Lossless Compression Algorithms for the REBL Direct-Write E-Beam Lithography System

by George Cramer

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor Avideh Zakhor
Research Advisor

Date

* * * * * * *

Professor Borivoje Nikolic
Second Reader

Date

ABSTRACT

Future lithography systems must produce microchips with smaller feature sizes, while maintaining throughputs comparable to those of today's optical lithography systems. This places stringent constraints on the effective input data rate of any maskless lithography system. In recent years, my research group has developed a datapath architecture for direct-write lithography systems, and has shown that compression plays a key role in reducing throughput requirements of such systems. The approach integrates a low complexity hardware-based decoder with the writers, in order to decompress a compressed data layer in real time on the fly. In doing so, we have developed a spectrum of lossless compression algorithms for integrated circuit layout data to provide a tradeoff between compression efficiency and hardware complexity, the latest of which was Block Golomb Context Copy Coding (Block GC3). This thesis presents two lossless compression techniques, which optimize compression efficiency for use with the REBL direct-write E-beam lithography system. The first method, called Block RGC3, is a modified version of Block GC3, specifically tailored to the REBL system; the second achieves compression using vector quantization. Two characteristic features of the REBL system are a rotary stage and E-beam correction prior to writing the data. The former results in arbitrarily-rotated layout imagery to be compressed, and as such, presents significant challenges to the lossless compression algorithms. The performance of Block RGC3 and vector quantization is characterized in terms of compression efficiency and encoding complexity on a number of rotated layouts and at various angles, and it is shown that they outperform existing lossless compression algorithms.

CONTENTS

1. Introduction ….…… 4

2. Review of Block GC3 ……... 7

3. Data Path for REBL System ……... 9

4. Compression Method #1: Block RGC3 ……. 11

4.1 Modifying the copy algorithm ……... 11

4.2 Decreasing the block size ……. 11

4.3 Growing one-dimensional copy regions ……. 12

4.4 Determining the optimal block shape …….. 13

4.5 Combining neighboring regions ……... 16

4.6 Removing pixel-based prediction …… 17

5. Block RGC3 Results …… 20

5.1 Compression Results …… 20

5.2 Encoding Complexity Results …….. 24

6. Compression Method #2: Vector Quantization ……... 31

6.1 Adapting VQ to lossless image compression …….. 31

6.2 Comparision with Block RGC3 …….. 33

6.3 Determining the codebook ……... 34

7. Vector Quantization Results …….. 37

7.1 Compression Results …… 37

7.2 Encoding Complexity Results …….. 38

8. Performance Comparison: VQ vs. Block RGC3 …… 42

9. Summary, Conclusions, and Future Work ……. 44

Appendix A: NP-Completeness Proof of Region-Growing ……... 46

Appendix B: Full Chip Characterization of an ASIC, Using Block C4 ……... 47

Acknowledgments …….. 48

References ……... 49

1. INTRODUCTION

Future lithography systems must produce chips with smaller feature sizes, while maintaining throughputs comparable to today's optical lithography systems. This places stringent data handling requirements on the design of any direct-write maskless system. Optical projection systems use a mask to project the entire chip pattern in one flash. An entire wafer can then be written in a few hundreds of such flashes. To be competitive with today's optical lithography systems, direct-write maskless lithography needs to achieve throughput of one wafer layer per minute. In addition, to achieve the required 1nm edge placement with 22 nm pixels in 45 nm technology, a 5-bit per pixel data representation is needed. Combining these together, the data rate requirement for a maskless lithography system is about 12 Tb/s [4]. To achieve such a data rate, Dai and Zakhor have recently proposed a data path architecture shown in Fig. 1[1]. In this architecture, rasterized, flattened layouts of an integrated circuit (IC) are compressed and stored in a mass storage system. The compressed layouts are then transferred to the processor board with enough memory to store one layer at a time. This board then transfers the compressed layout to the writer chip, composed of a large number of decoders and actual writing elements. The outputs of the decoders correspond to uncompressed layout data, and are fed into D/A converters driving the writing elements such as a micro-mirror array or E-beam writers. In this architecture, the writer system is independent of the data-delivery path, and as such, the path and the compression algorithm can be applied to arbitrary direct-write lithography systems.

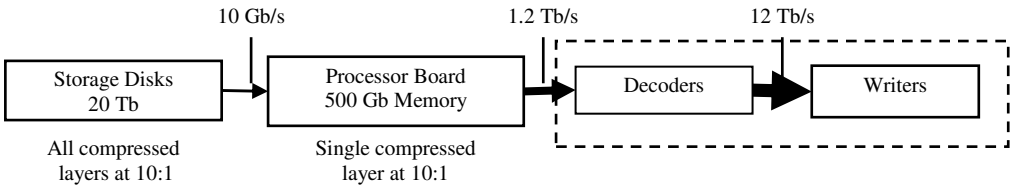


Fig. 1. The data-delivery path of the direct-write systems.

In the proposed data-delivery path, compression is needed to minimize the transfer rate between the processor board and the writer chip, and also to minimize the required disk space to store the layout. Since there are a large number of decoders operating in parallel on the writer chip, an important

requirement for any compression algorithm is to have an extremely low decoder complexity. To this end, our research group at UC Berkeley has developed a class of lossless layout compression algorithms for flattened, rasterized data [1-4], the latest of which, Block Golomb Context Copy Coding (Block GC3), has been shown to outperform all existing techniques such as BZIP2, 2D-LZ, and LZ77 in terms of compression efficiency, especially with limited decoder buffer size and hardware complexity, as required for hardware implementation[2][4].

A new writing system, called Reflective Electron Beam Lithography (REBL), is currently under development at KLA-Tencor[5]. In this system, the layout patterns are written on a rotary writing stage, resulting in layout data which is rotated at arbitrary angles with respect to the pixel grid. Moreover, the data is subjected to E-beam proximity correction effects. We have empirically found that applying the Block GC3 algorithm [4] to E-beam proximity corrected and rotated layout data results in poor compression efficiency far below those obtained on Manhattan geometry and without E-beam proximity correction. Consequently, Block GC3 needs to be modified to accommodate the characteristics of REBL data while maintaining a low-complexity decoder for the hardware implementation. This thesis modifies Block GC3 in a number of ways in order to make it applicable to the REBL system; we refer to this new algorithm as Block Rotated Golomb Context Copy Coding (Block RGC3). Additionally, the performance of a vector quantization-based compression technique is compared with that of Block RGC3.

The outline of this thesis is as follows. Section 2 gives a brief summary of the Block GC3 algorithm. Section 3 introduces the data-delivery path of the REBL system and the requirements it imposes on the compression techniques. Section 4 describes the modifications that form Block RGC3. These include an alternate copy algorithm, which better suits layout patterns rotated with respect to the pixel grid, as well as a modified Block GC3 encoder, which introduces spatial coherence among neighboring blocks of Block GC3. Section 5 presents Block RGC3 compression results for different layout data with different parameters; a characterization of the additional encoding complexity required to implement the proposed changes to Block GC3 is also included. Section 6 describes the vector quantization compression

algorithm, with results presented in Section 7. A performance comparison between Block RGC3 and vector quantization is given in Section 8. Conclusions and future work are presented in Section 9.

2. REVIEW OF BLOCK GC3

The Block GC3 architecture is shown in Fig. 2, and is detailed in [4]. A summary of the algorithm is as follows. Each non-overlapping $H \times W$ section of pixels is grouped together as a "block". Each block is assigned a "segmentation value," which takes on either (a) a copy distance selecting an already-decoded $H \times W$ image section to copy from in LZ77 fashion, or (b) a copy distance of "0", which signals the decoder to predict the pixel values based on the properties of neighboring pixels. The set of segmentation values for the blocks forms its own pixel grid called the segmentation map, which is compressed by the region encoder block using a similar prediction method. Copy distances are strictly aligned either above or to the left of the block being encoded; diagonal copying is not allowed, in order to minimize encoding complexity. In essence, Block GC3 attempts to exploit the benefits of both LZ-style copying and local context-based prediction. However, neither the segmentation values nor the image pixels are likely to be encoded free of errors. Specifically, to achieve higher compression efficiency, we allow errors to be introduced within the copy/prediction operation for the image and within the prediction operation for the segmentation map. For both the image and the segmentation map, error correction is performed by separating the bitstream into two sections: (a) an error map, in which a "zero" ("one") pixel signifies a correctly (incorrectly) predicted value for the corresponding pixel, and (b) a list of error values. Both the image and segmentation error maps are compressed using Golomb run-length codes. The image error values are compressed using Huffman codes, while the segmentation error values are transmitted uncompressed.

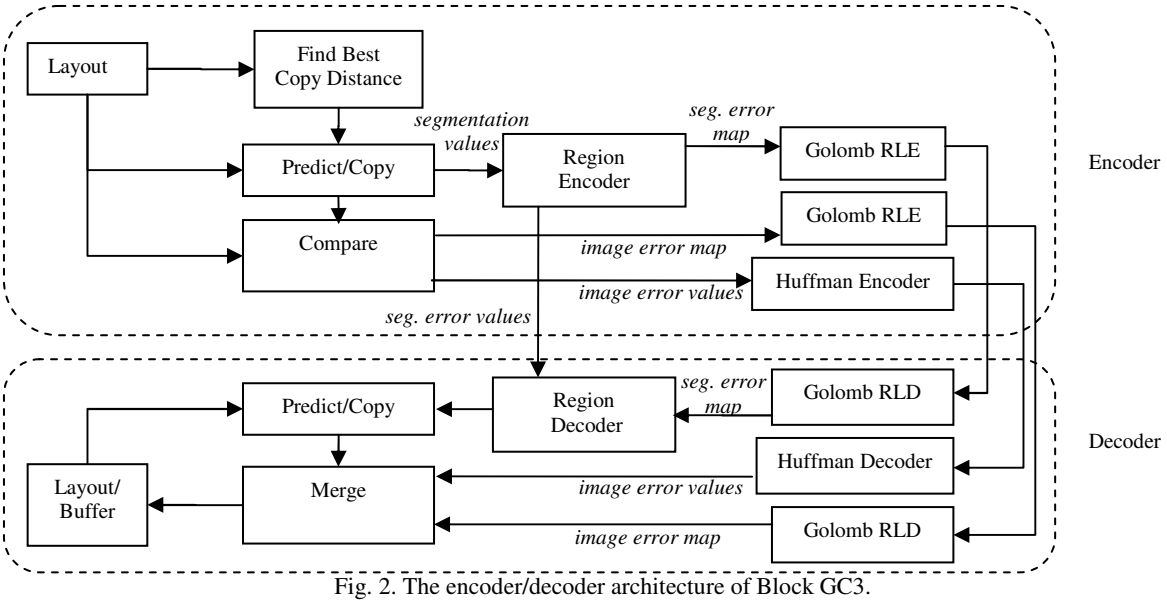
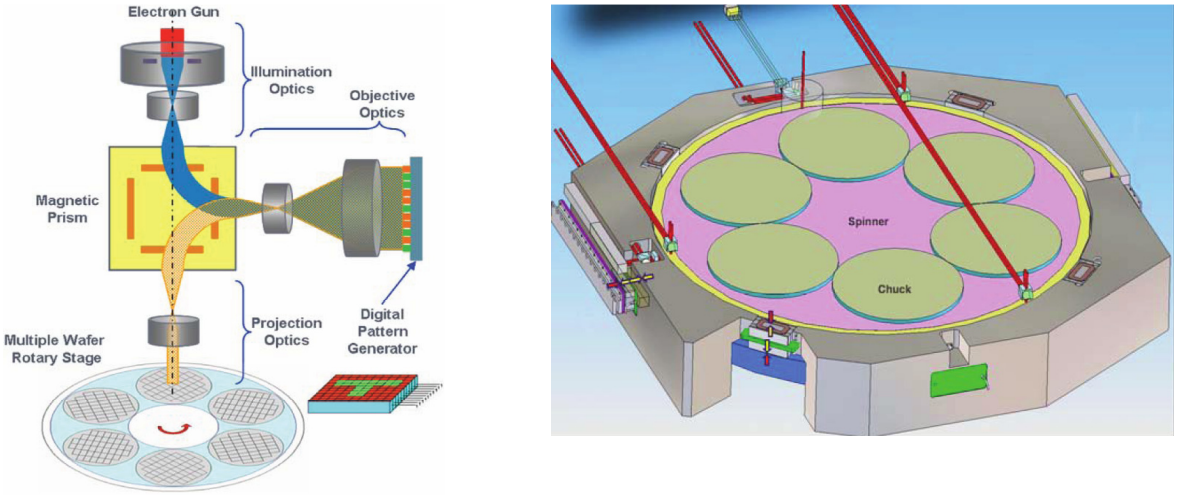


Fig. 2. The encoder/decoder architecture of Block GC3.

Each Block GC3 copy distance points to a location in a history buffer, which stores the most recently decoded pixels. A larger buffer increases the range of possible copy distances, and tends to improve compression efficiency at the expense of higher encoding complexity. In our previous work [4], limited decoder layout area led to the choice of a 1.7 KB buffer, which did not greatly degrade compression efficiency for non-rotated image patterns. However, in the REBL system, layout area considerations allow for a maximum buffer of 40 KB; this extra buffer is potentially valuable, since finding repetition is more challenging for arbitrarily-rotated image patterns.

3. DATA PATH FOR REBL SYSTEM

The REBL system is visualized in Fig. 3(a) and detailed in [5][6]. REBL's goal is to produce the high resolution of electron-beam lithography while maintaining throughputs comparable to those of today's optical lithography systems. The Digital Pattern Generator (DPG) uses reflective electron optics to constantly shape the electron beam as it scans across the wafers, which are located on a rotary stage shown in Fig. 3(b). This thesis focuses on the data delivery path of the REBL system, which constrains the compression hardware implementation. As shown in Fig. 4, the compressed layout data is decoded by Block GC3 decoders in parallel, and then fed into the DPG, which can be located on the same chip as the decoder. In order to meet the required minimum wafer layer throughput of the REBL system, namely 5-7 wafer layers per hour (WPH), given the data rate of the available optical input data link of about 10Gb/s/link, a required minimum compression ratio of 5 is projected.



(a) (b)

Fig. 3. (a) Block diagram of the REBL Nanowriter, (b) detailed view of the rotary stage. [5]

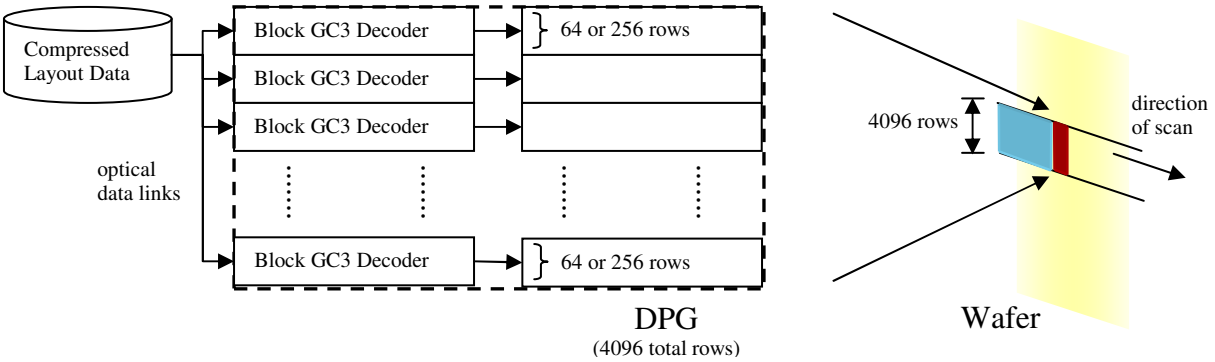


Fig. 4. The data-delivery path of the REBL system.

In the REBL system architecture, similar to the architecture presented in [1], every data path can be handled independently, with its own input data link. Moreover, in the REBL system, the DPG reads layout patterns from the decoders in a column-by-column fashion. Every decoder provides data for a set number of pixel rows: either 64 or 256 rows. The number of columns in each compressed 64- or 256-row "image" effectively can be thought of as being infinite, since the writing system runs continuously until the entire wafer is written. For testing purposes, we restrict the number of columns to either 1024 or 2048. Properties of the test layout images are listed in Table 1.

Table 1. Properties of the test layout images.

Image Size	64×1024, 64×2048, 256×1024, 256×2048
Pixel Value	0-31 (5 bit)
Tilting Angle	25, 35, 38

Each image pixel can take on one of 32 gray levels, in order to guarantee a 1 nm edge placement. In addition, due to the unique rotating writing stage of the REBL system, shown in Fig. 3, the layout images are rotated at arbitrary angles, ranging from 15° to 75°. In our testing set, we have collected layout images of three angles, as listed in Table 1. All the images have undergone E-beam proximity correction (EPC) compatible with the REBL system.

4. COMPRESSION METHOD #1: BLOCK RGC3

This section discusses the modifications that distinguish Block RGC3 from Block GC3. To ensure a feasible hardware implementation for the decoder, modifications have been added mainly to the encoding process, while keeping the decoding process as simple as possible. As briefly mentioned in Sections 4.1 and 4.2 and detailed in [14], a diagonal copying algorithm and a smaller block size allow repetition in rotated layouts to be better exploited. A region-growing technique described in Section 4.3 adds spatial coherence to the segmentation values, increasing the compression efficiency. In Section 4.6, prediction of image pixels is removed, simplifying the decoder hardware without impacting compression efficiency.

4.1 Modifying the copy algorithm

Fig. 5 describes a modification to Block GC3's copy function, detailed in [14]. To accommodate REBL's arbitrary rotation of layout data, the copy method has been modified to allow the decoder to copy from anywhere within the copy range, at any arbitrary direction and distance. This enables repetition discovery regardless of the layout's angle of rotation.

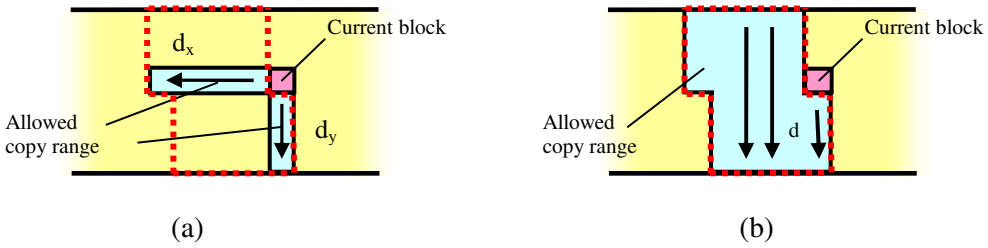


Fig. 5. Two copying methods: (a) Block GC3: only horizontal/vertical copying is allowed. (b) Block RGC3: blocks may be copied from any diagonal direction. In both cases, the dashed areas must be stored in the history buffer [14].

4.2 Decreasing the block size

REBL's arbitrary rotation of layout data also makes it more difficult for Block GC3's copy function to pick up repetition. This is remedied by reducing the block size from 8×8 to 4×4, as described in [14]. Unfortunately, this modification sharply increases the number of segmentation regions, so that the bitstreams transmitting segmentation information make up 50-75% of the compressed bits. In order to

counteract this effect, the encoding process is modified to enforce spatial coherence, i.e., using the same copy distances in adjacent blocks. This is described in the next subsection.

4.3 Growing one-dimensional copy regions

As described in Section 4.6, we have empirically found that for all rotated layouts tested, the "copy" function consistently generates fewer image errors per block than the "predict" function. Thus, for the remainder of this paper, we assume all image blocks to be copied from previously-encoded blocks. In other words, each segmentation value represents a (d_x, d_y) copy distance. Maximizing compression efficiency involves minimizing both the number of image errors and the number of copy distance errors. The latter can be reduced by enforcing spatial coherence among the copy distances of neighboring blocks, as explained shortly. Unfortunately, these two metrics are not independent: minimizing image errors likely reduces the potential for spatial coherence, while enforcing maximum coherence increases image errors.

Enforcing spatial coherence in the segmentation map is made possible by the fact that multiple copy distances often lead to the minimum number of image errors for a given image block. This flexibility can be utilized by creating "regions" consisting of one or more adjacent blocks, each assigned the same copy distance. Regions can be grown in any preferred 2-dimensional way. The optimal region-growing metric is to minimize the total number of 2-D regions, assuming a known fixed number of image errors for each block. However, this problem is NP-complete, even if the number of image errors per block is already known, as we have shown in the Appendix. An alternative is to grow 1-D regions after first assuming each block contains minimal image errors. We have empirically determined that one-dimensional regions of size $1 \times N$ blocks result in higher compression efficiencies than polynomial-time 2-D region-growing heuristics. We refer to dimension N as the "length" of the region, and the region may be oriented either in the horizontal or vertical direction. Since the images associated with the REBL system may require a height as small as 64 pixels, we have chosen to grow regions horizontally in order to maximize N.

The following region-growing heuristic shown in Fig. 6 is solvable in polynomial time and is proven to minimize the number of 1-D regions, if the number of image errors per block is first minimized [12]. Starting at the left of each image row, the longest possible region is assigned. This is iterated, region by region, until the entire row has been assigned:

Let M = image width / block width. Let $k = 1$.

For each non-overlapping $1 \times M$ block image row C_j
    For each block $B_i \in C_j$
        Determine S_i, the set of all copy distances which generates the minimum # image errors
    While ($k \leq M$)
        Find/assign the longest region in C_j with a common copy distance in $\{S_k,...,S_{k+N-1}\}$, $B_k \in C_j$
        $k = k + N$.

Fig. 6. Region-growing algorithm.

To encode the 1-D horizontal segmentation regions, we propose a new prediction method shown in Fig. 7(a), whereby each block's copy distance is predicted to be equal to that of the block to its left. This ensures that the (d_x, d_y) copy distance associated with a given horizontally-oriented region is only sent once. Specifically, only the left-most block in a region is predicted incorrectly. The prediction method for Block GC3 is shown in Fig. 7(b).

(a) (b)

Fig. 7. The block-based prediction method of (a) Block RGC3, (b) Block GC3.

4.4 Determining the optimal block shape

In Section 4.2, we discussed the tradeoffs of changing the block size; in this section, we discuss the optimal block aspect ratio. We assume that layout features are not preferentially oriented in either a horizontal or vertical direction, regardless of layout rotation. Regardless of the region-growing method utilized, we expect square-shaped segmentation regions to maximize the average region size, and therefore improve compression efficiency. This is because the longer dimension of an oblong segmentation region is likely to infringe on neighboring features, thus limiting repetition discovery and compression efficiency. In effect, square segmentation regions minimize the effect of this limitation.

Block GC3 uses square blocks [4] and grows regions in either a horizontal or vertical direction, thus resulting in approximately square-shaped segmentation regions. In contrast, Block RGC3 clearly grows regions using a directional preference, by using identical copy distances for neighboring blocks along the horizontal direction only. For a square block size, the average Block RGC3 region has a greater horizontal width than a vertical height. For example, assuming a 4×4 block size, the average Metal 1 region contains 2.7 blocks, which corresponds to a region size of 4×10.8. Since square-shaped segmentation regions are likely to optimize compression efficiency, one approach is to choose block shapes resulting in more or less square regions. Since Block RGC3 grows segmentation regions in a horizontal direction, this would mean that the block shape resulting in near-square segmentation regions is likely to have smaller width than height.

Table 2 compares compression and encoding time results for various block sizes, using the final Block RGC3 algorithm which includes the region-absorbing method discussed in Section 4.5. Two factors improve compression efficiency. First, the image error rate decreases as the block size decreases, as justified in Section 4.2; the resulting increase in blocks per image area also increases the encoding time. Second, a tall, narrow block aspect ratio such as 7×1 likely yields a more square-shaped average region, which, in turn, increases the average region size. As verified in Table 2, block shapes such as {6,7} × {1,2}, resulting in more square-shaped average regions are likely to have a larger average region size, and hence fewer regions, and hence a higher compression ratio.

For our final design, we have decided to use a 5×3 block size, due to its high compression ratio and low encoding time. Even though 7×1 and 6×1 block sizes slightly improve compression efficiency in the example shown, the encoding times for these are nearly twice as much as for 5×3; furthermore, the compression gains are lost if the buffer size is reduced or if a Via layer is compressed instead of Metal 1.

Table 2. Performance comparison of various block sizes, using a 1024×256 25°-oriented Metal 1 image and a 40 KB buffer.

Block Size	4×4	5×3	3×5	7×2	2×7	6×2	2×6	4×3	3×4	7×1	6×1
Average region shape (pixels)	4×10.8	5×9.3	3×12.6	7×7.1	2×15.3	6×7.9	2×15.0	4×10.5	3×12.4	7×6.6	6×7.5
Average region size (pixels)	43.2	46.5	37.8	49.7	30.6	47.4	30.0	42.0	37.2	46.2	45.0
Number of regions	6065	5572	6878	5185	8555	5412	8706	6198	7020	5557	5761
Image error rate (%)	2.67	2.60	2.64	2.70	2.83	2.41	2.55	2.32	2.37	2.08	1.84
Compression ratio	6.79	7.12	6.32	7.24	5.46	7.32	5.50	6.93	6.40	7.39	7.43
Encoding time (seconds)	182	180	189	191	213	209	230	202	226	295	334

Assuming a 4×4 block size and comparing with Section 4.1's results, growing regions as per Section 4.3 increases the number of correctly-predicted copy distances from 35% to 55%, while the image error rate remains unchanged at 2.18%. Now using a 5×3 block size, 59% of the blocks' copy distances are correctly predicted, while the image error rate is reduced to 2.09%. Thus, growing regions and utilizing a 5×3 block size both improve compression efficiency, as shown in Table 3. A sample of 25° Metal 1 copy regions is shown in Fig. 8 and Fig. 9(a), where each copy distance is randomly assigned a different color. The effectiveness of region-growing is loosely proportional to the number of candidate copy distances per block which generate minimal image errors. A cumulative distribution function of this is shown in Fig. 10; notice that 71% of blocks in the given layout have at least ten optimal copy distances, assuming a 40KB buffer size. As this number increases, the probability of finding a common copy distance among neighboring blocks also increases, thereby resulting in larger 1-D regions.

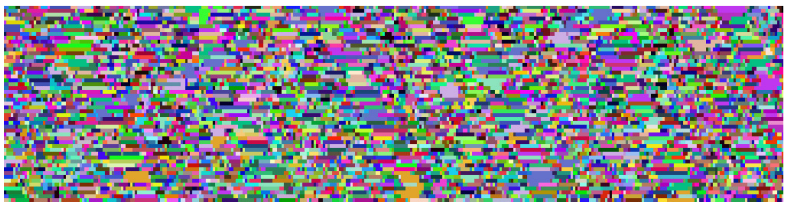
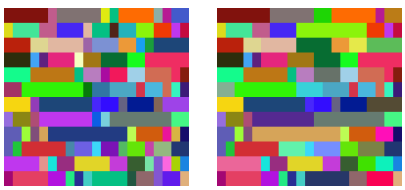


Fig. 8. Segmentation map, utilizing 1-D region-growing.



(a) (b)

Fig. 9. (a) Zoomed in portion of Fig. 8. (b) Regions after applying the region-absorbing method.

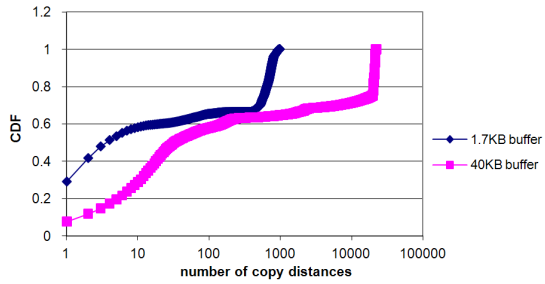


Fig. 10. Cumulative distribution function (cdf) of the number of copy distances per block which generate minimal image errors.

4.5 Combining neighboring regions

As mentioned in Section 4.3, maximizing the compression ratio requires a tradeoff between minimizing image errors and maximizing spatial coherence of copy distances. However, our 1-D region-growing method minimizes image errors as much as possible, at the expense of reducing spatial coherence. This constraint can be alleviated by adding a post-processing step, which combines neighboring regions while accepting a small penalty of image errors. The encoder judges this tradeoff using two parameters: α_1, the marginal decrease of transmitted bits resulting from one fewer segmentation error, and α_2, the marginal increase of transmitted bits resulting from one additional image error. α_1 and α_2 are empirically calibrated, since these marginal changes depend on how effectively Huffman codes and Golomb run-length codes compress the segmentation/image error maps and values. For each image row, we apply the following heuristic to optimally combine the first five regions by finding combinations which maximize the expression: $\alpha_1 \times$ [*# regions absorbed*] $- \alpha_2 \times$ [*# image errors incurred*]; if no beneficial combinations are found, the set of five observed regions is shifted by three. We iterate on this until the entire row has been inspected. Region absorbing is only effective if $\alpha_1 > \alpha_2$; otherwise a single additional image error more than negates the benefit of combining two regions. Empirically, we find $\alpha_1 = 14$ and $\alpha_2 = 10$ to optimize for 25°-rotated Metal 1, with a 40 KB buffer. By optimizing *five* regions and shifting by *three*, the new set of five regions includes two regions from the previous set; this allows nearly all beneficial absorptions of one or two regions to be carried out, including those straddling the boundary of each five-region set. Fig. 11 shows the pseudo-code for our proposed region-absorbing algorithm.

Let M = image width / block width. Let $k = 1$.

For each non-overlapping $1 \times M$ block image row C_j, containing regions $R_{j,1}$ through $R_{j,m}$
  While ($k < m$)
    For the sub-row $D_{j,k} = \{R_{j,k},\ldots,R_{j,k+4}\}$
      For each region $R_{j,k+n} \in D_{j,k}$
        For each "optimal" copy distance for $R_{j,k+n}$, as determined in Fig. 6
          List costs associated with absorbing one or more regions from $D_{j,k}$ into $R_{j,k+n}$
            - break if the number of image errors exceeds a known threshold
      If the "best" cost is beneficial, re-assign regions accordingly.
      Else, $k = k + 3$.

Fig. 11. Region-absorbing algorithm.

Fig. 9(b) shows how Fig. 9(a) has been modified by the region-absorbing algorithm. Specifically, the number of correctly-predicted copy distances has increased from 59% to 68%, but the image error rate has increased from 2.09% to 2.60%. This tradeoff has improved the compression efficiency overall, as shown in the right column of Table 3.

Table 3. Compression efficiency of Block RGC3, compared with other lossless compression algorithms.

	w/ 1-D region-growing, 4×4 Section 4.3		w/ 1-D region-growing, 5×3 Section 4.4		w/ region-absorbing, 5x3 Section 4.5	
Image size	Avg.	Min.	Avg.	Min.	Avg.	Min.
64×1024	5.73	5.53	5.87	5.70	6.19	6.00
64×2048	6.13	5.98	6.28	6.16	6.60	6.46
256×1024	6.51	6.50	6.76	6.75	7.12	7.12
256×2048	6.67	6.67	6.91	6.91	7.29	7.29

4.6 Removing pixel-based prediction

Segmentation values in Block GC3 are either copy distances or a copy distance of "0", which instructs the decoder to predict the pixel values based on the properties of neighboring pixels. For every layout sample available for testing, using a 5×3 block size, the predict function is never utilized because the copy function consistently generates fewer image errors per block. This section justifies the removal of prediction from Block GC3, in order to simplify the decoder's hardware implementation.

Fig. 12(a) shows Block GC3's prediction method. Pixel "*z*" is predicted with the aid of pixels "*a*", "*b*", and "*c*". Fig. 12(b) shows an example of a 0°-rotated layout image this prediction method is designed to compress. The prediction method handles this structure nicely, correctly predicting any horizontal or vertical feature boundaries as long as neighboring boundary pixel values are equal. This follows from: if

$a = c$, then $z = b$; additionally, if $b = a$, then $z = c$. Thus, almost all errors occur near the corners of each feature.

(a) (b) (c)

Fig. 12. (a) Pixel-based prediction method used by Block GC3. (b) Rasterized layout assumed by Block GC3. (c) Rasterized layout assumed in this work.

Contrast Fig. 12(b) with Fig. 12(c), in which the layout has been rotated by an arbitrary angle. Neighboring edge pixels have different gray values and, in fact, do not even follow a set repeating pattern, but are dependent on the angle of the layout as well as any second-order proximity-correcting effects. This renders edge pixels effectively unpredictable, if a simple prediction method such as in Fig. 12(a) is used. The copy function is guaranteed to perform at least as well as prediction if all boundary pixels are unpredictable and if all non-boundary pixels can be correctly determined using the copy function. The former is actually very lenient for the given prediction method, since pixels adjacent to boundaries are also frequently predicted incorrectly. The latter is true given three conditions:

(1) All foreground (and background) pixels have equal values. This is nearly true for the layout shown in Fig. 12(c).

(2) For a given block, a (d_x, d_y) copy distance can be found such that, if boundary pixels are set equal to "1" and non-boundary pixels are set equal to "0", the image block and the copied block are equal. This should not be difficult to achieve since, for a given angle of rotation, there is a limited number of expected boundary "shapes" that should occur within a 5×3 block.

(3) For this same (d_x, d_y) distance, the copied block's assignment of either "foreground" or "background" on either side of each edge matches that of the image block. This condition

depends on (1); it should, at worst, divide the number of possible copy values by two, since if "foreground" is chosen for one side of a boundary, the other side must be labeled "background".

Our empirical results support this argument. Out of 17338 3×5 blocks in the 256x1024 image, Block GC3 chooses copy over prediction in each case, on the basis of minimizing image errors. Using the copy function exclusively results in a 2% image error rate, while using only the prediction method from Fig. 12(a) results in a 27% image error rate. Attempts to adapt the prediction to the layout in Fig. 12(c) reduce the error rate only to 22%. Thus, options include removing prediction entirely, leaving prediction functionality intact but simply not using it, or else pursuing a significantly more intricate prediction method capable of predicting edge pixels, but at the expense of complicating the hardware implementation. We have chosen to remove prediction completely, in hopes of simplifying the decoder's hardware implementation.

5. BLOCK RGC3 RESULTS

5.1 Compression Results

Table 4 shows the average compression efficiency for several layout samples of different buffer and image sizes. In total, three block sizes, three buffer sizes, three image sizes, and over four different layers were tested. Different wafer layers have significantly different compression ratios, ranging from 18.6 for the via layer, to 13.2 for the poly layer, to 7.3 for Metal 1. From most significant to least significant, the factors affecting compression efficiency are: wafer layer, buffer size, block size, image size, and angle of rotation.

The 5×3 block size consistently outperforms the 4×4 block size, independent of the buffer size, the layout, or its angle of rotation. Note the 8×8 block size actually performs best for the Via layer, because of its sparse pattern density.

The 38° Metal 1 layout has been tested in an effort to minimize pixel repetition, as per the $\tan^{-1}(b/a)$ approximation discussed in [14]. Since $38° \approx \tan^{-1}(25/32)$, we expected the integers in this ratio to be too large for a copy distance of $(d_x, d_y) = (32, 25)$ to be utilized. However, due to REBL's E-beam proximity correction, repetition was often found using $(d_x, d_y) = (9, 7)$, where $\tan^{-1}(7/9)$ is a less accurate approximation for 38°. As a result, 38° compression results are not much different from results using rotations of 25° and 35°, which can be approximated by $\tan^{-1}(7/15)$ and $\tan^{-1}(7/10)$, respectively.

256×1024 images outperform 64×2048 images, partially due to their larger size. In addition, the buffer is more square-shaped for 256×1024 images, assuming the buffer size is greater than 10KB. This increases the overall likelihood of finding repetition at an arbitrary diagonal direction. While encoding the first blocks of an image, very few copy distances are available to choose from; this detrimental effect dies out as the history buffer becomes "full", and is less prominent for larger image sizes. Thus, 256×2048 images perform still better than 256×1024 images. In the REBL system, the expected image width approaches infinity, resulting in a further increase in compression efficiency.

Table 4. Average Block RGC3 compression efficiency for various layouts, image sizes, block sizes, buffer sizes.

Image size	Block size	Buffer size	Metal 1			Metal 1b[1]	Poly	Via	
			25°	35°	38°	25°	35°	25°	35°
64×2048	5×3	1.7KB	4.92	5.37	5.14	--	8.49	13.14	12.67
256×1024	5×3	1.7KB	5.09	5.43	5.33	8.55	8.47	13.58	13.17
256×2048	5×3	1.7KB	5.10	5.45	5.35	--	8.51	13.62	13.22
64×2048	5×3	20KB	6.54	6.68	6.63	--	11.17	15.31	15.40
256×1024	5×3	20KB	6.91	7.08	7.11	14.06	12.50	16.14	16.00
256×2048	5×3	20KB	7.01	7.20	7.22	--	12.77	16.35	16.22
64×2048	5×3	40KB	6.60	6.79	6.71	--	11.91	15.86	16.11
256×1024	5×3	40KB	7.12	7.23	7.34	14.87	12.80	17.05	17.27
256×2048	5×3	40KB	7.29	7.41	7.50	--	13.17	17.45	17.79
64×2048	4×4	1.7KB	4.93	5.38	5.22	--	8.64	13.21	12.88
256×1024	4×4	1.7KB	4.99	5.32	5.22	8.40	8.24	13.46	13.02
256×2048	4×4	1.7KB	5.01	5.33	5.23	--	8.29	13.54	13.10
64×2048	4×4	20KB	6.37	6.57	6.57	--	10.77	15.36	15.51
256×1024	4×4	20KB	6.61	6.86	6.84	13.57	11.85	15.91	15.74
256×2048	4×4	20KB	6.71	6.96	6.94	--	12.12	16.14	15.99
64×2048	4×4	40KB	6.40	6.69	6.60	--	11.43	15.84	16.15
256×1024	4×4	40KB	6.81	6.98	7.05	14.29	12.09	16.76	16.94
256×2048	4×4	40KB	6.97	7.14	7.19	--	12.44	17.16	17.43
64×2048	8×8	1.7KB	3.97	4.55	4.47	--	7.61	13.54	13.24
256×1024	8×8	1.7KB	4.02	4.51	4.5	7.71	7.34	13.92	13.41
256×2048	8×8	1.7KB	4.03	4.52	4.51	--	7.37	13.96	13.48
64×2048	8×8	20KB	5.23	5.75	5.76	--	9.88	16.29	16.56
256×1024	8×8	20KB	5.46	6.04	5.99	12.82	11.09	17.01	16.92
256×2048	8×8	20KB	5.53	6.12	6.07	--	11.32	17.24	17.19
64×2048	8×8	40KB	5.38	5.96	5.91	--	10.59	16.94	17.36
256×1024	8×8	40KB	5.75	6.26	6.30	13.66	11.52	18.09	18.48
256×2048	8×8	40KB	5.88	6.39	6.43	--	11.86	18.55	19.08

1 Only one 720×256 image is available for Metal 1b. The “Metal 1” and “Metal 1b” image samples are taken from different locations of the Metal 1 layer.

Table 5. Average Block GC3 compression efficiency for various layouts, image sizes, block sizes, buffer sizes.

Image size	Block size	Buffer size	Metal 1			Metal 1b[1]	Poly	Via	
			25°	35°	38°	25°	35°	25°	35°
2048×64	3×5	1.7KB	3.04	3.55	3.70	--	4.83	10.11	9.72
1024×256	3×5	1.7KB	3.05	3.42	3.64	4.77	4.78	9.85	9.57
2048×256	3×5	1.7KB	3.05	3.42	3.65	--	4.79	9.88	9.59
2048×64	3×5	20KB	3.46	3.90	3.97	--	5.37	10.71	10.10
1024×256	3×5	20KB	3.28	3.62	3.82	4.99	4.96	10.04	9.75
2048×256	3×5	20KB	3.29	3.63	3.83	--	4.98	10.09	9.77
2048×64	3×5	40KB	3.54	4.00	4.03	--	5.68	11.00	10.44
1024×256	3×5	40KB	3.39	3.70	3.88	5.25	5.11	10.22	9.86
2048×256	3×5	40KB	3.42	3.73	3.91	--	5.16	10.26	9.90
2048×64	4×4	1.7KB	3.05	3.58	3.74	--	4.85	10.23	9.91
1024×256	4×4	1.7KB	3.08	3.47	3.71	4.83	4.85	10.05	9.79
2048×256	4×4	1.7KB	3.08	3.48	3.71	--	4.85	10.07	9.81
2048×64	4×4	20KB	3.51	3.99	4.07	--	5.44	10.86	10.35
1024×256	4×4	20KB	3.33	3.68	3.90	5.07	5.05	10.26	9.97
2048×256	4×4	20KB	3.34	3.70	3.91	--	5.07	10.29	10.00
2048×64	4×4	40KB	3.61	4.11	4.15		5.77	11.14	10.68
1024×256	4×4	40KB	3.38	3.79	3.97	5.39	5.22	10.42	10.10
2048×256	4×4	40KB	3.49	3.82	4.00	--	5.26	10.47	10.15
2048×64	8×8	1.7KB	3.13	3.77	3.94	--	5.13	10.81	10.44
1024×256	8×8	1.7KB	3.19	3.75	4.00	5.16	5.24	10.76	10.43
2048×256	8×8	1.7KB	3.19	3.76	4.00	--	5.25	10.79	10.45
2048×64	8×8	20KB	3.37	3.98	4.14	--	5.70	11.57	11.07
1024×256	8×8	20KB	3.30	3.83	4.07	5.33	5.39	11.08	10.67
2048×256	8×8	20KB	3.30	3.84	4.07	--	5.41	11.12	10.70
2048×64	8×8	40KB	3.44	4.07	4.20	--	6.00	11.93	11.47
1024×256	8×8	40KB	3.36	3.88	4.11	5.59	5.55	11.27	10.86
2048×256	8×8	40KB	3.37	3.90	4.12	--	5.59	11.34	10.92

Table 5 is identical to Table 4, except that images are compressed using the original Block GC3 algorithm. In terms of compression efficiency, Block RGC3 consistently outperforms Block GC3. Since a horizontal/vertical-only copy range does not adequately pick up repetition from rotated layouts, further extending Block GC3's copy range along the horizontal and vertical directions by increasing its buffer size only slightly improves efficiency. In fact, a larger buffer can actually degrade performance, because increasing the allowed copy range increases the number of bits per segmentation value, without substantially reducing the number of image errors.

Block GC3's compression efficiency is best for 64×2048 image sizes, assuming a 40 KB buffer. Since the buffer size = $d_{x,max} \times d_{y,max}$ but the allowed copy range is $d_{x,max} + d_{y,max}$, the number of allowed copy distances increases as the aspect ratio of the buffered image area becomes more skewed. Since $d_{y,max} \approx$ the image height, 64×2048 images tend to create a short, wide buffer, thus increasing the number of possible copy distances for each block and improving Block GC3 compression efficiency, as compared with 256×2048 images. Note, however, that increasing the allowed copy range also increases the encoding time.

Unlike Block RGC3, an 8×8 block size usually produces the best results for Block GC3. Because Block GC3 does not methodically grow regions of blocks, the number of segmentation values is substantial. This can be alleviated by using an 8×8 block size or by entropy-coding the segmentation values; results for the former are shown in Table 5, while results for the latter approach are not shown here. Also, since 1-D region-growing is not used in Block GC3, an oblong block size is no longer necessary; thus, a 4×4 block size outperforms a 5×3 block size by around 3%, as also shown in Table 5.

Table 6. Average compression efficiency for various layouts, image sizes, block sizes, buffer sizes, and compression algorithms.

Image size	Block size	Buffer size	Metal 1, 25°					Via, 25°				
			Block RGC3	Block GC3	ZIP (30 KB)	BZIP2 (900 KB)	JPEG-LS (2.2 KB)	Block RGC3	Block GC3	ZIP (30 KB)	BZIP2 (900 KB)	JPEG-LS (2.2 KB)
64×2048	5×3	1.7KB	4.92	3.04	3.23	3.95	0.95	13.14	10.11	10.64	14.24	3.91
256×2048	5×3	1.7KB	5.10	3.05	3.43	4.68	0.97	13.62	9.88	11.68	15.98	4.03
64×2048	5×3	40KB	6.60	3.54	3.23	3.95	0.95	15.86	11.00	10.64	14.24	3.91
256×2048	5×3	40KB	7.29	3.42	3.43	4.68	0.97	17.45	10.26	11.68	15.98	4.03
64×2048	8×8	1.7KB	3.97	3.13	3.23	3.95	0.95	13.54	10.81	10.64	14.24	3.91
256×2048	8×8	1.7KB	4.03	3.19	3.43	4.68	0.97	13.96	10.79	11.68	15.98	4.03
64×2048	8×8	40KB	5.38	3.44	3.23	3.95	0.95	16.94	11.93	10.64	14.24	3.91
256×2048	8×8	40KB	5.88	3.37	3.43	4.68	0.97	18.55	11.34	11.68	15.98	4.03

Table 6 compares the compression efficiency of Block RGC3 with that of Block GC3, ZIP, BZIP2, and JPEG-LS, for both Metal 1 and Via layers [7][8][10][11]. Block RGC3 and Block GC3 are tested using buffer sizes of 1.7 KB and 40 KB, while ZIP, BZIP2, and JPEG-LS have constant buffer sizes of 30 KB, 900 KB, and 2.2 KB, respectively. Block RGC3 consistently outperforms both ZIP and JPEG-LS, even

using a 1.7 KB buffer size. BZIP2 outperforms Block RGC3 only when compressing the via layer using a 1.7 KB buffer. However, impractical hardware implementation and high buffer requirements prevent BZIP2 from being integrated into the REBL system.

Removal of Huffman Codes

If the complexity of the decoder's hardware is too high, because of area constraints or other factors, it may become necessary to simplify the compression algorithm. Block GC3 typically Huffman codes the image error values, as shown in Fig. 2 [9]. However, as shown in Table 7, we have found that eliminating Huffman coding reduces compression efficiency only slightly, i.e., 0-2%. Accordingly, removing the Huffman decoder from the decoder hardware reduces the area by 4.4% [4].

Table 7. Average Block RGC3 compression ratios, assuming a 40KB buffer, both with and without Huffman coding.

Image size	Block size	Use Huffman?	Metal 1			Metal 1b[1]	Poly	Via	
			25°	35°	38°	25°	35°	25°	35°
64×2048	5×3	Yes	6.60	6.79	6.71	--	11.91	15.86	16.11
256×1024	5×3	Yes	7.12	7.23	7.34	14.87	12.80	17.05	17.27
256×2048	5×3	Yes	7.29	7.41	7.50	--	13.17	17.45	17.79
64×2048	5×3	No	6.51	6.68	6.59	--	11.66	15.90	16.21
256×1024	5×3	No	7.02	7.11	7.20	14.71	12.48	17.05	17.33
256×2048	5×3	No	7.18	7.28	7.36	--	12.84	17.43	17.82

5.2 Encoding Complexity Results

The total encoding time required to compress a given image is dominated by three main factors: (1) determining a list of optimal copy distances for each block, allowing diagonal copies, (2) choosing a copy distance for each block from this list such that the total number of regions is minimized, and (3) absorbing neighboring regions.

Contribution #1 essentially requires each image pixel to be compared with the pixel associated with each available copy distance. Thus, this portion of the encoding time is independent of the input image pattern. The image is both encoded and decoded in a column-by-column fashion. For Block RGC3, the number of

possible copy distances per block is $d_{x,max} \times d_{y,max}$, where $d_{y,max}$ typically equals the height of the image and $d_{x,max}$ = buffer_size/$d_{y,max}$; in contrast, for Block GC3's horizontal/vertical copying, the copy candidate range is $d_{x,max} + d_{y,max}$. Due to extra computational overhead which is inversely proportional to the block size, we have empirically found encoding time to vary with $1/\beta + 1/(H \times W)$, where $H \times W$ represents the block size and $\beta \approx 10$ The β-dependent and block size-dependent factors equally affect encoding time when $H \times W = \beta$. Thus, Block RGC3 encoding time for a given image area is proportional to

$$O\left(d_{x,\max} d_{y,\max}\left(\frac{1}{\beta}+\frac{1}{HW}\right)\right) = O\left(buffer\_size\left(\frac{1}{\beta}+\frac{1}{HW}\right)\right).$$

Contribution #2 depends on $\overline{d_{matches,block}}$, the average number of copy distances per block resulting in minimal image errors. Contribution #3 depends on $\overline{N}$, the average number of blocks per region, and $\overline{d_{matches,region}}$, the average number of copy distances common to all the blocks in an entire region which result in minimal image errors. These three parameters depend on the layout image and the buffer size. Regions are grown by searching through each block's list of copy distances until the longest region is found. Similarly, regions are absorbed by applying each region's list of copy distances to its neighboring regions, to find the best possible absorptions. Encoding time increases as the block size decreases, since each block corresponds to a smaller image area. Thus, for a given image area, Contribution #2's encoding complexity is proportional to

$$O\left(\frac{\overline{d_{matches,block}}}{HW}\right),$$

while Contribution #3's encoding complexity is proportional to

$$O\left(\frac{\overline{d_{matches,region}}}{\overline{N}HW}\right).$$

Note that encoding time increases substantially if $\overline{d_{matches,block}}$ is very high; this typically occurs in images containing large amounts of background area, such as the Via layer.

Table 8 shows examples of software encoding times for various layouts and encoding parameters, using a 2.66GHz Intel Xeon processor with 2.75GB RAM. As expected, encoding times are proportional to image size and inversely proportional to block size. Notice that the easily-compressible via layer, composed of 96% background pixels, requires more time to encode because of its high $\overline{d_{matches,block}}$. Additionally, larger buffer sizes increase the number of possible copy distances per block; this increases $\overline{d_{matches,block}}$ and $\overline{d_{matches,region}}$, resulting in larger encoding times.

The encoding and decoding times for Block GC3 are shown in Table 9 as a comparison. The vast increase in Block RGC3 encoding times is due to the use of diagonal copying, which greatly increases the allowed copy range, as shown in Fig. 5. In the worst case, if $d_{y,max} = d_{x,max}$ = image height, diagonal copying increases the allowed copy range by a factor of approximately half the image height, in pixels, e.g., 32 or 128 for an image height of 64 or 256, respectively.

Table 8. Block RGC3 encoding times in seconds for various layouts, image sizes, block sizes, and buffer sizes.

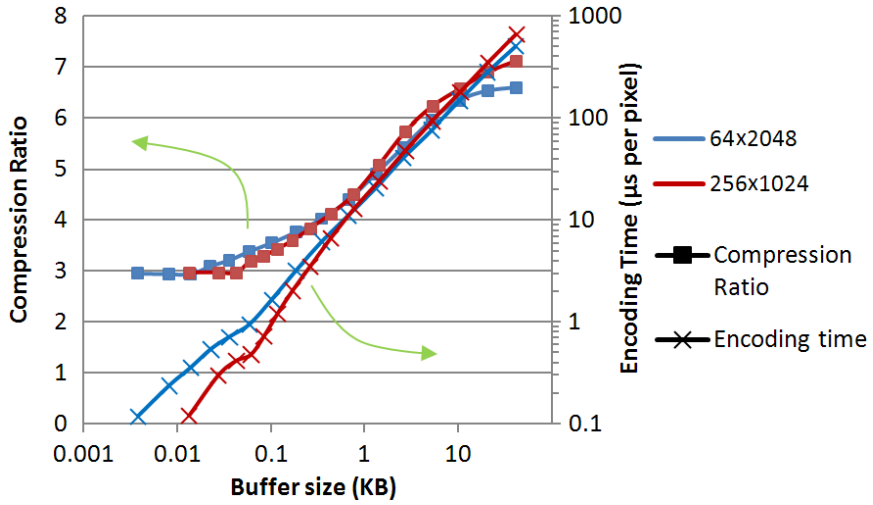
			Contribution #1		Contribution #2		Contribution #3		Total encoding time	
Image size	Block size	Buffer size	Metal 1 25°	Via 25°	Metal 1 25°	Via 25°	Metal 1 25°	Via 25°	Metal 1 25°	Via 25°
64×2048	5×3	20KB	35.3	35.4	1.6	5.4	0.2	0.6	37.0	41.4
256×1024	5×3	20KB	87.7	92.9	3.9	15.1	0.4	1.2	92.1	109.2
64×2048	5×3	40KB	63.6	66.8	2.4	11.0	0.2	0.9	66.2	78.7
256×1024	5×3	40KB	166.3	184.0	7.0	40.9	0.5	2.0	173.9	226.9
64×2048	8×8	20KB	28.2	28.4	0.2	2.5	0.2	0.8	28.6	31.7
256×1024	8×8	20KB	66.9	62.8	0.4	4.9	0.4	2.1	67.7	69.8
64×2048	8×8	40KB	49.1	49.1	0.3	4.3	0.2	1.2	49.7	54.6
256×1024	8×8	40KB	124.5	116.2	0.7	9.6	0.5	3.3	125.7	129.1

Table 9. Encoding and Decoding Times: Comparison between Block RGC3 and Block GC3.

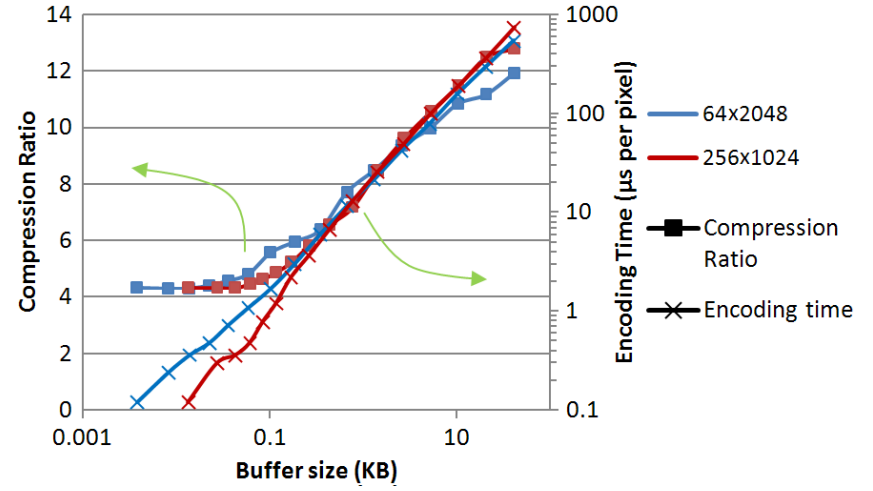
			Encoding Time (sec) (Block GC3)		Encoding Time (sec) (Block RGC3)		Decoding Time (ms) (Block GC3)		Decoding Time (ms) (Block RGC3)	
Image size	Block size	Buffer size	Metal 1 25°	Via 25°	Metal 1 25°	Via 25°	Metal 1 25°	Via 25°	Metal 1 25°	Via 25°
64×2048	5×3	20KB	0.58	0.55	37.0	41.4	6.7	5.5	6.3	4.7
256×1024	5×3	20KB	0.69	0.63	92.1	109.2	16.0	11.8	12.5	12.7
64×2048	5×3	40KB	1.00	0.92	66.2	78.7	6.1	5.0	6.3	5.3
256×1024	5×3	40KB	0.92	0.88	173.9	226.9	13.7	11.2	12.0	10.8
64×2048	8×8	20KB	0.52	0.45	28.6	31.7	6.8	4.8	6.1	4.8
256×1024	8×8	20KB	0.53	0.47	67.7	69.8	13.8	10.2	11.7	10.8
64×2048	8×8	40KB	0.859	0.734	49.7	54.6	6.8	4.5	5.8	5.5
256×1024	8×8	40KB	0.766	0.672	125.7	129.1	14.7	10.8	11.6	10.6

The effect of buffer size on compression efficiency and encoding complexity is visualized in Fig. 13. A 5×3 block size is assumed. Even though higher buffer sizes result in steadily higher compression ratios, this comes at a high price. First, larger buffer sizes result in memory occupying more decoder chip area, which is likely severely constrained to begin with. Second, Block RGC3 encoding time is linearly proportional to the buffer size. Although the encoding process may be completed offline in software, there are practical limits to how long this process may take. For example, using Block RGC3 with a 40KB buffer to compress a 20mm × 10mm 25° via layer in 64×2048 pixel segments would require 7.9 CPU years, assuming each pixel corresponds to a 22nm × 22nm area. If a 1.7KB buffer is used instead, computing time is reduced to 110 CPU days; this is clearly practical if several multiple-core processors are used in parallel.

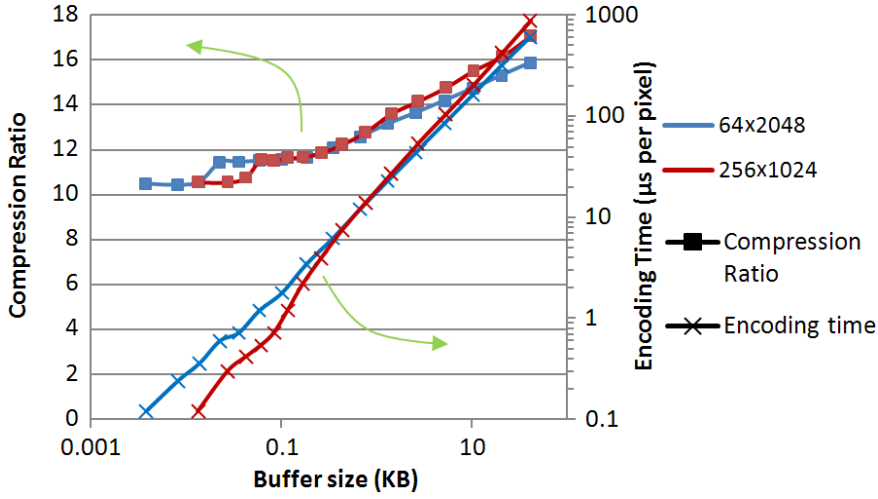
For buffer sizes less than 200 bytes, the history buffer is no longer the dominant contribution to the overall buffer size. Due to hardware implementation constraints, a larger image requires a larger buffer size in order to implement a given number of copy distances. This accounts for the compression and encoding time discrepancies between 64×2048 and 256×1024 image sizes.



(a)



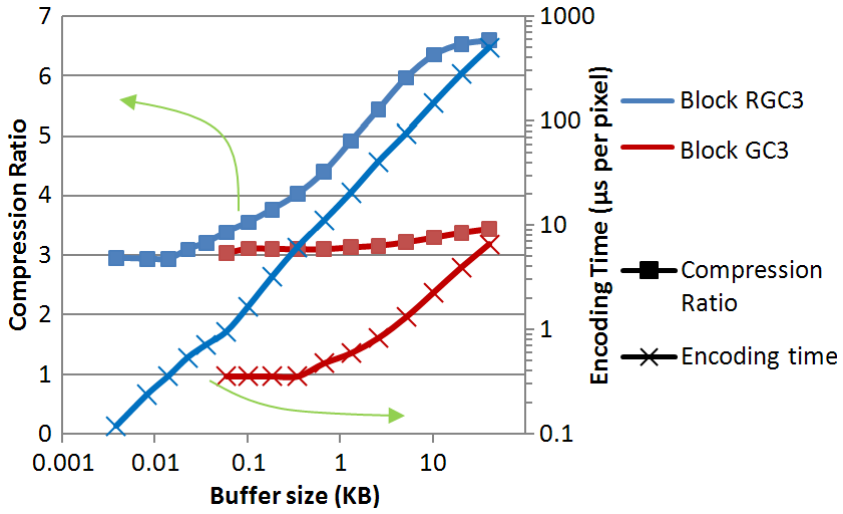
(b)



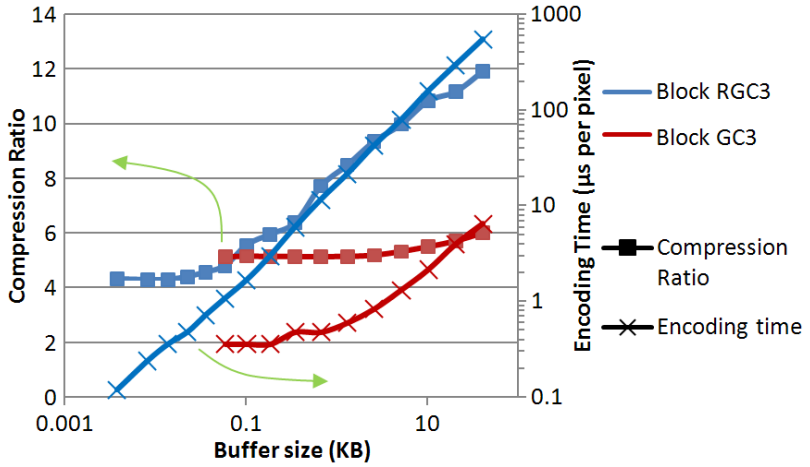
(c)

Fig. 13. Average compression ratios and encoding times vs. buffer size for (a) 25° Metal 1, (b) 35° Poly, (c) 25° Via

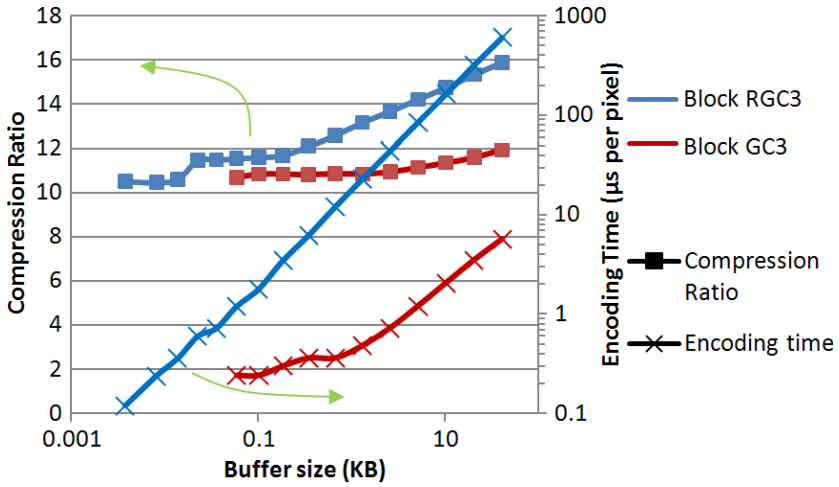
Block RGC3 and Block GC3 compression efficiency and encoding complexity, as a function of buffer size, are compared in Fig. 14. As shown, Block RGC3 is capable of producing significantly higher compression efficiencies than Block GC3. Also, for a given compression ratio, Block RGC3 requires less encoding time and buffering in almost every instance. The only exception occurs when compressing Via and Poly using a 60 byte buffer size, at which Block RGC3 and Block GC3 have an identical 64×1-pixel copy range. At such a small copy range, region-growing is somewhat ineffective; thus, the main design distinction is Block GC3's 8×8 block size, which yields similar compression efficiency but requires four times lower encoding time than Block RGC3's 5×3 block size.



(a)



(b)



(c)

Fig. 14. Avg. compression ratios and encoding times vs. buffer size for 64×2048 (a) 25° Metal 1; (b) 35° Poly; (c) 25° Via

The largest Block RGC3 buffer size shown in Fig. 14, 40 KB, allows each block to be assigned one of 2^{16} copy distances; the smallest buffer size represents the trivial case in which each pixel is always copied from the pixel immediately below it. For a given compression ratio, we have found that the encoding time can be reduced by roughly a factor of two with respect to Fig. 14 if the copy range is not searched thoroughly. Most of this reduction results from, whenever possible, blindly picking the first discovered copy distance resulting in zero image errors. Further reduction results from checking, at most, a small percentage of the available copy range, specifically those copy distances in close proximity to the copy range from Fig. 5(a), but rotated to match the wafer pattern's current angle of orientation.

6. COMPRESSION METHOD #2: VECTOR QUANTIZATION

The process of encoding quantized data points one by one is called scalar quantization. If correlated data points can be grouped into "vectors", patterns within each group can be exploited and compression can be achieved by encoding entire groups at once. This method is called vector quantization (VQ) [15] [16].

6.1 Adapting VQ to lossless image compression

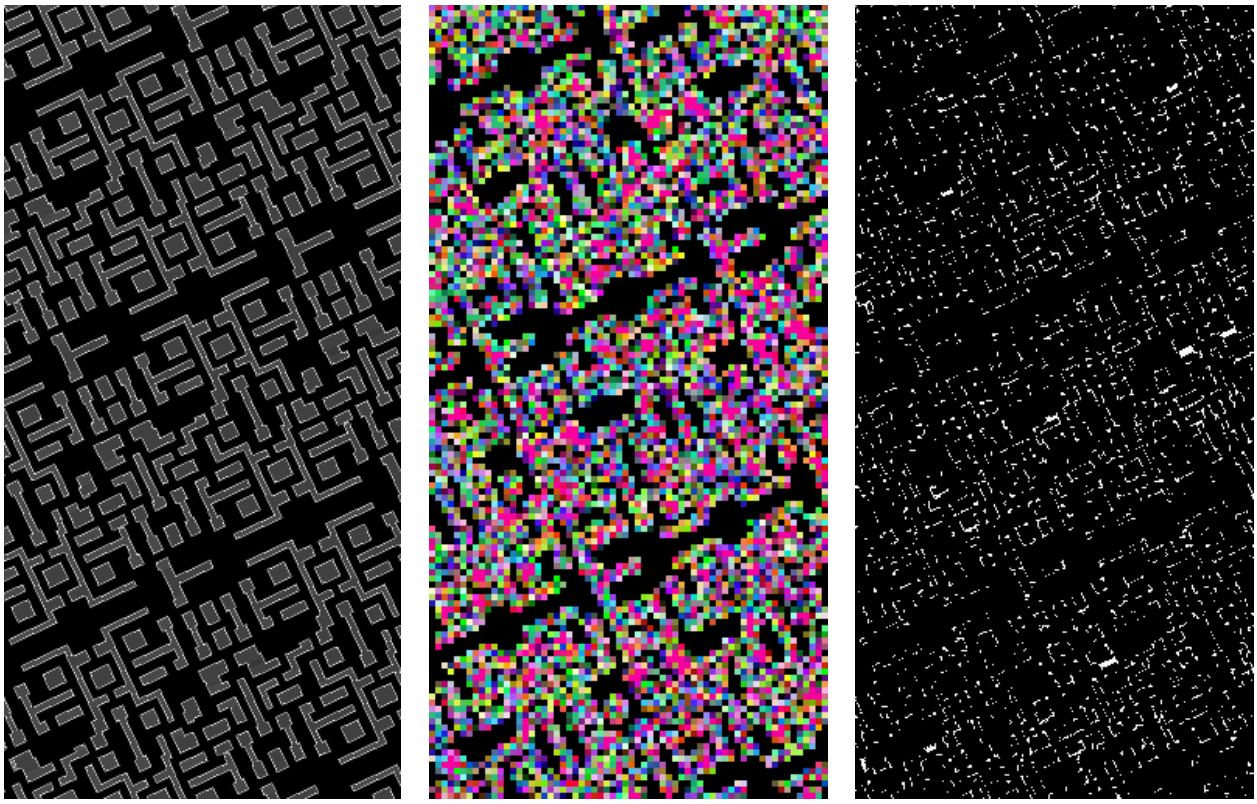
Consider our case of compressing rasterized layout images. Divide an image grid into a set of non-overlapping 4×4-pixel blocks. A 4×4-pixel codeword is assigned to each block which should closely approximate the actual image block; for now, assume the two are equal. The set of possible codewords, known as the codebook, is assumed to be known by the decoder. Each codeword represents a 16-dimensional vector; each dimension can take one of 32 pixel values, and is represented by 5 bits in the simple case of scalar quantization. Thus, a pointer to each 4×4 codeword should be represented by, at most, 16×5 = 80 bits, assuming that all 2^{80} possible codeword values are possible. However, the layout image in Fig. 15(a) is highly structured: feature boundaries are locally restricted to a particular angle of orientation, and are surrounded by background pixels on one side and foreground pixels on the other. Assuming that boundary pixels are inherently random, and that each codeword can contain a maximum of 6 boundary pixels, the number of bits per codeword index is reduced from 80 bits to 6×5 + 1 = 31 bits, where the extra bit allocates "foreground" or "background" to either side of each edge in the given codeword. In reality, boundary pixel values and locations are not random; this further reduces the size of the required codebook.

One way of implementing VQ is to construct a single codebook for an entire wafer layer. The advantage of this method is that a decoder chip repeatedly writing the same wafer layer uses a static codebook. Thus, the codebook can be stored internally without impacting the compression ratio. In this case, the compression ratio is $\frac{\#\text{ bits per pixel} \times \#\text{ pixels per codeword}}{\lceil \log_2(\#\text{ codewords}) \rceil}$, which equals 80/31 in the previous

example. Unfortunately, the memory requirement of this implementation is huge: empirically, the codebook corresponding to a 1024×256 pixel section of the 25° Metal 1 layer requires 51 KB, already beyond the 40KB limit of each decoder in REBL's DPG chip. This observation leads to two conclusions. First, the codebook must be repeatedly refreshed as the wafer is being written. This has the added benefit of ensuring that, at any given moment, each codeword is well-adapted to the current angle of orientation and the local properties of the image pattern. Second, the codebook likely should not be so large that it can reproduce the original image without errors, as this also would require too many memory buffers in the decoder.

The VQ compression algorithm pursued below is structured as follows: (1) send a codebook of a limited size, which is well-adapted to the image currently being compressed, (2) transmit a codebook index for each 4×4 image block, which determines which codeword to copy from, and (3) correct image errors wherever they occur, by transmitting an error map and a list of error values. The image error map transmits "1" if the associated pixel contains an error, else "0"; it is compressed using Golomb run-length codes. A 5-bit error value is transmitted for each pixel associated with a "1" in the error map. The codebook, codebook indices, and image error values are all compressed using Huffman codes.

The VQ algorithm is visualized in Fig. 15. Fig. 15(a) is the layout image being compressed. Fig. 15(b) assigns each 4×4 image block a codeword index, where each index is randomly assigned a different color. The black codebook index corresponds to a codeword composed entirely of background pixels, while the bright pink codebook index represents a codeword composed entirely of gray foreground pixels. Before Huffman coding, each uncompressed codebook index requires $\log_2[\textit{codebook size}]$ bits. Each white pixel in Fig. 15(c) indicates a discrepancy between the original image pixel and the pixel value determined by the block's codebook index. 5.1% of the pixels in this image contain errors, which are corrected using the image error map and the corresponding list of error values.



(a) (b) (c)

Fig. 15. (a) Original 256×512 layout image segment. (b) Codebook indices (c) Image error map

6.2 Comparison with Block RGC3

Block RGC3 and the proposed VQ approach are quite similar. Both algorithms approximate each non-overlapping $H \times W$ image block by copying the best match from a list of blocks known by the decoder. Pixel errors are corrected using an image error map and the associated image error values. The only difference between the two algorithms is the origin of the list of possible copy blocks. Block RGC3 uses a list of copy distances, in tandem with a history buffer of recently-decoded pixels. The general VQ approach, however, is free to construct a list of codewords however it wants, with the intent of minimizing image errors for the layout section it is applied to. The main advantage of VQ is that a smaller list of copy blocks is required to provide a given maximum image error rate. Having a smaller codebook reduces the number of bits per codebook index, which may improve compression efficiency; it also reduces the encoding time required to find which codebook indices will minimize image errors.

However, disadvantages include: (1) the codebook will likely need to be transmitted as part of the compressed image bitstream, thus degrading compression efficiency, (2) constructing an optimal codebook may require extra encoding time, and (3) since each codeword is unique, nearly every image block has only one optimal codebook index, so growing 1-D codeword regions is ineffective, unlike with Block RGC3 as in Section 4.3.

6.3 Determining the codebook

The single advantage VQ has over Block RGC3 is its ability to construct a smaller, customized list of copy blocks (i.e., codewords, or copy distances in the case of Block RGC3) which generates fewer image errors. Thus, optimizing this codebook is the biggest opportunity to improve performance.

Fig. 16(a) shows a sorted cumulative distribution function (cdf) of the codewords composing a 1024×256, 25°-oriented Metal 1 image, assuming zero image errors are allowed. There are 5144 codewords in total, the most common of which is composed entirely of background pixels and makes up 32.7% of the image. 50% of image blocks can be encoded error-free using only 116 codewords; however this number extends to 1222 codewords for 75% of image blocks, or 3505 codewords for 90% of image blocks. Given n_c codewords, the decoder storage requirement for the codebook is n_c codewords × 16 pixels per codeword × 5/8 bytes per pixel = n_c × 10 bytes. Thus, a codebook large enough to eliminate all errors requires 51 KB, which, as previously mentioned, is beyond the storage requirements of the REBL system. Additionally, transmitting such a large codebook for each 1024×256 image section requires 31 KB even after applying Huffman codes. Even if this 31 KB codebook could magically decode the image by itself, compression efficiency would still be limited to 5.3. Clearly, the codebook size must be limited in some way.

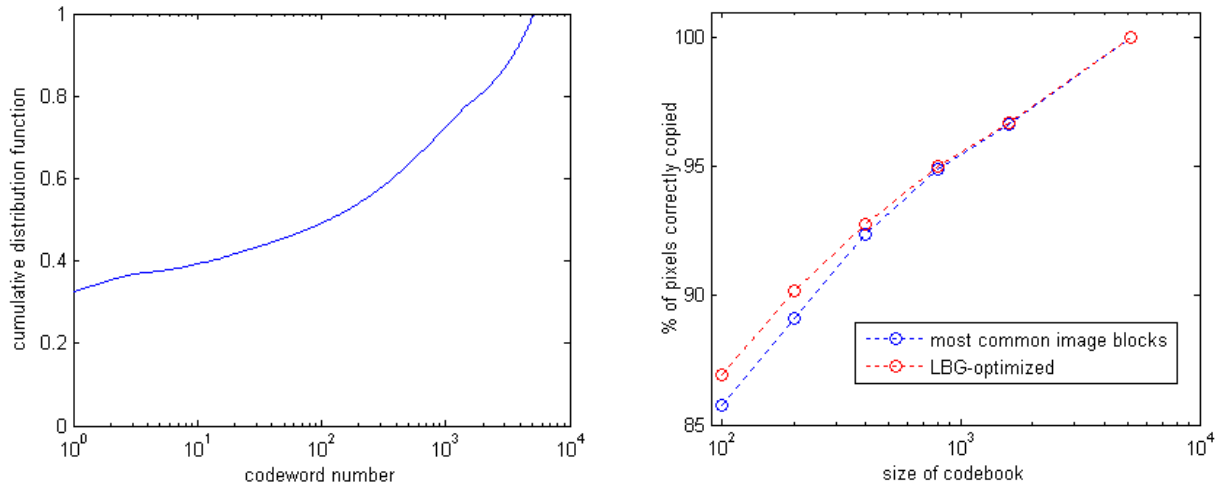


Fig. 16. (a) Cumulative distribution function (cdf) of the codewords composing a 1024×256, 25°-oriented Metal 1 image, assuming zero image errors. (b) The percentage of correctly-copied pixels, as a function of the codebook size.

One way to limit the codebook size is to use only the n_c most common image blocks. Each image block is then approximated using the codeword that minimizes the number of image errors, just as is performed by Block RGC3. The effectiveness of this heuristic is shown by the blue curve in Fig. 16(b), which graphs the percentage of correctly-copied image pixels as a function of the codebook size. Assuming the codebook, codebook indices, and image error values are transmitted uncompressed, the optimal codebook size minimizes the expression: $\gamma_1 \times$ (# codewords) + $\gamma_2 \times \log_2$(# codewords) + $\gamma_3 \times$ (# image errors), where γ_1 = # bits per codeword = 16 pixels × 5 bits per pixel = 80, γ_2 = # blocks in image = 1024×256 pixels / 16 pixels per block, and γ_3 = # bits per image error = 5. Interpolating the blue curve from Fig. 16(b) and minimizing the above expression yields an optimal codebook size of 400. In practice, Huffman coding compresses the image errors less than than the codebook or the codebook indices; thus, a codebook size of 800 has been empirically found to maximize compression efficiency for 1024×256 Metal 1 images. Sparse wafer layers and smaller image sizes tend to have a smaller optimal codebook size.

A codebook composed of the n_c most common image blocks is not guaranteed to minimize image errors. A common vector quantization technique for optimizing the codebook is the Linde-Buzo-Gray (LBG)

algorithm [15] [17]. The algorithm works as follows: (1) start with an initial codebook, (2) associate a codeword with each image block, such that the "distance" between the codeword and the image block is minimized, (3) compute the total "distortion" in the image, which is the sum of the distances between each image block and its associated codeword, (4) if the distortion has decreased by a certain percentage since the last iteration, then continue iterating; otherwise stop, (5) reassign each codeword to equal the "centroid" of all image blocks currently associated with that codeword, then go to Step (2). The terms "distance" and "centroid" depend on the metric being minimized, which in this case is the number of image errors. Therefore, distance should equal the number of image pixel errors incurred by comparing a given image block with a given codeword. The centroid of each codeword pixel is the pixel value which minimizes the number of errors incurred by all of its associated image pixels.

The LBG algorithm requires an initial codebook, as mentioned in Step (1). This can be determined in a variety of ways. The "splitting" technique starts by creating a single codeword located at the centroid of all image blocks; it then splits each existing codeword in two and runs the LBG algorithm; this step is iterated until the codebook has the desired size. For our application, we have found that the splitting technique requires a prohibitively large encoding time and suffers greatly from the "empty cell" problem described in [15]. The "pairwise nearest neighbor" technique initializes the codebook to the entire set of image blocks, then iteratively combines the two closest codewords until the codebook has the desired size. This technique also requires prohibitively large encoding times. A third method is to start the LBG optimization using the set of n_c most common image blocks. Results using this method are shown in Fig. 16(b). As shown, the LBG algorithm is guaranteed to improve performance. Unfortunately, it also increases the encoding time from 6.6 seconds to 150 seconds for the given image. Additionally, improvement is marginal for the known "optimal" codebook size of 800, since LBG loses its effectiveness as the codebook size approaches the total number of unique image blocks. For these reasons, our final VQ algorithm excludes the LBG algorithm, and the codebook is simply composed of the n_c most common image blocks.

7. VECTOR QUANTIZATION RESULTS

7.1 Compression Results

Table 10 shows the average VQ compression efficiencies for several layout samples of different image sizes and codebook sizes. A new codebook is assumed to be transmitted for each image segment. Different wafer layers have significantly different compression ratios, ranging from 12.3 for the via layer, to 7.6 for the poly layer, to 4.9 for Metal 1. The impact of codebook size, image size, and angle of rotation on compression efficiency is not nearly as large.

Table 10. Average VQ compression efficiency for various layouts, image sizes, and codebook sizes.

Image size	Codebook size	Metal 1			Metal 1b[1]	Poly	Via	
		25°	35°	38°	25°	35°	25°	35°
64×2048	250	3.77	4.39	4.21	--	6.08	11.75	11.53
256×1024	250	3.88	4.20	4.36	6.12	6.26	12.28	11.77
256×2048	250	3.82	4.24	4.39	--	6.37	11.86	11.52
64×2048	800	3.92	4.57	4.20	--	5.86	10.09	9.84
256×1024	800	4.46	4.67	4.86	6.28	6.88	11.71	11.50
256×2048	800	4.76	4.82	5.10	--	7.27	12.71	12.51
64×2048	1200	3.62	4.41	3.83	--	5.28	--	--
256×1024	1200	4.39	4.54	4.72	5.71	6.61	10.88	10.69
256×2048	1200	4.91	4.89	5.11	--	7.56	12.35	12.22

For a given codebook size, larger image sizes consistently yield higher compression efficiency, because each transmitted codebook is applied to a larger image area. The single exception is enlarging the image size from 256×1024 to 256×2048, assuming 250 codewords. In this case, the codebook size is simply too small for such a large image area, and the high number of image errors outweighs the benefit of transmitting fewer codewords for a given image area. Ignoring this one exception, the general conclusion is that each codebook should be applied to images having a width of 4096 pixels or beyond. Unfortunately, the test images available for this research were not large enough to verify this hypothesis. Nevertheless, an upper-bound on this improvement is reached when the codebook's impact on compression efficiency becomes negligible. For 25° Metal 1 images with a 256-pixel height, compression

efficiency is bounded by 5.5 assuming a codebook size of 1200, or by 5.1 assuming a codebook size of 800.

Unlike with Block RGC3, the codebook must be transmitted for the VQ implementation. Thus, larger codebooks can degrade compression efficiency, offsetting the benefits of fewer image errors and fewer bits per codebook index. The optimal codebook size depends on the number of unique patterns in a given image; thus, smaller image sizes and sparser wafer layers tend to yield smaller optimal codebook sizes.

7.2 Encoding Complexity Results

The VQ encoding process is dominated by two steps:

(1) Generating the codebook requires compiling a histogram of the set of all unique image blocks, then sorting this list and saving the most common members. Encoding time is independent of the codebook size but is layout-dependent, since each consecutive image block must be searched for in the current histogram before moving on. In the best case, each image block in a layout containing only "background" pixels is immediately matched to the only member of the histogram; here, the encoding time per image area for generating the histogram is independent of the image size. In the worst case, encoding time per image area is linearly proportional to the image size; this occurs if each image block is unique and must be found in a histogram whose size is proportional to the image size. In short, this contribution to encoding time is proportional to $O(\textit{image size})$ for a given image area.

(2) Each image block is compared with each codeword, in order to determine the set of codeword indices which minimizes image errors; the encoding time of this step is layout-independent, is dominant for high codebook sizes, and is linearly proportional to the codebook size. In other words, encoding time is proportional to $O(\textit{codebook size})$ for a given image area.

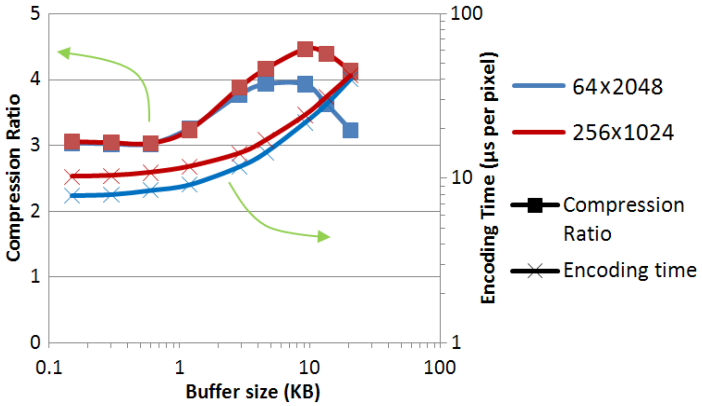
Table 11 shows encoding times for Metal 1 and Via layers, for various image sizes and codebook sizes. For high codebook sizes, encoding time is linearly proportional to image size; also, the linear *increase* in

encoding time as a function of the codebook size is independent of the wafer layer. However, assuming codebook size of 250, there is a clear discrepancy between the Metal 1 and Via encoding times, which results from the layout dependency of Contribution (1), discussed above. Because the Via layer is composed mainly of "background" pixels, its histogram of unique image blocks is generated much faster.

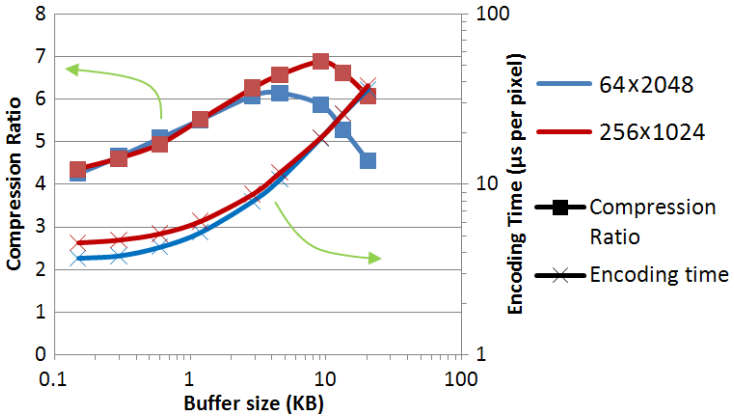
Table 11. Encoding and Decoding Times, using Vector Quantization

		Encoding Time (sec)	
Image size	Codebook size	Metal 1 25°	Via 25°
64×2048	250	1.55	0.83
256×1024	250	3.73	1.72
256×2048	250	9.66	3.75
64×2048	800	2.88	2.16
256×1024	800	6.38	4.41
256×2048	800	14.75	9.13
64×2048	1200	3.73	--
256×1024	1200	8.27	6.38
256×2048	1200	18.38	13.08

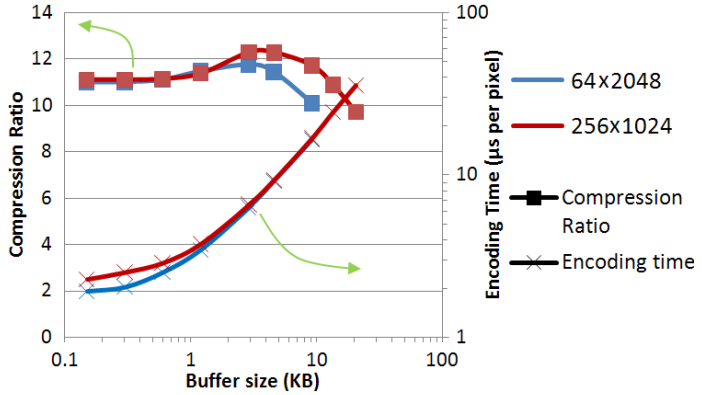
The effect of buffer size on VQ compression efficiency and encoding complexity is visualized in Fig. 17. The uncompressed codebook makes up about 85% of the required buffer. The remainder comprises Huffman codetables for the codebook, the codebook indices, and the error values. The plotted points for each image section correspond to a codebook size of 12, 25, 50, 100, 250, 400, 800, 1200, or 1800. Unlike with Block RGC3, larger buffer sizes do not necessarily improve the compression efficiency of a given test image. This results from VQ's need to transmit the codebook for each image section. As expected, smaller image sizes and sparser wafer layers yield smaller optimal codebook sizes; specifically, Metal 1 is less sparse than Poly, which is less sparse than Via.



(a)



(b)



(c)

Fig. 17. Avg. VQ compression ratios, encoding times vs. buffer size for (a) 25° Metal 1, (b) 35° Poly, (c) 25° Via

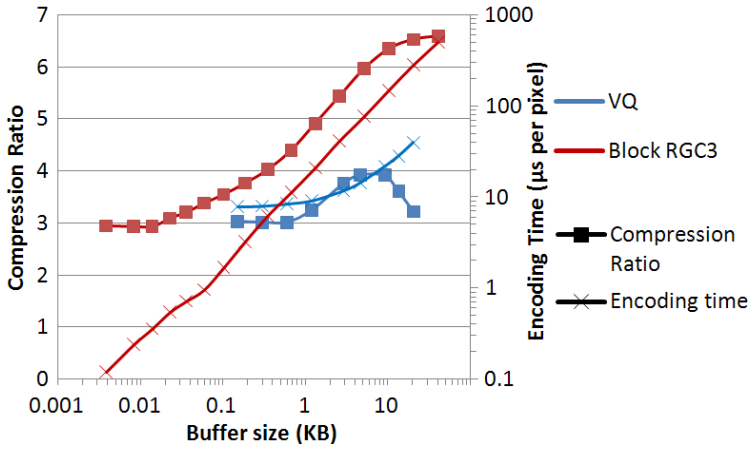
Assuming a large buffer size, encoding time increases linearly as a function of buffer size. This behavior occurs as the codebook grows larger and Contribution (2) becomes dominant. Notice that this linear

behavior is particularly noticeable for the Via layer, since Contribution (1) is relatively small. At low buffer sizes, the Metal 1 and Poly encoding times lose their dependency on codebook size, as Contribution (1) becomes dominant. In this region, the encoding time per pixel is higher for larger image sizes, as expected.

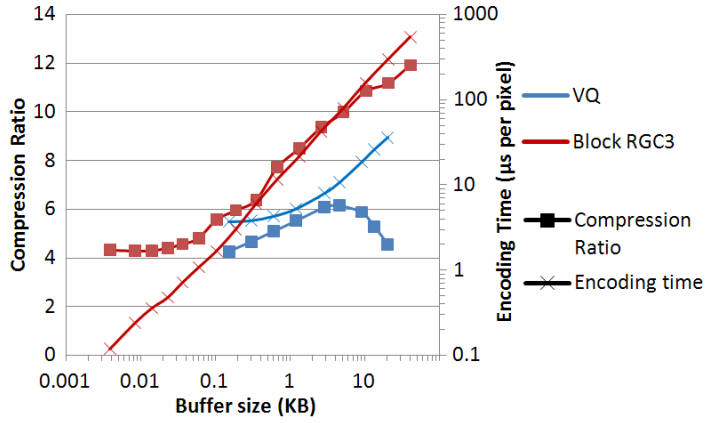
8. PERFORMANCE COMPARISON: VQ VS. BLOCK RGC3

The compression efficiency and software encoding complexity of both VQ and Block RGC3 are compared in Fig. 18, assuming a 64×2048 image size. In all cases, VQ has a lower encoding time than Block RGC3 for a given buffer size; since Block RGC3 copy distances allow copy blocks to overlap, but VQ codebook indices do not, Block RGC3 yields a higher number of possible copy blocks for a given buffer size, which increases encoding complexity. Nevertheless, using a very small Block RGC3 buffer still yields higher compression efficiency than the best VQ result. As shown in Fig. 18(c), the Via layer yields the best performance for VQ, relative to Block RGC3. Still, for any desired compression ratio, Block RGC3 requires a smaller buffer and less encoding time than VQ.

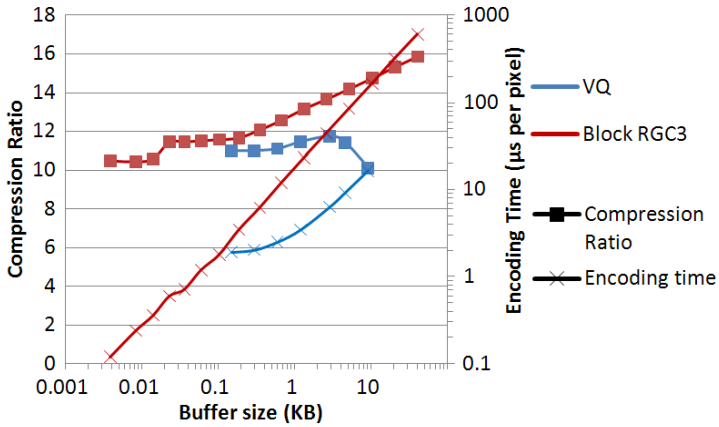
Fig. 18 makes it clear that Block RGC3 outperforms VQ in terms of compression efficiency, encoding time, and required buffers. This is true whether the goal is solely to maximize compression efficiency at any cost, or to sacrifice some compression performance in favor of either reduced buffering or reduced encoding times. Additionally, the hardware design of the Block RGC3 decoder is inherently easier to implement. Removing Huffman codes reduces Block RGC3 compression efficiency by only 1-2%, but reduces VQ compression efficiency by around 25%. Moreover, the Block RGC3 history buffer is updated automatically, whereas VQ codebooks must be periodically loaded into memory, possibly interrupting the decode process. Thus, from every standpoint, Block RGC3 clearly appears to be the better choice.



(a)



(b)



(c)

Fig. 18. VQ and Block RGC3: average compression ratios and software encoding times vs. buffer size for (a) 25° Metal 1, (b) 35° Poly, (c) 25° Via. A 64×2048 image size is assumed.

9. SUMMARY, CONCLUSIONS, AND FUTURE WORK

This thesis has described Block RGC3, a lossless compression algorithm which optimizes compression efficiency for layouts compatible with the REBL Direct-Write E-Beam Lithography System. Layouts are assumed to be rotated at an arbitrary angle with respect to the pixel grid, and have undergone E-beam proximity correction compatible with the REBL system. Three main modifications to the original Block GC3 algorithm have been described. First, diagonal copying allows repetition of image data to be exploited, regardless of the layout's rotation. Second, a region-growing method assigns equal segmentation values to neighboring blocks, reducing the impact of segmentation information on compression efficiency. Third, pixel-based prediction is removed, thus simplifying the decoder's hardware implementation without degrading performance. On average, Block RGC3 has compressed Metal 1 images by a factor of 7.3, assuming a worst-case rotation of 25°. This compares with a compression ratio of 3.3, if the original Block GC3 algorithm is used. The drawback of Block RGC3 is increased encoding time, which is roughly proportional to the allowed copy range.

An alternate compression algorithm utilizing vector quantization has also been explored. A new codebook is transmitted for every rectangular image section of a chosen size. Each codebook consists of its image section's most common 4×4-pixel blocks, and a codeword is matched to each image block such that the number of image errors is minimized. Performance tradeoffs involving the sizes of each image section and its codebook have been explored. If the optimal buffer size is chosen, Block RGC3 has been found to outperform vector quantization for all desirable corners of operation.

For future research, it would be worthwhile to test our algorithm on a larger, more diverse set of layout images, in order to better characterize both compression efficiency and encoding complexity. Also, we might explore the idea of matching vector quantization codewords to image blocks which do not conform to a strict 4×4-pixel grid; this extra flexibility may reduce the number of image errors for a given codebook size. Finally, Section 5.2 notes that portions of Block RGC3's encoding complexity are image-

dependent. In order to bound the maximum encoding time, it may be necessary to set a hard limit on the size of $d_{matches,block}$.

In a practical implementation, controlling the position of the writer chip relative to the wafer to within 1 nm accuracy at all times may not be feasible. If so, this uncertainty dictates that layout data must be compressed in real-time through hardware, instead of being compressed in advance through software. Given the significant encoding complexity of both Block RGC3 and the described vector quantization technique, the process of exhaustively searching through a list of possible copy blocks for every image block likely must be restricted.

APPENDIX A: NP-COMPLETENESS PROOF OF 2-D REGION-GROWING

Assume that each Block RGC3 image block is given a list of copy distances which yield no more than some pre-determined number of image errors, as described in Section 4.3. In this Appendix, we show that the process of choosing one copy distance from each block's list such that the total number of regions is minimized is NP-complete. We define a "region" as a group of 4-connected blocks each having the same copy distance. Let X_1 be an image with n optimal copy distances $D(p) = \{d_{1(p)}, d_{2(p)}, \ldots d_{n(p)}\}$ for all blocks $p \in X_1$. Our goal is to minimize the number of regions in X_1, such that each block in a given region has at least one copy distance in common. More formally, the following uniformity predicate holds:

$$U_1(X_1) = \text{ true iff } \forall p \in X_{1,i}, \exists\, a \text{ s.t. } a \in D(p),$$

where $X_{1,i}$ is any region in X_1. As proven by MIN2DSEG in [12], minimizing the number of 2-D regions in an image X_2 is an NP-complete problem, assuming the uniformity predicate

$$U_2(X_2) = \text{ true iff } \forall\, p,\, q \in X_{2,i}, \left|I(p) - I(q)\right| \leq 1,$$

where $I(p)$ are the pixel values for all pixels $p \in X_2$. For X_2, let $D(p) = \{I(p), I(p) + 1\}$. $U_1(X_2)$ and $U_2(X_2)$ are equivalent; this proves the reduction from MIN2DSEG in [12] to our 2-D region-growing method, which is thus NP-complete.

APPENDIX B: FULL CHIP CHARACTERIZATION OF AN ASIC, USING BLOCK C4

Full chip performance characterization of Block C4 has been performed in [13], with both an industry microprocessor and a low-volume ASIC being tested. Note that the original Block C4 is being tested, not Block RGC3. The test assumes a zero degree rotation for all images. Layout rasterization is performed as in Section 2.2 of [3], instead of using REBL E-Beam correction techniques.

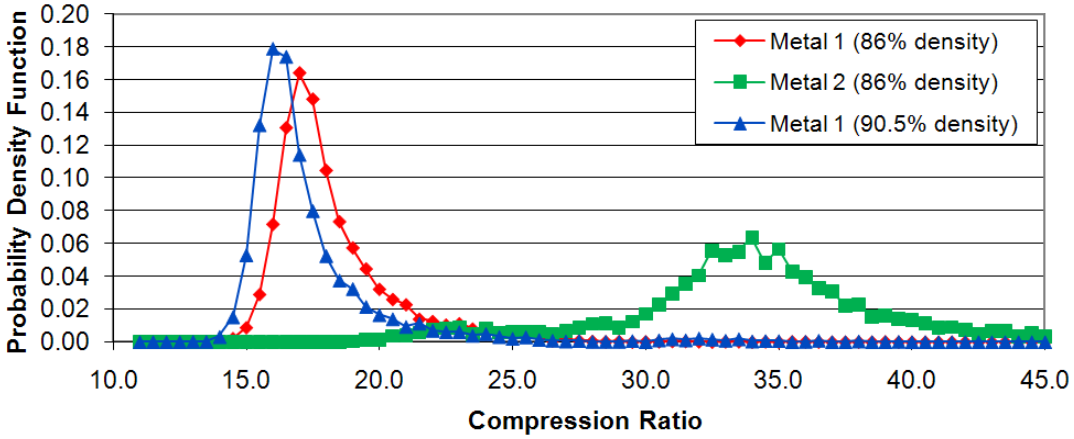


Fig. 19. Probability density functions of compression ratios for Block C4 for an LDPC ASIC.

The compression performance of the ASIC is shown in Fig. 19. The chip is a Low Density Parity Check (LDPC) decoder chip in the 65 nm technology, with layout placement and routing generated using Synopsys Astro. Assuming a pixel size of 32 nm, each block is 1024 × 1024 pixels, or 32 $\mu m \times 32 \mu m$. Figure 9 shows the histogram of compression ratios for Metal 1 and Metal 2 layers. For the Metal 1 layer, we have applied the routing tool twice in order to generate two different layout densities, namely 86 % and 90.5 %. As expected, the compression ratio drops as the density goes up. In addition, Metal 1 blocks are harder to compress than those of Metal 2. Metal 1 contains optimally dense wires inherent to each standard cell and between neighboring cells, while Metal 2 wires are used to connect nearby cells. Thus, despite the presence of easily-compressible V_{dd} and ground rails on the Metal 1 layer, Metal 1 is consistently more difficult to compress than Metal 2, which often contains large blank spaces in areas where inter-cell routing is straightforward. The minimum compression ratios for Metal 1 (86%), Metal 1 (90%) and Metal 2 (86%) are 14.3, 13.2 and 18.7 respectively.

ACKNOWLEDGMENTS

This research was conducted under the Research Network for Advanced Lithography, supported by MICRO, KLA Tencor, and California State Micro award #07-027.

I would like to thank my research advisor, Professor Avideh Zakhor, for her encouragement, guidance, and insight, which have helped focus my research aspirations and subsequently ensure my steady progress. Without her vision, my efforts would have been far less productive. Thanks also to Hsin-I Liu, who helped me to understand the Block GC3 algorithm and made many useful suggestions to guide my research.

Also, I would like to thank Allen Carroll and Andrea Trave at KLA-Tencor for providing access to the REBL system, for providing EPC layouts, and for accommodating the needs of this project. Zhengya Zhang and Brian Richards at the Berkeley Wireless Research Center were instrumental in providing ASIC chips and software tools necessary to test the Block C4 algorithm. Finally, without the original groundwork for Block C4 laid by Vito Dai, this work could never have been accomplished.

REFERENCES

[1] V. Dai and A. Zakhor, "Lossless Compression Techniques for Maskless Lithography Data", Emerging Lithographic Technologies VI, Proc. of the SPIE Vol. 4688, pp. 583–594, 2002.

[2] V. Dai and A. Zakhor, "Advanced Low-complexity Compression for Maskless Lithography Data", Emerging Lithographic Technologies VIII, Proc. of the SPIE Vol. 5374, pp. 610–618, 2004.

[3] V. Dai. "Data Compression for Maskless Lithography Systems: Architecture, Algorithms and Implementation," Ph.D. Dissertation, UC Berkeley, 2008.

[4] H. Liu, V. Dai, A. Zakhor, B. Nikolic, "Reduced Complexity Compression Algorithms for Direct-Write Maskless Lithography Systems," SPIE Journal of Microlithography, MEMS, and MOEMS (JM3), Vol. 6, 013007, Feb. 2, 2007.

[5] P. Petric *et.* al, "REBL Nanowriter: A Novel Approach to High Speed Maskless Electron Beam Direct Write Lithography," International conference on Electron, Ion, and Photon Beam Technology and Nanofabrication (EIPBN), 2008.

[6] P. Petric *et.* al, " Reflective electron-beam lithography (REBL)," Alternative Lithographic Technologies, Proc. of SPIE Vol. 7271, 2009.

[7] J. Ziv, and A. Lempel, "A universal algorithm for sequential data compression," IEEE Trans. on Information Theory, IT-23 (3), pp. 337–43, 1977.

[8] M. Burrows and D. J. Wheeler, "A block-sorting lossless data compression algorithm," Technical report 124, Digital Equipment Corporation, Palo Alto CA, 1994.

[9] D. A. Huffman, "A Method for the Construction of Minimum-Redundancy Codes," Proceedings of the IRE, 40(9), pp. 1098-1101, September 1952.

[10] M. J. Weinberger, G. Seroussi, and G. Sapiro, "The LOCO-I lossless image compression algorithm: principles and standardization into JPEG-LS," IEEE Transactions on Image Processing, 9 (8), pp. 1309–1324, 2000.

[11] M. E. Papadonikolakis, A. P. Kakarountas, and C. E. Coutis," Efficient high-performance implementation of JPEG-LS encoder," Journal of Real-Time Image Processing, 3, pp. 303-310, 2008.

[12] M. C. Cooper. “The Tractability of Segmentation and Scene Analysis,” International Journal of Computer Vision 30(1), pp. 27-42, 1998.

[13] A. Zakhor, V. Dai, and G. Cramer. “Full-chip Characterization of Compression Algorithms for Direct-Write Maskless Lithography Systems,” Alternative Lithographic Technologies, Proc. of SPIE Vol. 7271, 2009.

[14] *pending* G. Cramer, H. Liu, A. Zakhor. “Lossless Compression Algorithm for REBL Direct-Write E-Beam Lithography System,” Alternative Lithographic Technologies, Proc. of SPIE, 2010.

[15] K. Sayood. “Introduction to Data Compression,” 3rd Ed. Morgan Kaufmann, 2006.

[16] A. Gersho and and R. M. Gray. “Vector Quantization and Signal Compression,” Springer, 1991.

[17] Y. Linde, A. Buzo, and R. M. Gray. “An algorithm for vector quantization design,” IEEE Transactions on Communications, COM-28:84-95, Jan. 1980.