EFFICIENT IMAGE QUILTING: EARLY STOPPING AND COLOR PERMUTATION

Dominic Steiner, Etienne Mettaz, Oleh Kuzyk, Ondrej Cernin

Department of Computer Science
ETH Zurich, Switzerland

ABSTRACT

Image quilting is a common texture synthesis algorithm for image generation. However due to its computationally intensive nature, it is prone to high runtimes. In this paper, we explore various code implementations and their effect on performance. Further, we present early stopping and color permutation as improvements to the algorithm.

1. INTRODUCTION

Within the field of computer graphics, the task of creating new images has been an ongoing challenge which has seen various methods be introduced. One such approach for image generation is texture synthesis, where instead of completely constructing an image from scratch, an input image is provided to serve as a source for sampling of textures. After a sample texture is extracted from the model image, a texture synthesis algorithm should be able to extend the sample to unlimited image data, such that the final result appears similar, but not identical to the original input.

Such algorithm, referred to as "image quilting", was first presented by Efros and Freeman [1]. The image quilting algorithm relies on the simple idea of continually extending the synthesized image with blocks from the input image that mutually align optimally at their respective borders.

Unlike other methods based on complicated analysis of the histograms of filter responses to the image at various scales and orientations [2] or by matching statistics such as marginal and joint distribution [3] and other Markov random field models [4], the image quilting algorithm identifies suitable image patches only by calculating the difference of red, green and blue color intensities of corresponding pixels at the overlapping regions.

Another component of image quilting that further enhances the simplicity of the algorithm is the use of blocks at each synthesization step. Compared to schemes where only a single pixel is added at a time such as [5], image quilting is able to preserve spatial relationship between neighboring image pixels. Further, the one-pixel-at-a-time algorithms are prone to extremely slow runtimes due to their high computational cost using image search for each new single pixel.

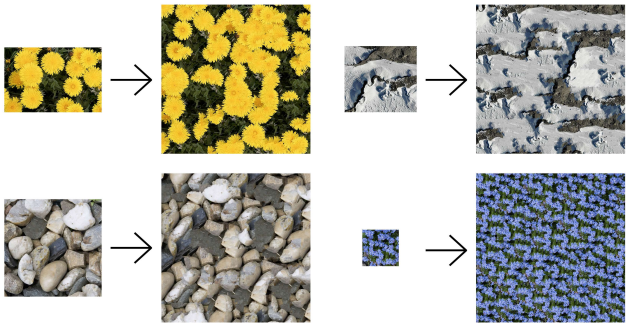


Fig. 1. Example input and outputs of image quilting.

These two aspects have made image quilting a common method for image generation as part of data augmentation for machine learning models [6]. Despite the simplicity of the algorithm, the computational runtime even for small images is quite high. Wang and Mecon report an average runtime exceeding 20 seconds for a 250 x 250 pixel image synthesis using their published code implementation [7]. Considering most machine learning models use similarly sized images as part of their training dataset, this would for example lead to an elongated process exceeding half an hour for a simple data augmentation of 100 new images. Additionally, the concept of image quilting has been used in simulations of earth textures [8] and for building geological models [9].

However further work on image quilting since the original paper by Efros and Freeman has mainly focused on its applications and improvements to the correctness and aesthetics of the output, such as by introducing Poisson blending at the block edges [10], while possible improvements to the algorithm's runtime have been overlooked outside of a parallel implementation from Pu [11].

Therefore as part of this paper, we experimented with multiple code implementations of the algorithm and analyzed their runtime based on input images of various types and sizes, along with runtime dependence on output image parameters such as block-size and overlap width. Further, we propose enhancements to the quilting algorithm such as early stopping and color permutation to avoid unnecessary calculations and costly data movement. Together with smart

coding practices, we are able to achieve a speed-up of over 30x compared to a baseline code version based on Wang and Mecon's implementation [7].

2. BACKGROUND

In this section, we introduce the image quilting algorithm in more detail and perform a cost analysis.

Image Quilting. The image quilting algorithm takes as input a sample texture image, which it further synthesizes into a new larger output image. The process of quilting is composed of two main repetitive procedures, which are block selection and block merging. The algorithm relies on blocks within the input image as the key component of the synthesis, as the new generated image is built block by block. Therefore block-size b defining the size of the square block of pixels is an important parameter, as it not only defines the size of the block added to the output at each step, but also the size of the block being considered to be added from the original input. Further we define n to be the number of blocks added in each of the two dimensions within the output image. The next parameter o controls the overlap width, which sets the number of pixels to be compared within the merging process. Lastly we define w and h to be the width and height of the input image in terms of pixels.

The first step of the image quilting algorithm consists of randomly selecting a block of size $b \times b$ from the input image, and copying it to the left-top corner of the eventual output image. Next, the remaining n-1 remaining blocks within the top row are filled out, before further selecting blocks for the n-1 remaining rows below. Unlike the initial first block which was selected completely randomly, all further blocks are chosen based on their similarity to the neighboring blocks in the output based on the pixels within the corresponding overlap area OA. The most similar blocks are calculated to be those with the lowest sum of square differences

$$E_{tot} = \sum_{\in OA} (R^{on} - R^{ic})^2 + (B^{on} - B^{ic})^2 + (G^{on} - G^{ic})^2$$

where R,G and B stand for the intensity of the red, green and blue colors of a given pixel in the "output neighbor" and "input candidate" blocks.

The overlap area is defined as the subsection of $o \times b$ pixels at the block's edge. The algorithm presents three distinct cases "left", "above" and "corner", which designate the location of the overlap area as illustrated in figure 2. In the "left" case, the overlapping pixels are alongside the vertical boundary of the blocks, with the candidate's left o columns of pixels overlapping the left neighbor block's right subsection. Similarly for the above case, the candidate block's upper o row of pixels along the horizontal boundary overlaps the above neighboring block's bottom subsection.

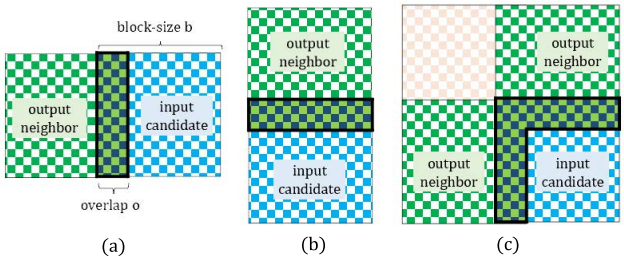


Fig. 2. Illustration of "left", "above" and "corner" overlap areas for error calculation.

In the corner case, the overlap is defined as the union of overlaps from the left and above cases, forming an L shape. The sum of square differences, further simply referred to as error, has to be calculated for all the potential candidate blocks. With an input image of $w \times h$ and block-size b, there are $(w-b) \times (h-b)$ total blocks which can be chosen, and for which the error has to be calculated at each block selection step. After the errors are calculated for all blocks, only blocks which exceed the lowest calculated error by 10% of the minimum error are considered as the final candidates for selection. From these final candidates blocks, a single final block fb is chosen randomly. This is done to ensure that the output image does not end up being an identical copy of the input image, which would occur if simply the block with the minimum error would be copied each iteration. Further, if the tolerance for the difference in error was too large, the adjoining blocks would be too dissimilar and create noticeable borders within the output.

Once the next block fb to be added to the output is selected, the algorithm moves to the block merge step. In order to optimize the aesthetics of the output image and remove potentially harsh transitions at the borders between blocks, the quilting algorithm uses the minimum cut algorithm to calculate the minimum cut path minimizing the error between two blocks. This is done by calculating an error matrix, with each value being the squared difference of the three RGB color values between the selected input candidate and output neighbor block. This is done for all pixels within the overlap area:

$$e_{i,j} = (R^{on}_{i,j} - R^{fb}_{i,j})^2 + (B^{on}_{i,j} - B^{fb}_{i,j})^2 + (G^{on}_{i,j} - G^{fb}_{i,j})^2$$

Then dynamic programming is used to compute the cumulative minimum error E for all paths. In the case of the "left" overlap case, where we have overlapping regions along the vertical border, we can compute E as:

$$\begin{aligned} E_{0,j} &= e_{0,j} \\ E_{i,j} &= e_{i,j} + min(E_{i-1,j-1}, E_{i-1,j}, E_{i-1,j+1}) \end{aligned}$$

Finally the minimal path can be extracted by finding the minimum value $E_{b-1,j}$ in the bottom row of the E matrix,

and by recursively finding the lowest E value from the adjoining pixels in the above rows. Similarly the min-cut algorithm can be adapted for the "above" and "corner" cases.

Cost Analysis. The whole quilting algorithm can be split into two main parts: error calculation during the block selection process, and the min-cut algorithm during block merging. Both functions are dependent on parameters of both the input and output images as defined above. Let us define asymptotic time complexity for both functions:

$$T_{\text{error\_calc}}(n, b, w, h) = \mathcal{O}(w \cdot h \cdot n^2 \cdot b^2) \quad (1)$$

$$T_{\text{min\_cut}}(n, b) = \mathcal{O}(n^2 \cdot b^2) \quad (2)$$

where w and h correspond to the width and height of the input image. Additionally to simplify, we consider the overlap to be bounded by a constant of the block-size, i.e. $o = \mathcal{O}(b)$. Therefore, we can state that for sufficiently large parameters, the runtime of the whole quilting algorithm will be dominated by the error calculation.

Our implementation uses only integer-based operations since initially pixel values are stored as 8-bit integer values. Apart from that, we use 16-bit and 32-bit integer data types after applying subtractions and multiplications correspondingly. All in all, we use integer additions/subtractions, multiplications, comparisons, and divisions. Thus, we define our cost measure in the following way:

$$C(w, h, b, n) = C_{\text{add}}N_{\text{add}} + C_{\text{mul}}N_{\text{mul}} + C_{\text{cmp}}N_{\text{cmp}} + C_{\text{div}}N_{\text{div}}$$

In our case, we count all the operations together, i.e. $C_{\text{add}} = C_{\text{mul}} = C_{\text{cmp}} = C_{\text{div}} = 1$. The reason why this assumption works with divisions is that we have few integer divisions and all of them are simply divisions by a constant of 2, which can be easily interpreted as a bit shift. Therefore, as a result, our cost measure is as follows:

$$C(w, h, b, n) = N_{\text{add}} + N_{\text{mul}} + N_{\text{cmp}} + N_{\text{div}}$$

3. OUR PROPOSED METHOD

The baseline implementation of the image quilting algorithm based on [7] consists of two major procedures, both of which offered room for performance improvements.

Min-Cut. During the project, we made multiple optimizations to the block merge step. However, each of these optimizations only resulted in a very slight improvement in the overall runtime of the quilting algorithm.

The baseline algorithm only had a single implementation of the dynamic programming algorithm for the three different merge cases, and relied on transposing the input and resulting mask, which lead to unnecessary copying. To eliminate the need for these transpose operations and memory allocations, we wrote a second DP algorithm that could compute the horizontal min cut path directly. We could also eliminate some memory allocation by reusing the min-cut overlap error matrix in the DP step instead of creating a new DP matrix from scratch, as in the baseline implementation.

The baseline implementation uses int arrays to store the cut mask computed by the DP algorithm. However, because we only need to store three distinct values (-1 to select the corresponding pixel from the output image, 0 to mark the cut path, and 1 to indicate that a pixel has to be copied from the selected block in the input image), we reduced the size of this cut mask matrix by switching to the 8-bit char type.

While we were able to improve the efficiency of the min-cut portion of the quilting algorithm with these optimizations, we found that the effect on the overall runtime was very minimal. Further analysis revealed that the min cut part of the algorithm was only responsible for about 0.4% of the overall runtime. As a result, we decided to shift our focus to the first part of the algorithm. This decision allowed us to prioritize our efforts and focus on the parts of the algorithm that would have the greatest impact on overall performance.

Image format. Our baseline implementation used an image struct that contained an array of RGB structs, each containing the three color components of a pixel as unsigned 8-bit integers.

```
struct RGB {
    unsigned char r;
    unsigned char g;
    unsigned char b;
};

struct Image {
    int width;
    int height;
    struct RGB *data;
};
```

However, using arrays of structs can prevent the compiler from making certain optimizations. To address this issue, we switched to an image format that uses three arrays of unsigned 8-bit integers, one for each color channel.

```
struct Image {
    int width;
    int height;
    unsigned char *r_data;
    unsigned char *g_data;
    unsigned char *b_data;
};
```

With this new deinterleaved image format, the RGB struct is no longer required, resulting in more efficient code. By optimizing the data structure used in the algorithm, we were able to achieve a speedup of 4x compared to the baseline.

Vectorization. We attempted to vectorize the error calculation function using Intel intrinsics in several different

ways. Since we are working with integer values, it is important to ensure that we do not cause any overflows. In memory, an image is stored as three unsigned 8-bit integer arrays, one for each color channel. To calculate the overlap error for a block, we need to compute the squared difference for each pixel and color channel, and then sum up all of these squared errors. The final result must be a 32-bit integer to avoid overflows.

There are different options for converting the value. Our initial vectorized error calculation function loads 16 8-bit integers from memory into a 128-bit register. We then convert the last 8 integers in this register into 32-bit signed integers, which are stored in a 256-bit AVX register. We then permute the initial 128-bit register and run the conversion instruction again to convert the remaining integers to 32-bit. Finally, all remaining operations can be performed using 32-bit values.

```
__m128i src = _mm_loadu_si128(
    (const __m128i *) (src_r_data + src_idx)
);
__m256i src_0 = _mm256_cvtepu8_epi32(src);
r_src = (__m128i) _mm_permute_pd(
    (__m128d) src, 0b01
);
__m256i src_1 = _mm256_cvtepu8_epi32(src);
// ...
```

However, converting to 32-bit integers at an early stage has the disadvantage of reducing the number of operations that can be performed with a single vector instruction.

In the next iteration of the vectorized code, we still load 16 8-bit integers from memory into a 128-bit register, but then we convert them to 16 16-bit signed integers. These 16-bit values fit into a single 256 AVX register, so we no longer need to use permutations. We can then perform subtractions using 16-bits. With the $\_mm256\_mullo\_epi16$ and $\_mm256\_mulhi\_epi16$ instructions, we can square the error and extend it to 32-bit integers simultaneously. Below is a simplified code snippet:

```
__m128i src = _mm_loadu_si128(
    (const __m128i *) (src_r_data + src_idx)
);
__m128i out = _mm_loadu_si128(
    (const __m128i *) (out_r_data + out_idx)
);

__m256i a = _mm256_cvtepu8_epi16(src);
__m256i b = _mm256_cvtepu8_epi16(out);

a = _mm256_sub_epi16(a, b);
__m256i err_lo = _mm256_mullo_epi16(a, a);
__m256i err_hi = _mm256_mulhi_epi16(a, a);

__m256i res_0 = _mm256_unpackhi_epi16(
    err_lo, err_hi
);
__m256i res_1 = _mm256_unpacklo_epi16(
    err_lo, err_hi
);

err = _mm256_add_epi32(err, res_0);
err = _mm256_add_epi32(err, res_1);
```

Early stopping. The baseline algorithm for the quilting technique involves calculating the overlap error for all potential blocks, and then selecting a random block that has an overlap error within a tolerance of the minimum error. However, there is a simple optimization that can be made to reduce the number of computations and data transfer.

Since we are only interested in whether or not a block should be considered in the final random selection, it is not necessary to calculate the exact overlap error for each candidate block. Instead, by keeping track of the current minimum error during the calculation of the overlap error, we can abort the error calculation for a block as soon as it exceeds the current error tolerance. Due to the current minimum error only decreasing during the iteration over the candidate blocks, we can be sure that this optimization will not affect the final result of the quilting algorithm.

This optimization reduces both the amount of data transferred and the number of calculations involved. It is worth noting that with this optimization, the number of operations no longer depends solely on the input size and the algorithm parameters, but also on the exact pixel values of the input image and the random block selections during the algorithm. However, in real-world scenarios, we observed significant performance improvements when using this optimization. Overall, this optimization allows us to achieve equal results with less computation time and data transfer, making the quilting algorithm more efficient.

Color permutation. After observing a significant performance gain with early stopping, we wanted to further explore how to make this optimization even more effective.

During the overlap error calculation for a single candidate block, we initially sum up the error of the red channel over all the pixels in the block, followed by the green and blue channels. However, we discovered that for most images, the channels do not contribute equally to the overlap error. For instance, in an image full of blue and green, the red color error may not be as significant. To maximize the benefits of early stopping, we should sum up the pixel error in decreasing order, as this would require the least operations and memory transfers to reach the error threshold. At that point, we can abort the error calculation of the current block. While it is not possible to know the error in decreasing order for each block, we experimented with switching the order in which we compute the error of each color channel. We found that for images with lots of red, the default RGB color order is the fastest, but for images with lots of

blue, a BGR order is significantly faster.

Through these experiments, we discovered that the optimal color calculation order depends on the input image, and that there is no single optimal order for all possible images. To address this issue, we implemented a pre-processing step that sums up all the color intensities for each individual color channel in the image. This allows us to permute the color arrays in such a way that the dominant color error is always calculated first.

By taking advantage of this optimization, we can even further reduce the number of operations and memory transfers required to reach the error threshold, leading to faster and more efficient image generation.

Loop order optimizations. We also experimented with modifying the loop order of the error calculation loops to improve cache performance. The error calculation uses data from three different locations: the source image, the current block of the output image, and the errors array that stores the overlap error of all the candidate blocks. The baseline used the order shown in the simplified code below.

```
for (int y = 0; y < max_src_y; ++y) {
  for (int x = 0; x < max_src_x; ++x) {
    int error = 0;
    // ...
    for (int by = 0; by < block_size; ++by){
      for (int bx = 0; bx < overlap; ++bx){
        src_idx =
          (by + y) * src->width + bx + x;
        out_idx = (by + out_coord.y)
          * out->width + bx + out_coord.x;
        error += rgb_sq_error(
          src->data + src_idx,
          out->data + out_idx
        );
      }
    }
    // ...
    errors[err_idx] = error;
  }
}
```

The complete overlap error is calculated for each block and then stored in the errors array. This is the best possible order for the errors array because all values are only written once. It also allows for the early stopping optimization.

Nonetheless, we experimented with a different loop order. Assuming the block-size is relatively small, optimizing the loop order of the output image does not bring major benefits, because we are only accessing a $b \times b$ section of the output image. Instead, we optimized the loop order for the input image. This optimized loop order accesses each pixel in the image only once and compares it with all output pixels that it could potentially overlap. The error is then added to the errors array in the appropriate place so that, in the end, the entire overlap error is aggregated in the errors array. As a result, we achieve the best possible loop order for the input image, but this order makes the access pattern worse for the errors array. Below is a simplified code snippet of the optimized loop order. Note that all of the code that handles the edge cases at the borders of the input image has been removed for simplicity.

```
for (int y = 0; y < max_src_y; ++y){
  for (int x = 0; x < max_src_x; ++x){
    // ...
    src_idx = y * src_width + x;
    src_r = src_data_r[src_idx];
    src_g = src_data_g[src_idx];
    src_b = src_data_b[src_idx];
    for (int by = 0; by < overlap; ++by){
      for (
        int bx = 0; bx < block_size; ++bx
      ){
        out_idx = (by + out_y)
          * out_width + out_x + bx;
        err_idx = (y - by)
          * max_src_x + x - bx;

        e_0 = src_r - out_data_r[out_idx];
        e_1 = src_g - out_data_g[out_idx];
        e_2 = src_b - out_data_b[out_idx];
        e_0 = e_0 * e_0;
        e_1 = e_1 * e_1;
        e_2 = e_2 * e_2;
        errors[err_idx] += e_0 + e_1 + e_2;
      }
    }
    // ...
  }
}
```

It is worth noting that the worse access pattern of the errors array eliminates the advantage of using early stopping. This is due to multiple errors being aggregated at the same time, making it difficult to keep track of the current minimum. Additionally, this loop order makes it difficult to stop aggregating the error for certain blocks, due to the calculations no longer being grouped by blocks. Despite these disadvantages, we were interested in testing the performance of this loop order.

4. EXPERIMENTAL RESULTS

We conducted a comprehensive evaluation of our optimizations on three different systems and three different images (see Fig. 3). The radishes image (Fig. 3(a)) is mostly red in color, and the blue flower image (Fig. 3(b)) is mostly blue. We selected these two smaller images to evaluate the effectiveness of our color permutation optimization. Additionally, we included the significantly larger dandelion image (Fig. 3(c)) to test the scalability of our optimizations.

(a) radishes 192x192

(b) blue flowers 192x192

(c) dandelion 1024x683

Fig. 3. Input images used for evaluation

By evaluating on a range of images and systems, we were able to demonstrate the effectiveness and versatility of our optimizations. Details about the test systems and compiler flags are listed below.

Experimental setup.

- AMD Ryzen 9 3900x @ 3.6 GHz
  - **cache** L1: 64 KB, L2: 512 KB, L3: 64 MB
  - **compiler** gcc -Wall -Wextra -Wpedantic -O3 -ffast-math -march=native –mavx
- Intel i7-8550U @ 1.8 GHz
  - **cache** L1: 64 KB, L2: 256 KB, L3: 8 MB
  - **compiler** gcc -Wall -Wextra -Wpedantic -O3 -ffast-math -march=native –mavx
- Intel i5-1135G7 @ 2.4 GHz
  - **cache** L1: 96 KB, L2: 1280 KB, L3: 8 MB
  - **compiler** icx -Wall -Wextra -Wpedantic -O3 -ffast-math -march=native -mavx -qopt-zmm-usage=high
  - supports AVX-512

Optimization versions.

We conducted our experiments for multiple intermediate optimization steps to check and to show the effect of each of them individually. This way we obtained the following code versions:

- **Opt1:** Standard loop unrolling to error calculation with smarter indexing to decrease computations.
- **Opt2:** Applied blocking for cache optimization.
- **Opt3:** Optimized min-cut function and removal of unnecessary transpose operations by implementation of two case-specific functions cases.
- **Opt4:** Optimized minimum error in index search.
- **Opt5:** Image storage format change from a single interleaved RGB array to 3 separate arrays for each color. Cut mask data type change from chars to ints.
- **Opt6:** Vectorized min-cut overlap error calculation and cut masks merge.
- **Opt6a:** (*branched from opt6*) Vectorized error calculation function for 16-bit integers and improved block finding algorithm.
- **Opt6b:** Initial version of early stopping optimization.
- **Opt6c:** Early stopping plus separate calculation of each color's error.
- **Opt6d:** Improved source image locality by optimizing loop order for input image access.
- **Opt7:** (*another branch from opt6*) Loop unrolling and smarter indexing to decrease computations for separate RGB array instead of data structure version.
- **Opt8:** Vectorized error calculation for 32-bit ints.
- **Opt9:** Finalized early stopping with vectorization.
- **Opt10:** Implementation of automatic color permutation technique.
- **Opt11:** Minimum error predictions.

For further experiments and more detailed analysis, we selected the optimization that are the most interesting from the optimization perspective and that produced the biggest runtime/performance improvement. These are: opt4, opt5, opt6a, opt6b, opt6c, opt6d, opt9, and opt10.

Runtime analysis. In Figure 4, we can see the runtime plot. We can see the dependency between the runtime on AMD processor and block-size b. Overlap size was chosen to be $o = 0.5b$ (here and for further measurements) and output size n was set to 12 blocks x 12 blocks. As an input image here we used image of red radishes and we measured all the above-mentioned optimizations.

We used different styles to distinguish between various optimizations. Dotted lines represent optimizations that use the interleaved image format while dashed lines represent non-interleaved format. Solid lines were used to indicate the use of early stopping. Dash-dotted lines represent modified loop order. We marked vectorized versions with X.

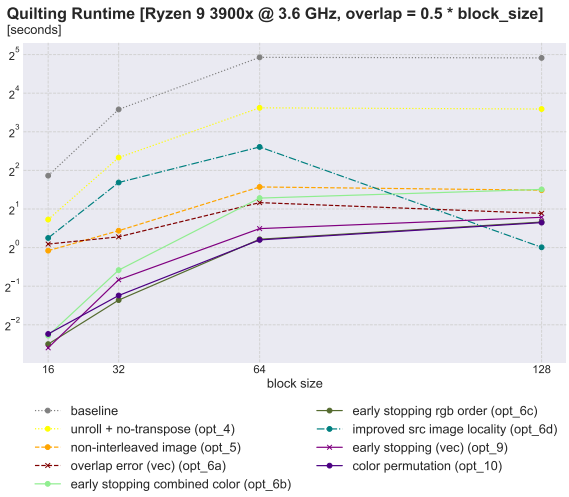


Fig. 4. Runtime of different optimizations

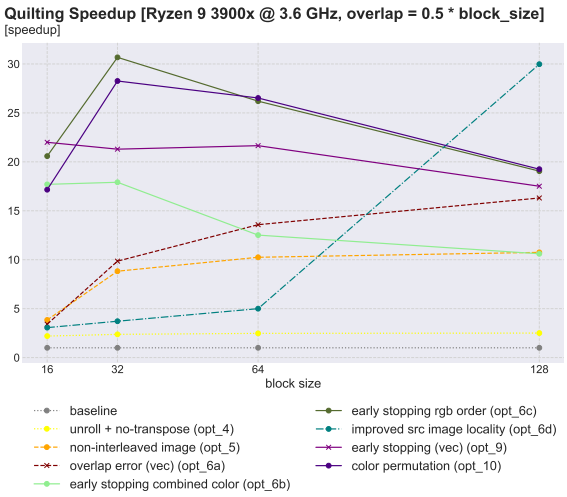
We can see that as expected the runtime increases as the block-size increases since the number of calculations also increases correspondingly.

Effect of the color permutation on the speedup. In Figure 5, we can see speedup plots for different images on AMD processor. In all cases, we used output size n of 12 blocks x 12 blocks and set overlap o as half of the block-size. We measure dependency between block-size b and speedup all the previously mentioned optimization versions.

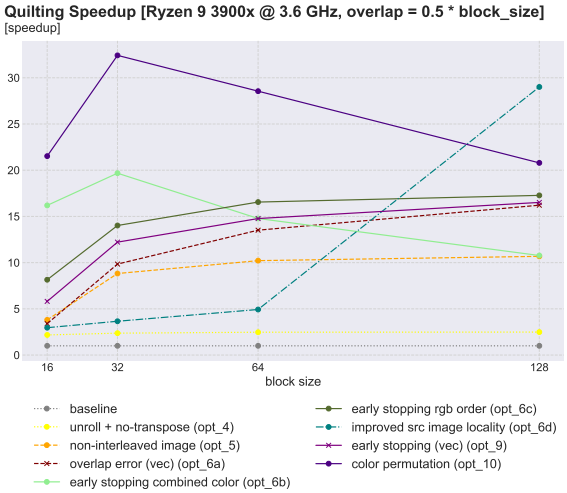
Figure 5(a) plots the speed-up for the red radishes image. We can see that with optimization 6c which incorporates early stopping technique, we achieve a slightly better performance compared to the final optimization 10, which already involves automatic color permutation. The reason behind this lies in the fact that the RGB order proves to be the most effective for the red radishes image since red is the dominant color. Consequently, the color permutation optimization simply involves additional overhead for color analysis and does not yield any benefits here.

On the other hand, when we conduct a similar experiment for blue flowers (see Figure 5(b)), where blue is the dominant color, we observe a slowdown in all optimizations except for optimization 10 with color permutation, which remains fast. This is due to other optimizations being negatively impacted by the sub-optimal RGB order. The color permutation technique enables the switch to BGR color order and fully benefits from the early stopping.

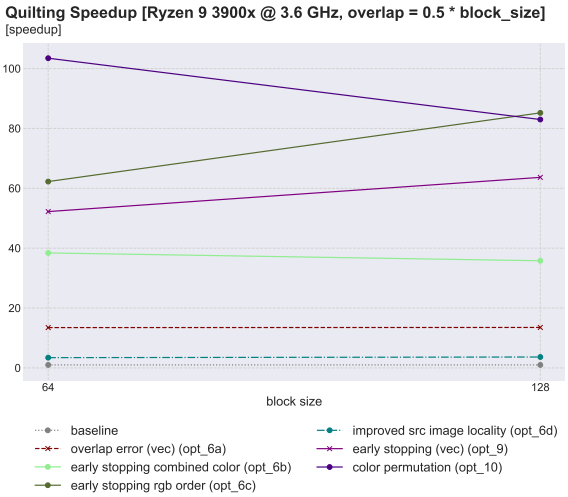
Effect of source image locality. From Figure 5, we can also see that optimization 6d, which enhances source image locality, shows a significant improvement in performance when using a larger block-size, especially at block-size of 128 pixels. This would make this optimization highly suitable for quilting images with large block-sizes.



(a) radishes (192x192)



(b) blueflowers (192x192)



(c) dandelions (1024x683)

Fig. 5. Speedup for different input images

Fig. 6. Speedup on Intel processor with AVX-512

To further evaluate this effect, we conducted experiments with a larger input image of dandelions (see Figure 5(c)). We used the same setup and the same parameter set as in the previously. Surprisingly, we observed that speedup does not translate for this optimization for larger input sizes. In fact, the speedup practically vanished, with this optimization only marginally outperforming the baseline. However, it is worth noting that early stopping still remains effective enabling us to achieve speedup of 100 compared to baseline combined with early stopping and color permutations.

It is also interesting to see that the effect of early stopping with color permutations surpassed the efficiency of the fastest vectorized code, since optimization 9 is only the third most effective as seen in the plot.

Intel processor with AVX-512. We also conducted measurements of speedup on Intel i5 processor with AVX-512 capabilities. All the other setup parameters remained the same. Interestingly, we observed a notable difference in the results as seen in Figure 6. Specifically, code optimization 6b which incorporates early stopping, but with all colors combined had the best speedup, contrary to our previous measurements. This unexpected outcome can be a result of the support of AVX-512 and potentially a more intelligent compiler. Due to limited access to this system, we did not have time to further evaluate these results. It is worth noting that we did not observe such performance improvement for this code version on a different Intel system which indicates the specificity of the hardware/software configuration of the given system.

Roofline analysis. We also conducted a roofline analysis for the various implementations. The result plot for the blue flowers image (192x192), $b = 64$, $o = 16$, and $n = 12$ blocks can be seen in Figure 7.

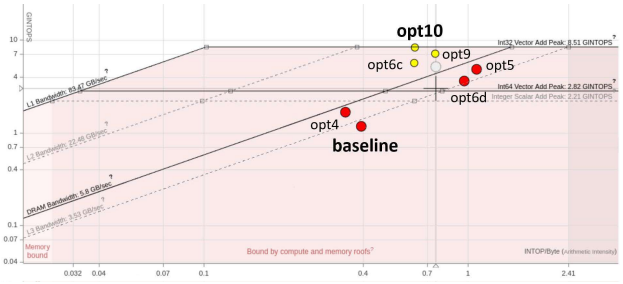


Fig. 7. Roofline plot of the performance

Our initial (baseline) implementation had an operation intensity of 0.39 operations per byte. The performance was around 40% of the integer scalar peak and only 14% of the integer vector peak performance. This clearly indicates that there was considerable room for improvement. For our final implementation version (opt10), we see a performance of 99% of the 32-bit integer vector additions peak with operation intensity of 0.63 integer operations per byte. Therefore, it represents 7-times improvement in performance compared to the baseline implementation while nearly doubling the computational intensity. Interestingly, we saw that maximizing the operational intensity did not necessarily result in the best performance. This is particularly evident in case of opt6d, since it was designed to enhance source image locality, but in the end it did not yield the best performance.

5. CONCLUSION

In this paper, we explored different methods to improve the runtime of the image quilting algorithm for texture synthesis. We discovered that data format choice is critical, as the simple switch from a single interleaved RGB array to three separate color arrays led to a quadruple increase in performance. Further coding optimizations using techniques such as loop unrolling, improvement of cache locality, and vectorization led to different performances based on the the input image size and content, output image parameters, and system environments. Next, we introduced early stopping and color permutation to improve the overall algorithm by reducing the number of calculations and memory data transfers. The utilization of these two techniques however leads to major performance improvements in all test cases. Overall, our final code implementation provides a speed-up exceeding 30x the baseline implementation runtime for small images, and up to 100x for large input images.

6. CONTRIBUTIONS OF TEAM MEMBERS

Dominic. Implemented the optimizations to the min-cut part, which includes the min-cut algorithm for the horizontal cut path and vectorization of the min cut error calculation and cut matrix merge function. Created various helper function needed in order to adapt the quilting algorithm to use the non-interleaved image format. Adapted the quilting algorithm and the infrastructure to this new image format. Improved how the error calculation function was vectorized by switching to 16-bit integers when calculating the difference between pixels. Implementation of early stopping used in the optimizations 6b and 6c. Experimented with permuting color of the input images, which lead to the implementation of the automatic color permutation functionality used in optimization 10. Experimented with modifying the loop order in the error calculation function that concluded in optimization 6d.

Etienne. Made various improvements to the initial quilting implementation, experimented with various vectorization modifications. Set-up code for testing and timing. Ran the measurements and created the plots for the results. Analyzed code performance using VTune profiler.

Oleh. Worked mostly on the optimizations related to improving cache locality, particularly experimented with various versions of blocking for the cache for the initial algorithm and for the latter version. Made minor improvements to the vectorized code versions, especially to the operations ordering to get as much as possible benefit. Explored tools for conducting roofline analysis on my AMD processor and experimented with it. Analyzed the cost measure of our implementation. Counted all the operations in the algorithm and created the formula to compute the exact number of operations for given input parameters.

Ondrej. Implemented the first wave of optimizations to the baseline code, which included loop unrolling, smarter indexing, removal of unnecessary operations and data. Implemented the first error-calculation vectorization, which was later modified due to inefficient error summation of this version. Implemented the optimization, which included error prediction of minimum error at each iteration, which was supposed to speed-up early stopping, but did not see runtime improvement there. Also experimented with loop-unrolling for the early stopping optimizations, but stopped once the loop unrolling led to double the runtime, due to the unpredictable branch behavior in the early stopping. Analyzed the code runtime of the different optimization using Intel's VTune, which was also used to create the roofline analysis plot. Also took various photos used to test the code output throughout the project.

7. REFERENCES

[1] Alexei Efros and William Freeman, "Image quilting for texture synthesis and transfer," *Computer Graphics (Proc. SIGGRAPH'01)*, vol. 35, 07 2001.

[2] David J. Heeger and James R. Bergen, "Pyramid-based texture analysis/synthesis," in *Proceedings of the 22nd Annual Conference on Computer Graphics and Interactive Techniques*, New York, NY, USA, 1995, SIGGRAPH '95, p. 229–238, Association for Computing Machinery.

[3] Javier Portilla and Eero P. Simoncelli, "A parametric texture model based on joint statistics of complex wavelet coefficients," *Int. J. Comput. Vision*, vol. 40, no. 1, pp. 49–70, oct 2000.

[4] Jeremy S. De Bonet, "Multiresolution sampling procedure for analysis and synthesis of texture images," in *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, USA, 1997, SIGGRAPH '97, p. 361–368, ACM Press/Addison-Wesley Publishing Co.

[5] A.A. Efros and T.K. Leung, "Texture synthesis by non-parametric sampling," in *Proceedings of the Seventh IEEE International Conference on Computer Vision*, 1999, vol. 2, pp. 1033–1038 vol.2.

[6] Khaled Alomar, Halil Aysel, and Xiaohao Cai, "Data augmentation in classification and segmentation: A survey and new strategies," *Journal of Imaging*, vol. 9, pp. 46, 02 2023.

[7] Alice Wang and Jordan Mecom, "Image quilting for texture synthesis and transfer," www.jmecom.github.io/projects/computational-photography/texture-synthesis/.

[8] Kashif Mahmud, Gregoire Mariethoz, Jef Caers, Pejman Tahmasebi, and Andy Baker, "Simulation of earth textures by conditional image quilting," *Water Resources Research*, vol. 50, 04 2014.

[9] Júlio Hoffimann, Céline Scheidt, Adrian Barfod, and Jef Caers, "Stochastic simulation by image quilting of process-based geological models," *Computers & Geosciences*, vol. 106, pp. 18–32, May 2017.

[10] Xiaofeng Tao, "Image quilting for texture synthesis and transfer," www.cs.brown.edu/courses/cs129/results/proj4/taox/.

[11] Yuanyuan Pu, Dan Xu, Wenhua Qian, Yaqun Huang, and Youyan Dan, "An improved texture synthesis algorithm," in *Transactions on Edutainment XI*, Zhigeng Pan, Adrian David Cheok, Wolfgang Mueller,

and Mingmin Zhang, Eds., Berlin, Heidelberg, 2015,
pp. 103–113, Springer Berlin Heidelberg.