Global Stitching of Integrated Circuit Images

Colin Macdonald* and Benjamin Ong†

September 3, 2002

1 Introduction

The basic idea behind our approach to globally stitching integrated circuit (IC) images is to associate a cost function for any given alignment of images, and minimize the resulting cost function to find the optimal alignment of images.

We represent each image as a matrix $[u^{ij}]_{L_y \times L_x}$ of integers (corresponding to grey scale pixel values), where L_y and L_x denotes the size of each image. Note that i and j do not correspond to the elements of u^{ij}, but rather, they denote the column and row positions of the image respectively (see Figure 1).

There are target offset overlaps (S_x and S_y) which provide overlap regions to help align the images. Assuming that the target overlaps are achieved, the top left corner of each image define the reference grid (again, see Figure 1). In practice however, errors during image acquisition modify these overlap regions. This can be quantified by associated an offset error $\vec{\Delta}^{pq} = (\Delta_x^{pq}, \Delta_y^{pq})$ with every image u^{pq}. The offset error is defined relative to the reference grid. The left overlap region between image u^{ij} and $u^{i,j-1}$ has width $\big(S_x - (\Delta_x^{ij} - \Delta_x^{i,j-1})\big)$ and height $\big(L_y - (\Delta_y^{ij} - \Delta_y^{i,j-1})\big)$. Similarly, the top overlap region between image u^{ij} and $u^{i-1,j}$ has width $\big(L_x - (\Delta_x^{ij} - \Delta_x^{i-1,j})\big)$ and height $\big(S_y - (\Delta_y^{ij} - \Delta_y^{i-1,j})\big)$.

It is also assumed that we have an upper bound on the offset errors, i.e., $|\Delta_x^{ij}| \leq \beta_x$ and $|\Delta_y^{ij}| \leq \beta_y$ where β_x and β_y are constants, or, an upper bound

*Simon Fraser University, cbm@math.sfu.ca
†Simon Fraser University, bwo@math.sfu.ca

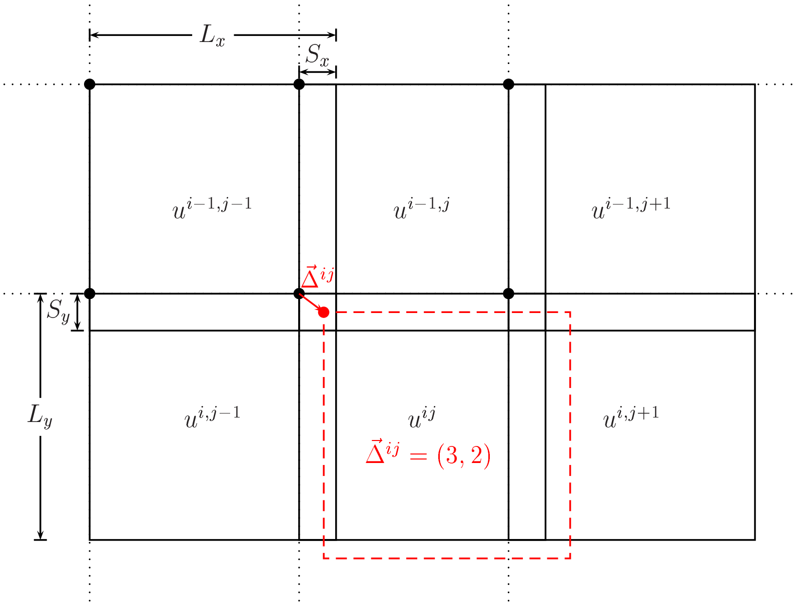

Figure 1: Image layout using reference grid approach. Solid lines show images at target offset overlaps (upper left corners are shown as •), dashed box shows image u^{ij} displaced by offset error $\vec{\Delta}^{ij}$.

on the relative offset errors between every pair of neighbouring images. Let

$$\eta_x = \Delta_x^{i,j-1} - \Delta_x^{ij}, \tag{1a}$$
$$\eta_y = \Delta_y^{i,j-1} - \Delta_y^{ij}, \tag{1b}$$
$$\xi_x = \Delta_x^{i-1,j} - \Delta_x^{ij}, \tag{1c}$$
$$\xi_y = \Delta_y^{i-1,j} - \Delta_y^{ij}, \tag{1d}$$

and thus

$$|\eta_x| \leq \beta_x, \quad |\eta_y| \leq \beta_y,$$
$$|\xi_x| \leq \beta_x, \quad |\xi_y| \leq \beta_y.$$

It is unclear from the problem specification whether the β_x and β_y bound the relative offset errors or the actual offset errors, but the following discussion and code handles both cases.

2 Cost Function

We calculate the cost function by summing the contributions from the top and left overlap regions for all images u^{ij}. The cost function contribution due to u^{ij} is

$$
\begin{aligned}
d^{ij} &= \frac{\sum\sum_{\text{left overlap pixels}} \left|u^{ij} - u^{i,j-1}\right|^2}{\text{\# of left overlap pixels}} + \frac{\sum\sum_{\text{top overlap pixels}} \left|u^{ij} - u^{i-1,j}\right|^2}{\text{\# of top overlap pixels}}, \\
&= \text{cost\_contrib\_left}(u^{ij},\, u^{i,j-1},\, \eta_x, \eta_y) \\
&\qquad\qquad + \text{cost\_contrib\_top}(u^{ij},\, u^{i-1,j},\, \xi_x, \xi_y),
\end{aligned} \tag{2}
$$

where the first term is 0 if $j = 1$ and the second term is 0 if $i = 1$. Note that d^{ij} depends explicitly on $\vec{\Delta}^{ij}$, $\vec{\Delta}^{i-1,j}$ and $\vec{\Delta}^{i,j-1}$. The total cost function is then calculated as

$$d = \sum_i \sum_j d^{ij}. \tag{3}$$

3 Minimization

At this point, we try minimizing the above cost function using Matlab's *fminsearch* over the space of $\vec{\Delta}$'s. However, we encounter an unexpected hurdle: the stitching of images is not a discrete problem whereas our cost function calculation assumes that the offset errors were integers. N.B., one may question the usefulness of resolving the offset errors to a "sub-pixel" level when feature sizes are $\mathcal{O}(10)$ pixels, however, finding a free black-box optimization code which generates and *uses* only integer values is difficult. We propose several fixes:

- round $\vec{\Delta}^{ij}$ when calculating the cost function.
- interpolate the image data on the sub-pixel level.
- interpolate our cost function.

As expected, rounding does not produce a convergent result. Interpolating the image is impractical for larger problems where realistically, a caching approach as discussed in Section 3.5 is required. We choose to implement interpolation of the cost function.

3.1 Cost Function Interpolation

Let cost_contrib_left() be a function that calculates the cost function contribution in the left overlap region between neighbouring images u^{ij} and $u^{i,j-1}$. Assuming that η_x and η_y are integers, the cost contribution is calculated as

$$\text{cost\_contrib\_left}(u^{ij},\ u^{i,j-1},\ \eta_x, \eta_y).$$

To accommodate fractional offset errors, consider

$$\eta_x = \lfloor \eta_x \rfloor + \delta_x, \tag{4a}$$
$$\eta_y = \lfloor \eta_y \rfloor + \delta_y, \tag{4b}$$

where $\vec{\delta} \in [0,1) \times [0,1)$, $\lfloor \cdot \rfloor = \text{floor}(\cdot)$ and $\lceil \cdot \rceil = \text{ceil}(\cdot)$.

We now build the lookup table for the cost function contribution at the 4 nearest neighbouring integer values of $\vec{\eta}$ by calculating

$$c_{\lfloor x \rfloor \lceil y \rceil} = \text{cost\_contrib\_left}(u^{ij},\ u^{i,j-1},\ \lfloor \eta_x \rfloor, \lceil \eta_y \rceil), \tag{5a}$$
$$c_{\lceil x \rceil \lceil y \rceil} = \text{cost\_contrib\_left}(u^{ij},\ u^{i,j-1},\ \lceil \eta_x \rceil, \lceil \eta_y \rceil), \tag{5b}$$
$$c_{\lfloor x \rfloor \lfloor y \rfloor} = \text{cost\_contrib\_left}(u^{ij},\ u^{i,j-1},\ \lfloor \eta_x \rfloor, \lfloor \eta_y \rfloor), \tag{5c}$$
$$c_{\lceil x \rceil \lfloor y \rfloor} = \text{cost\_contrib\_left}(u^{ij},\ u^{i,j-1},\ \lceil \eta_x \rceil, \lfloor \eta_y \rfloor). \tag{5d}$$

Our estimate, c of the cost function contribution is a weighted average of (5). Specifically, we use bilinear interpolation, by first calculating

$$c_{\lceil x \rceil} = (1 - \delta_y) c_{\lceil x \rceil \lfloor y \rfloor} + \delta_y c_{\lceil x \rceil \lceil y \rceil},$$
$$c_{\lfloor x \rfloor} = (1 - \delta_y) c_{\lfloor x \rfloor \lfloor y \rfloor} + \delta_y c_{\lfloor x \rfloor \lceil y \rceil},$$

and then calculating

$$c = (1 - \delta_x) c_{\lfloor x \rfloor} + \delta_x c_{\lceil x \rceil}.$$

A similar discussion applies for cost_contrib_up() using xi_x and xi_y.

3.2 Cost Function Space

The following set of images was used as Test Case 1. The original image of a chip (provided by Semiconductor Insights, Figure 2) was artificially broken into nine 700×600 pieces with overlap regions of 100 pixels and random offset

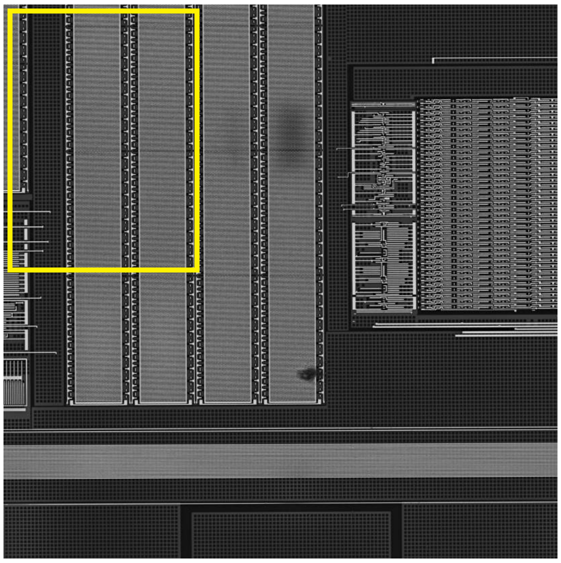

Figure 2: Ymag_shift.jpg provided by Semiconductor Insights.

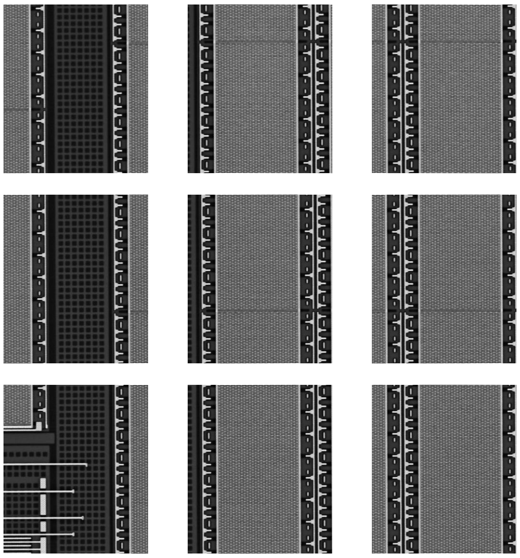

Figure 3: Test Case 1.

errors in the range of -6 to 6 in both the horizontal and vertical direction (see Figure 3).

Unfortunately, given a random initial guess for the $\vec{\Delta}$'s, *fminsearch* could not find the global minimizer. We present two figures of the function space in question. Given two sets of offset errors say $\vec{\Delta}$ and $\vec{\Delta}_{\text{exact}}$, we define the radius r as $\|\vec{\Delta} - \vec{\Delta}_{\text{exact}}\|$. Figure 4 shows the cost as a function of radius for fixed directions $\hat{\theta}$ from the exact solution, i.e., define

$$\hat{\theta} = \frac{\vec{\Delta} - \vec{\Delta}_{exact}}{\|\vec{\Delta} - \vec{\Delta}_{exact}\|} \tag{6}$$

and plot cost as a function of r. Notice that none of curves are monotonic,

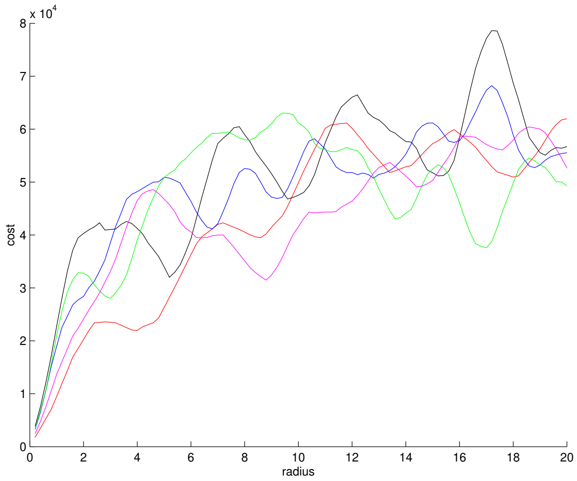

Figure 4: Cost as a function of radius for various directions in offset error space.

in fact, a plot of cost as a function of radius for 1000 $\hat{\theta}$'s reveal that only one or two curves were monotone. The cost function space has a lot of local mimima; this presents problems for most optimization routines.

We also present the plots obtained by keeping most of the offset variables constant at the known exact errors while varying pairs of offset variables (see Figure 5). One can clearly see the existence of many local minimas. Close to the exact solution however, the function spaces are very smooth (see Figure 6). This provides some hope that an optimization scheme can indeed be used, provided a good initial guess is given.

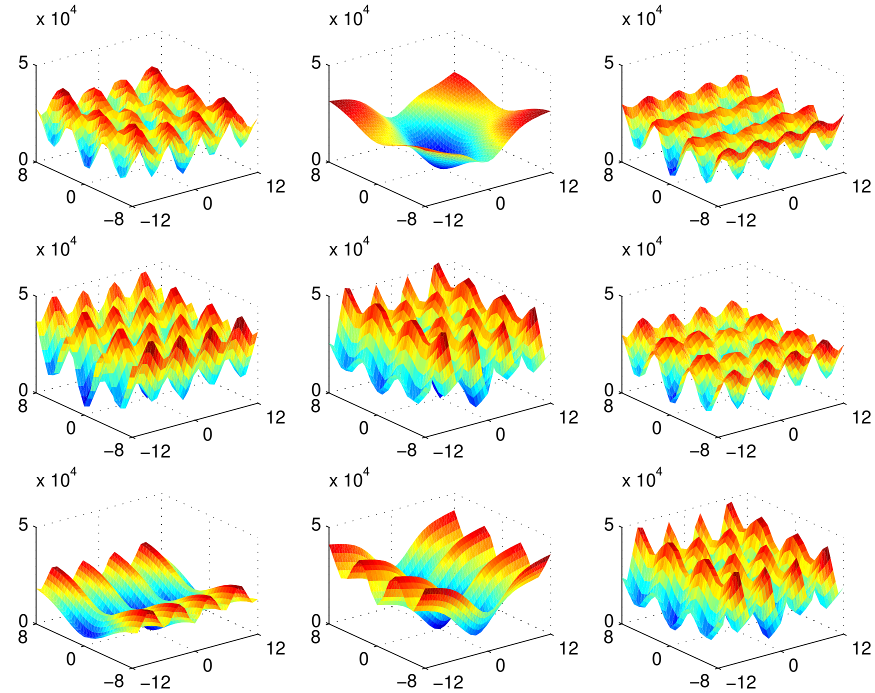

Figure 5: Multiple plots of the cost function space for arbitrary pairs of offset variables. (Other variables held at constant, correct minimizing values.)

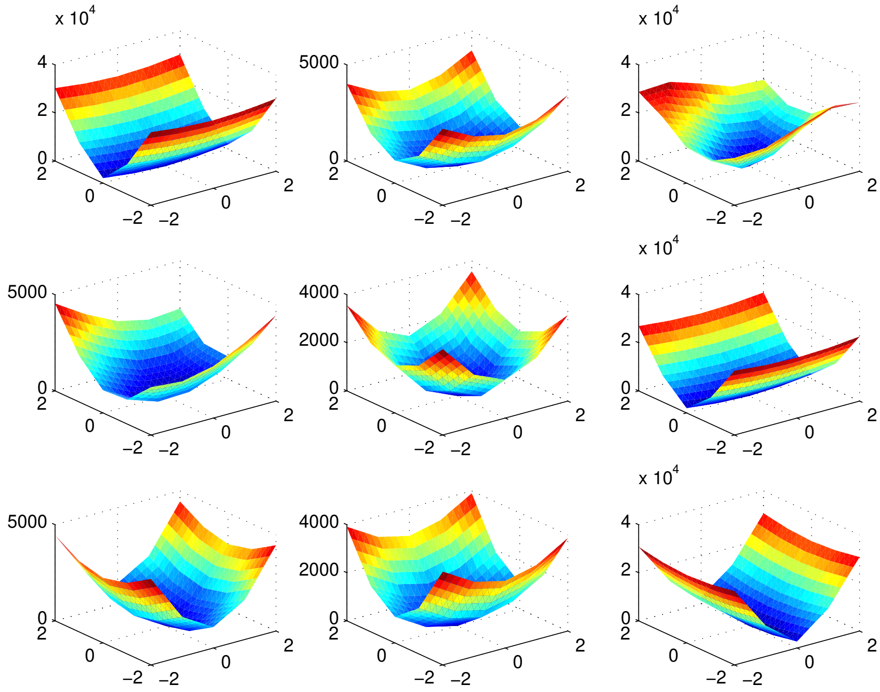

Figure 6: Multiple plots of the cost function space close to the global minimum. Also note the effect of bilinear interpolation.

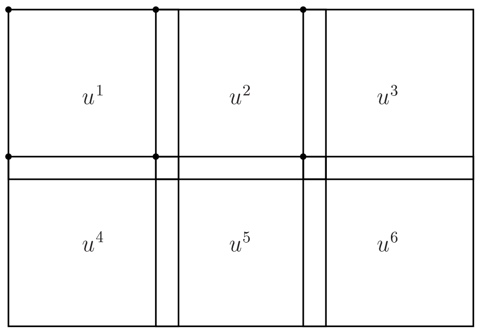

Figure 7: A simple ordering.

3.3 Iterative Scheme

To provide a better initial guess for the optimization routine, we adopt an iterative approach to the problem — this involves stitching one additional image each iteration.

We begin by choosing a particular ordering of the images, $(u^1, u^2, \ldots, u^K)$, $K = M * N$ where each u^k corresponds to a unique u^{ij} and u^k shares at least one common edge with $(u^1, \ldots, u^{k-1})$. A simple ordering is shown in Figure 7. More effective orderings are considered in Section [**Sequential Stitching**[1]]. The following pseudo-algorithm then illustrates the process of sequentially stitching one additional image per iteration:

1. Assume that $\vec{\Delta}^1 \equiv \vec{0}$ and set $k = 2$.
2. Stitch $u^1, \ldots, u^k$ assuming that the known $\vec{\Delta}^1, \ldots, \vec{\Delta}^{k-1}$ are fixed and only $\vec{\Delta}^k$ is allowed to change, i.e., find the optimal $\vec{\Delta}^k$ which stiches the images. (Note, we use a random initial guess for $\vec{\Delta}^k$)
3. Stitch $u^1, \ldots, u^k$ varying $\vec{\Delta}^1, \ldots, \vec{\Delta}^k$, using the values from step 2 as the initial guess.
4. Set $k = k + 1$ and return to step 2.

We acheived correct convergent results in approximately 90% of the trials, however, the convergence was very slow. On a 3×3 set of images (i.e., 14 offset variables), the code took over eight hours[2] to complete.

[1] Please reference to what is currently Section 2 in the "Stitching IC Images" document.
[2] On a 1.2 Ghz Athlon MP.

3.4 Pre-Conditioner

In retrospect, it is not necessary and quite inefficient to take a random initial guess for $\vec{\Delta}^k$ in step 2, Section 3.3. Figure 6 shows that an initial guess within two pixels is likely sufficient for convergence.

A better approach is to do an exhaustive search over a finite set of values for $\vec{\Delta}^k$, and use the $\vec{\Delta}^k$ providing the smallest contribution to the cost function as the initial guess in step 3 of Section 3.3. A natural space of relative offset errors is $\{\vec{\Delta} \in \mathbb{Z}, |\vec{\eta}| \leq |\vec{\beta}|\}$ (where $\vec{\beta} = (\beta_x, \beta_y)$) if we have bounds on the offset errors. If we instead have bounds on the relative error between images, we can do something similar: $\{\vec{\eta} \in \mathbb{Z}, |\vec{\eta}| \leq |\vec{\beta}|\}$ where the inequalities are between u^{ij} and $u^{i-1,j}$ unless $i \equiv 1$, in which case the inequalities are between u^{ij} and $u^{i,j-1}$. An important point to realize is that although our formulation uses a fixed reference grid, the contribution to the cost function between neighbouring images only depends on the relative difference in offset errors between them.

Upon implementation of the pre-conditioner, our code became more robust, with a noticable increase in speed. We were actually quite surprised at how robust the code was. It was unclear if our results were attributed to the artificial construction of test images (cutting a single picture into multiple parts), so we tested our code on a variety of altered images, for example:

- After cutting up the image into smaller test images, each test image was saved as a jpeg with a quality rating of 20. Quality ratings for jpeg range from 0 to 100 where a higher number means less image degradation due to compression.

- Each test image was also subjected to a filter which added gaussian bluring (within a radius of 5 pixels).

- Each test image was subjected to a noise filter. This noise filter "jittered" the image by randomly perturbing some pixels a small distance[3]. This filter actually made the images look quite horrible and hard to optimize.

- The output from the pre-conditioner was perturbed by $|\epsilon| < 1$ to simulate the case when the pre-conditioner does not find the correct values.

[3] "pick" filter in GIMP (www.gimp.org)

Our code still found the global minimum for all the test cases, giving us confidence that it could work when real images are used. Lots of nasty attributes such as lens abberation, difference in contrasts, warping of images would be noticed in real images.

3.5 Caching

It was suggested by an IPSW[4] group member that caching the cost function contribution for various $\vec{\Delta}$'s would provide significant speedup.[5] Another important motivation is the difficulty in storing a 8×14 grid of 8k $\times$ 8k images in memory (7 Gb at one byte per pixel). There are two possible stages to implement caching: pre-caching the cost function contributions before the optimization step or caching during the optimization step (i.e., cache the cost contribution for each new integer $\vec{\Delta}$). The former approach was taken because there is less book-keeping involved. It is likely though that caching during the optimization step will provide further speedup.

For every image u^{ij}, associate two matrices D_L and D_U which correspond to cached contributions from the left and top image respectively. As mentioned in Section 3.4, although our formulation uses the fixed reference grid, the contribution to the cost function from neighbouring images depends only on the relative difference in offset errors. We utilize the experimental constraint that the offset errors are bounded above by $\vec{\beta}$. Thus, D_L and D_U are matrices with $(2\beta_y + 1) \times (2\beta_x + 1)$ elements. Note, in practice, we cache a larger matrix to account for roundoff error.

For a 6×5 grid of 700×600 images (Test Case 2) with $\beta_x = 10$ and $\beta_y = 8$, caching took 7 minutes[6] and optimization took 14 minutes. Note, our non-caching code for this problem was halted after 4 days. A rough estimate for caching the 14×8 problem described above is approximately nine hours. Note that the caching algorithm scales linearly with the number of images and linearly with the size of the overlap region. If the computation time is too long, the caching algorithm is easily parallelizable for a cluster type network.

[4] IPSW 2002 held in Vancouver, Canada, May 28–June 1, 2002

[5] This should translate to significant computational savings during our bilinear interpolation.

[6] On a 1.4Ghz Athlon processor.

4 Results

The optimization code encounters no difficulties fitting Test Case 2. To determine if the code is feasible for a 8×14 problem, a plot of computation time as a function of the number of offset variables is shown in Figure 8. (Note, caching times are not included in the plot). A rough extrapolation

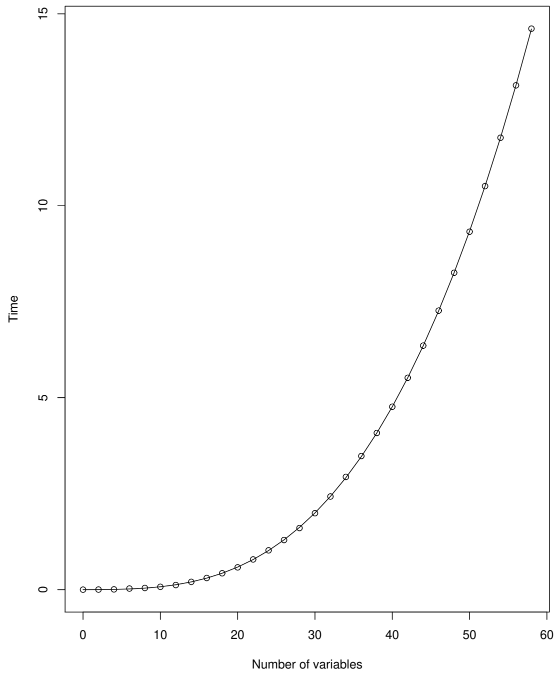

Figure 8: Computation time as a function of fitting parameters. Note: caching times are not included.

from this plot suggests that the 14×8 problem with 240 variables would take about one day.

5 Future Work

- One should *actually try* the fixed reference grid approach of sequentially fitting images. Global optimization may not be necessary.

- Combine the iterative scheme in Section 3.3 with a more effective ordering considered in Section [**Sequential Stitching**[7]].

- Employ a strategy whereby offset variables from images that have enough surrounding neighbours are removed from the optimization process. One could use this process to bound the number of parameters in the optimization.

- Throughout this discussion, we have assumed that the images have been pre-processed, e.g., the images have been rotated correctly. The pre-processing stage can be incorporated into the cost function.

- Recode our algorithm in a more efficient environment such as C or Fortran.

- The cost function is easily modified to fit two different layers of images. A first step would be to detect via-ups and via-downs in each image/layer. (A good segmentation algorithm based on levelsets or wavelets would probably do the job.) One could then calculate the cost function for each layer, sum them together to create the new cost function, and add a penalty term to the new cost function if the vias connect to wires either above or below. The parameter space for two layers of 14×8 images can be handled by a desktop computer in a reasonable amount of time.

- This problem can be reformulated to incorporate the power of parallel processing. For example, the caching routine is easily parallelized.

- For large images, it is not necessary to read in the whole image when optimizing for the offset errors. In fact, less than 5% of the image is required assuming an overlap width of 100 on 8k $\times$ 8k pictures. Incorporating this idea will relax memory constraints.

- Custom tailor a minimizer that pays attention to the pixel scale. It can use sub-pixel information if necessary but ideally should not because sub-pixel information is not important. A first attempt would probably involve a simple algorithm:

    1. Calculate an approximation to the Jacobian (finite differences).

[7] Please reference to what is currently Section 2 in the "Stitching IC Images" document.

2. Calculate an approximation to the Hessian (finite differences).
3. Calculate a search direction based on the Jacobian and Hessian (simple line search).
4. Choose an appropriate step in the search direction - i.e., integer values.
5. Evaluating the Hessian and Jacobian is probably going to be quite expensive, so it's probably better to "Freeze" the Jacobian and Hessian by returning to step 3 and recalculating a search direction.

- If the images were collected in a hexagonal pattern (i.e., acquisition of images for every other row is offset by $\frac{L_x}{2}$) then one would expect a better alignment due to increased connectivity.

- A completely different approach to stitching the images is to use levelsets to join the images. The idea is to warp the image slightly so that all the wires connect together. We are not sure whether this is feasible.

6 Codes

- **driver.m**: Main driver program.

- **set_constants.m**: Defines problem specific constants.

- **read_images.m**: Loads Ymag_shift.jpg and creates a grid of overlapped images to use as our Test Cases.

- **cache.m**: Creates caching matrices for cost contributions from left and top overlaps for each image.

- **cost.m**: Sums over all the contributions from left and top overlaps of all images.

- **cost_contrib_left.m**: Calculates the cost contribution from the left overlap region between image u^{ij} and $u^{i,j-1}$.

- **cost_contrib_up.m**: Calculates the cost contribution from the top overlap region between image u^{ij} and $u^{i-1,j}$.

- **optimize.m**: Iterative optimization scheme described in Section 3.3.

- **pre_cond1.m**: Given a set of relative offset errors and an upper bound on the relative offset errors, pre_cond1.m finds the relative offset error corresponding to the smallest cost function contribution.
- **pre_cond2.m**: Given a set of relative offset errors and an upper bound on the offset errors, pre_cond2.m finds the relative offset error corresponding to the smallest cost function contribution.
- **function_space1.m**: Makes a plot of cost as a function of radius in various fixed directions in offset error space (Figure 4).
- **function_space2.m**: Makes surface plots of the local behaviour of the cost function space for arbitrary pairs of offset variables (Figures 5, 6).
- **row_to_matrix.m**, **matrix_to_row.m**: Utility functions.
- **Ymag_shift.jpg**: Test image supplied by Semiconductor Insights.

7 Acknowledgments

Thanks to Isabelle Dechene and Fabien Youbissi, our IPSW sub-group members; Jim Verner for proof-reading and constructive criticism; Jason Nielson for help with extrapolation techniques.

8 Workbook

In this section, we list the ideas that were attempted but were unsuccesfully implemented, in hopes that others can improve them or learn from our mistakes.

8.1 Alternative Cost Function Construction

For small scale pictures, the following algorithm was developed based on the fixed reference grid idea. It involves constructing the following two pictures.

1. For a given set of offsets, $\Omega = [\vec{\Delta}^1, \vec{\Delta}^2, \ldots, \vec{\Delta}^{MN-1}]$, stitch the pictures a row at a time treating the overlap region from the image on the right as the correct overlap.

2. Next, stitch each rows one at a time assuming that the overlap regions from the bottom images are correct.

3. We have succesfully formed image A.

4. For the same set Ω, stitch the pictures one column at a time assuming that the ovelap regions from the top images are correct.

5. Next, stitch the columns together one at a time assuming that the overlap regions from images on the left are correct.

6. We have now formed image B.

7. If the offsets errors Ω are correctly choosen, A == B. To measure how bad the the offset errors are, we can calculate

$$C(\Omega) = \sum_p \sum_q [A(p,q) - B(p,q)]^2 \tag{7}$$

Although the above algorithm is easy to implement, it is not ideal because it requires $\mathcal{O}(L^2)$ calculations, where L is the image size. Furthermore, because offsets are not necessarily integer values (see Section 3.1), an expensive N-D interpolation scheme is required (where N is the number of variables).

8.2 Image Neighbours

Before switching to the fixed reference grid discussed in Section 2, we considered the position of each image to be measured relative to the images above and left. This approach turned out to be fundamentally flawed, however it is presented here for completeness (and to hopefully prevent others from repeating the same mistake!)

For every image u^{ij}, associate offset errors $\vec{\Delta} = (\Delta_x^{ij}, \Delta_y^{ij})$, where Δ_x^{ij} is measured relative to the image to the left (i.e., $u^{i-1,j}$) and Δ_y^{ij} is measured relative to the image above (i.e., $u^{i,j-1}$). The overlap regions are then calculated as shown in Figure 9.

Given a particular u^{ij} and associated $\vec{\Delta}$, the contribution to the cost function is again given by (2) but note that d^{ij} depends explicitly on only $\vec{\Delta}$ (in contrast with d^{ij} as calculated in Section 2). The total cost function is then calculated as in (3) by summing over all i, j.

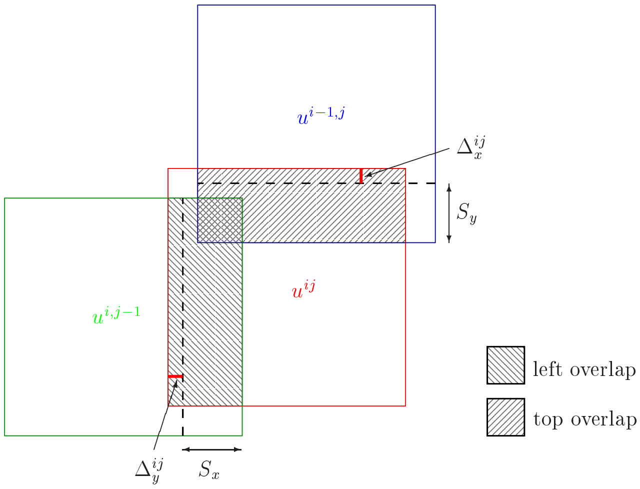

Figure 9: Overlap region for image neighbours approach.

8.3 Image Neighbours Flaw

Given three images, u^{ij}, $u^{i-1,j}$, $u^{i,j-1}$, both overlap regions (between image pairs $(u^{i-1,j}, u^{ij})$ and $(u^{i,j-1}, u^{ij})$) depend on *both* Δ_x and Δ_y (see Figure 9). However, the correct overlap region for each pair of images may be mutually exclusive. That is, the "correct value" for $\vec{\Delta}$ for the pair $(u^{i-1,j}, u^{ij})$ may not be the "correct value" for the pair $(u^{i,j-1}, u^{ij})$. This is best illustrated in the example shown in Figure 10.

One obvious solution might be to add additional unknowns so that each overlap region has its own set of $\vec{\Delta}$ values. However, this leads to an over-determined problem. Note that given a set of images, there is usually a unique alignment, so an over-determined system is a redundant approach to solving the problem of creating one final composite image. Furthermore, given a solution to the over-determined problem, it may be impossible to construct a single unique layout for the images.

Our solution to this problem was to abandon the image neighbours approach and adopt the fixed reference grid strategy described in Section 2.

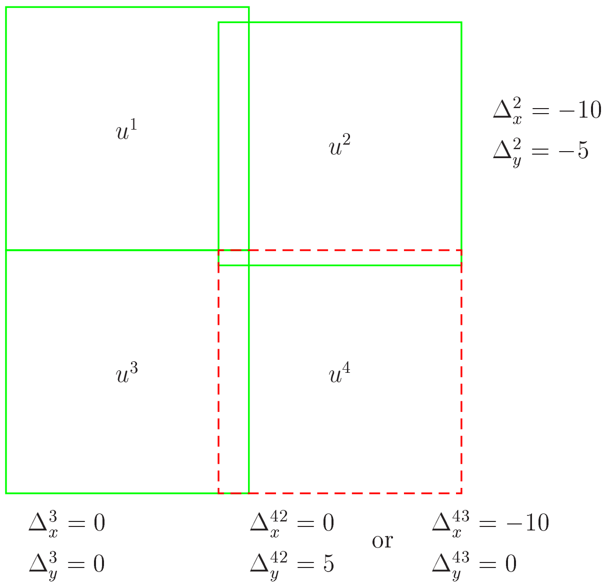

Figure 10: Fundamental flaw in image neighbours approach. The values of $\vec{\Delta}$ for u^4 differ depending on whether they are measured relative to the left or top neighbour. Note, the notation $\vec{\Delta}^{4(\cdot)}$ refers to value of $\vec{\Delta}^4$ with respect to $u^{(\cdot)}$, and that $S_x = S_y = 0$ for clarity.

8.4 Further Discussion

The creation of a single composite image is not the final goal of this project; rather the goal is to create a schematic diagram of the circuit on a given IC chip.

The current approach is to first create a single composite image and then convert this image to a vector format through some sort of feature detection. That is, there exists a (black box) code which takes a large composite image and produces this vector format. If this feature detection code were modified, then the over-determined image neighbours approach suggested at the end of Section 8.3 could provide a superior approach for generating the schematic diagram.

Consider the following:

- Run the "over-determined" image neighbours code to create a data structure where every image knows its offset errors relative to each of its four neighbours (or six neighbours if a hexagonal grid of images is used).
- The feature detection code would then work with this data structure by following traces. Whenever it encountered the edge of an image, it would simply consult the data structure about the appropriate neighbouring image and continue the trace onto that image.