu'un répéteur de type 8 contenant toutes ces formes qui est pourtant moins structuré *a priori*. Nous avons donc apporté une première amélioration à notre algorithme : il suffit de transformer tous nos répéteurs de type 2 et de taille 2 en répéteurs de type 8. Le tableau 4 montre que cette simple amélioration permet de réduire légèrement la taille des fichiers de sortie par rapport à la première version présentée précédemment. Le *design2* et le *design5* ne présentent que très peu de différence (au plus, de l'ordre du Ko) car très peu de répéteurs de type 2 et de taille 2 sont créés dans la version 1 de l'algorithme.

8 Conclusions et perspectives

Dans cette publication, nous avons présenté nos travaux préliminaires sur la recompaction des données non structurées destinées à la lithographie à faisceaux d'électrons. Un de nos objectifs est de pouvoir traiter des données *out of core*. Nous avons proposé une approche au fil de l'eau permettant de limiter l'utilisation de la mémoire. Nos résultats préliminaires permettent de mettre en avant un taux de compaction intéressant. Notre algorithme est capable de diminuer la taille d'un fichier d'un facteur 3 à ~ 40, selon le niveau de redondance des données d'entrée.

Il existe de nombreuses perspectives à ces travaux préliminaires. Comme indiqué dans la section 5, notre processus peut être amélioré en optimisant la recherche des répétitions [8], ou bien en privilégiant la création de certains répéteurs par rapport à d'autres, comme les résultats présentés dans la section l'ont démontré.

Références

[1] H.J. Levinson. *Principles of Lithography*. Press Monographs. Society of Photo Optical, 2005.

[2] G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.

[3] SEMI. Open artwork system interchange standard. Dans *SEMI P39-0304*, 2004.

[4] JEOL52. Data format specifications. Dans *http ://www.jeol.com/*.

[5] M. Barlaud C. Parisot, M. Antonini. 3d scan-based wavelet transform and quality control for video coding. *EURASIP J. Adv. Sig. Proc.*, 2003(1) :56–65, 2003.

[6] http ://www.aselta.com.

[7] A. B. Kahng et G. Robins. Optimal algorithms for extracting spatial regularity in images. Dans *Pattern Recognition Letters 12*, 1991.

[8] R. Veltman. Geometrical library recognition for mask data compression. Dans *Pattern Recognition Letters 12*, 1996.

[9] http ://www.klayout.de.