HIGH-PERFORMANCE IMPLEMENTATION STRATEGIES FOR TEXTURE SYNTHESIS

Tal Rastopchin *Svitlana Morkva* *Ivan Sobko* *Baptiste Goumain*

Department of Computer Science
ETH Zurich, Switzerland

ABSTRACT

Image Quilting is a widely used technique for texture synthesis. This paper focuses on the optimization of Image Quilting algorithms without thread parallelization. Through a comprehensive study, we explore various optimization strategies such as straightforward algorithm improvement, unrolling, blocking and vectorization to reduce computational overhead and enhance the overall performance. Experimental results demonstrate that the proposed optimization strategies effectively reduce the execution time of Image Quilting algorithms. Furthermore, we analyze the impact of different optimization techniques on memory consumption, performance, and perceptual quality to provide a holistic understanding of their benefits and limitations.

1. INTRODUCTION

Image Quilting algorithms have become indispensable tools in various image processing applications, providing solutions for texture synthesis, seamless image stitching, and image inpainting. However, with the ever-increasing demand for real-time performance and the handling of large-scale textures, there is a growing need for high-performance versions of Image Quilting algorithms. The objective of this paper is to introduce and evaluate a high-performance version of the Image Quilting algorithm that addresses the computational challenges and improves overall efficiency.

Motivation. The motivation behind this research arises from the limitations of traditional Image Quilting algorithms in meeting the demands of modern applications. Standard approaches often struggle to handle high-resolution images and it poses a challenge in terms of memory consumption and processing time. Therefore, a high-performance version of the Image Quilting algorithm is crucial to overcome these limitations and unlock its full potential in various domains.

Contribution. In this study, we present an optimized implementation of the Image Quilting algorithm, focusing on key areas such as enhancements to a straightforward algorithm, loop unrolling, block processing and SIMD vectorization. We explore innovative approaches that leverage modern hardware architectures, including Central Processing Units (CPUs), to accelerate the algorithm's execution time.

2. BACKGROUND ON IMAGE QUILTING

Within this section, our objective is to present a formal definition of Image Quilting, introduce the essential components of the algorithm alongside our implementation, conduct a cost analysis, and outline the asymptotic complexity of the problem.

Image Quilting. Texture synthesis refers to the process of generating new, realistic textures based on a given input texture or a set of exemplar textures. The goal is to create a larger texture that captures the visual characteristics and patterns present in the input textures. Image Quilting is a common approach of patch-based synthesis, where small patches of the input texture are used to iteratively reconstruct the larger texture. These patches are selected based on their similarity to the surrounding context, ensuring that they fit seamlessly into the overall texture. The algorithm consists of three key components that progressively enhance both the algorithm itself and the resulting output texture.

We can define B_i as a synthesis unit, which is a square block of user-specified size extracted from the set S_B containing all overlapping blocks in the input texture image. The most straightforward approach to generating a larger texture involves randomly sampling blocks from S_B. Although this method produces textures of the desired size, they exhibit noticeable flaws due to the lack of correspondence between selected blocks, resulting in clearly visible seams.

To address this imperfection, a solution is to randomly sample only the top-left corner block. For all other positions, we would search for a block in S_B that agrees with its neighboring blocks in the output texture. The agreement between the chosen B_i from S_B and its neighboring blocks is typically evaluated using the ℓ^2-norm function on the overlap regions. In the Image Quilting algorithm, the overlap region is commonly defined as one-sixth of the block size, with the blocks stitched in the middle of this overlap region. The error tolerance was set to be within 0.1 times the error

of the best matching block. The stitching process can be categorized into three separate cases: vertical stitching for the first row, horizontal stitching for the first column, and corner stitching for all other blocks.

While the previous approach has significantly improved the output, there are still instances where the seams remain noticeable. This is because the blocks are stitched together using straight lines, which can be easily detected by the human eye. Therefore, in the next component of the algorithm, we aim to further reduce this problem by identifying a minimum cost path through the error surface within the overlap region. To perform the Minimum Error Boundary Cut using dynamic programming, we define the error surface as $e = (B_1^{ov} - B_2^{ov})^2$, where B_1^{ov} and B_2^{ov} represent the overlap regions of two blocks. The goal is to compute the cumulative minimum error, E, for all possible paths as:

$$E = e_{i,j} + min(E_{i-1,j-1}, E_{i-1,j}, E_{i-1,j+1}).$$

The minimum value in the last row of the table represents the end of the best stitching path. To obtain the whole path, we traverse back from this minimum value, selecting the cells with the minimum cumulative error at each step until we reach the first row.

Cost Analysis. Our definition of the cost measure is informed by our initial profiling of our baseline implementation. In particular, generating a flame graph revealed that about 85% of the runtime of the algorithm was devoted to computing the ℓ^2-norm between block overlap regions. About only 3% of the runtime was devoted to computing the minimum cut and writing the block. Because the computation of the ℓ^2-norm dominates the runtime of the algorithm, we decided to define our cost measure in terms of the operations required to compute the ℓ^2-norm.

Because the input to our algorithm is an image, which are most commonly encoded as two-dimensional arrays of packed 1-byte wide red, green, blue, and alpha channels (RGBA), we decided to focus our implementation and cost measure on integer operations. CPUs are designed to have many more integer ALUs than Floating Point execution units because integer arithmetic is much more common than floating point arithmetic. Furthermore, using the narrowest numeric type appropriate would allow higher performance when applying applying SIMD intrinsics.

To define our cost measure let's derive the number of operations needed to compute the ℓ^2-norm between two pixels. Let's assume that each pixel $\boldsymbol{p}_i$ has 4 channels, namely the RGBA channels. We then have that

$$|\boldsymbol{p}_i - \boldsymbol{p}_j|^2 = (r_i - r_j)^2 + (g_i - g_j)^2 + (b_i - b_j)^2 + (a_i - a_j)^2.$$

We would lastly take the square root, but because we are interested in the minimum among all ℓ^2-norms, we can find the minimum squared ℓ^2-norm and get rid of the square root entirely. For n pixels we can then define the following integer cost measure:

$$C(n) = C_{\text{int add / sub}} \cdot N_{\text{int add / sub}} + C_{\text{int mul}} \cdot N_{\text{int mul}}.$$

We have $4n$ subtractions, $3n$ additions, $4n$ multiplications, and $n - 1$ additions from adding the resulting squared ℓ^2-norms together. This means our cost measure is

$$C(n) = C_{\text{int add / sub}} \cdot (8n - 1) + C_{\text{int mul}} \cdot 4n.$$

It is important to note that addition and subtraction usually have smaller latency and gap than multiplication. According to Agner Fog's Instruction tables, we have that for the Intel Skylake microarchitecture, the Add, Sub, and Mul integer instructions have the following latencies and throughputs:

	Latency	Throughput
Add, Sub	1 cycle	4 per cycle
Mul	3-4 cycles	1 per cycle

This means that the cost of an integer multiplication is greater than the cost of an integer addition or subtraction.

An asymptotic analysis of the Image Quilting algorithm further justifies our choice of cost measure. The original Image Quilting paper [1] does not perform an asymptotic analysis and so we provide a brief one here. For the purpose of the asymptotic analysis, assume that the input image is square and has $n \in \mathbb{N}$ pixels. Then, its width and height are $\sqrt{n}$ pixels. Further assume that the output image is also square and linearly proportional to the dimensions of the input image. Then, the output image has a width and height of $O(\sqrt{n})$ pixels. Further assume that the blocks are square and their dimensions are linearly proportional to the dimensions of the input image. Then, the block size has a width and height of $O(\sqrt{n})$ pixels.

Because the output image dimensions are linearly proportional the dimensions of the input image, the Image Quilting algorithm will select some constant number of blocks. Furthermore, we also have that the size of the set of potential source blocks is linear in the number of input pixels; that is, $|S_b| \in O(n)$. Selecting a block consists of iterating over every potential source block in S_b, computing its ℓ^2-norm with respect to the overlap region in the output texture, randomly sampling a block within a threshold of the lowest ℓ^2-norm, and computing the minimum cut to write the new block.

The overlap region between a potential source block and the output texture will always consist of one or two rectangular sub-regions. The original Image Quilting paper chooses the overlap width to be one sixth of the block width; we chose the same value in our implementation. Hence, the dimensions of the rectangular sub-regions are linear in the

dimensions of the block width, and so the number of pixels of the overlap region will be linear in the number of input pixels, $O(n)$. Iterating over every potential source block in S_b and computing its ℓ^2-norm with respect to the overlap region in the output texture then contributes $O(n^2)$ to the algorithm complexity.

Randomly sampling a block within a threshold of the lowest ℓ^2-norm can be achieved with a linear complexity by iterating over the n ℓ^2-norms to find the minimum, contributing $O(n)$ to the algorithm complexity.

Performing the minimum cut requires computing the error surface, filling up a dynamic programming table, and traversing the dynamic programming table to retrieve the cut. Each of these steps in the worst case iterates over every pixel of an overlap region. So computing the minimum cut and writing the block together is linear in the number of input pixels, , contributing $O(n)$ to the algorithm complexity.

Putting it all together, we conclude that the Image Quilting algorithm is quadratic in the number of input pixels n because; we say that Image Quilting $\in O(n^2)$.

3. OPTIMIZATION STRATEGIES

This section will delve into the specific optimization strategies implemented during the study, providing a comprehensive overview of the methods used and their corresponding rationale. The results and impact of each optimization strategy will be discussed in the next section, showcasing the effectiveness and benefits they brought to performance improvement.

Optimization plan. To formulate our optimization plan, we initially classified the potential optimizations for our algorithm into two categories: basic and advanced. The basic optimizations are general techniques that are applicable to a wide range of algorithms, usually yielding light but consistent improvements in speed. These optimizations may lead to a marginal reduction in the number of operations. Additionally, these optimizations typically simplify the code, which can aid the compiler in optimizing the algorithm more effectively.

Advanced optimizations require significant analysis of the algorithm. Upon careful examination, we have identified several potential optimizations in this block:

1. Applying loop unrolling with scalar replacement and optimizing dependencies.

2. Refactoring the functions into separate case-specific implementations for different overlap scenarios, including variations with and without boundary checks.

3. Implementing a simple optimization for spatial locality without introducing blocking.

4. Introducing blocking in the ℓ^2-norm function to enable data reuse and load data into the cache only once.

5. Performing manual vectorization of computations.

Some of these optimizations will be further explored in this section.

Basic optimizations. This optimization block involves introducing precomputations to improve temporal locality, inlining function calls, replacing computationally expensive operations with more optimal alternatives, rewriting the code in a C-style fashion, and replacing library functions with our own implementations to eliminate unnecessary computations.

Our initial implementation of texture synthesis required numerous index computations to access specific parts of the output texture and the blocks from the set S_B to determine overlap regions. To optimize this process, we precomputed various values, such as row and column shifts to align with the beginning of overlap regions. Moreover, we stored pointers to the start of these rows for both the output texture and the blocks, enabling us to easily increment the pointers while iterating over the columns to efficiently calculate the error between the overlapping blocks. By implementing this modification, we removed a total of 20 integer additions per pixel. However, the impact was even more substantial due to the removal of 8 multiplications and the elimination of various operational dependencies. These changes are particularly significant as they mitigate high latency and enhance performance through improved parallelization.

Algorithmic Optimizations. One of our algorithmic optimizations was to reduce the number of square root operations, as we know that they are very costly. In the original paper [1], the squared root of sum differences for each block is computed. The minimum value among these values then determines the best matching block, and a set of blocks within 0.1 times the error of this block is considered suitable for pasting into the desired output texture.

However, we can simplify the algorithm since we do not need squared root actual values. As the square root function is monotonically increasing, finding the minimum sum of squared differences suffices to identify the best matching block. To ensure consistency in the set of suitable blocks, we calculate the square root solely for the minimum value. To ensure that the set of suitable blocks remains the same, we can obtain the square root only for the minimum value, multiply it by the error tolerance to determine the threshold, and then square it again. With this approach the resulting value is proportional to the calculated sum of squared differences for all other blocks.

Another of our algorithmic optimizations was to merge the loops covering the horizontal and corner overlap cases for the ℓ^2-norm computation. Not only did this result in a simplified algorithm, but by eliminating the need to process

rows of the overlap region separately in distinct loops, data access became more contiguous and sequential, improving spatial locality.

Loop Unrolling. Another optimization that we implemented is loop unrolling with separate accumulators. The main objective of loop unrolling is to enhance Instruction Level Parallelism. To achieve maximum operation parallelization, we introduced multiple accumulators to ensure independence among operations and improve the computational efficiency of the algorithm. It is crucial to emphasize that one of the main challenges in this optimization lies in determining the appropriate unroll factor and the number of accumulators to use. In accordance with the cost analysis detailed in the second section, it was discussed that the ℓ^2-norm estimation function is responsible for approximately 85% of the total runtime. Consequently, we prioritized the majority of our analysis and optimization efforts on this specific aspect of the algorithm. Moving forward, our analysis of loop unrolling will be conducted based on this function.

In the ℓ^2-norm estimation function, each pixel in the overlap region involves the following integer operations: 4 independent subtractions, 4 multiplications, and 4 final additions per iteration. When considering the Intel Skylake microarchitecture, it is possible to execute additions (subtractions are considered to be additions as well) on a maximum of 4 ports, while multiplications can only be performed on 1 of these 4 ports. As a result, the computation is primarily bottlenecked by the multiplication operation.

Assuming no dependencies are taken into account and we do not wait for any computations to finish, for these set of the operations the optimal runtime per iteration can be calculated as follows:

$$max(4\ \{1 \text{ mults/cycle}\}, \frac{8}{3}\ \{3 \text{ adds/cycle}\}) = 4 \text{ cycle/pixel}.$$

Now that we have determined the most optimal theoretical runtime for the given machine, let's proceed with calculating the runtime of the ℓ^2-loss between two pixels, taking into account the dependencies present in the code. It is assumed that we have already unrolled the innermost loop, which iterates through the channels. As you can see in the dependency graph shown in Fig. 1, due to the presence of dependencies we require 2.5 times more cycles per iteration. The main problems arise from the delay caused by multiplication, where the machine has to wait for instructions to finish, and from not fully utilizing the machine's integer addition ports to their maximum capacity.

To optimize the algorithm, let us determine the ideal unrolling factor that can effectively address these limitations. Within the algorithm, the number of additions outweighs the number of multiplications by a factor of two. Furthermore, the throughput of additions surpasses that of multiplications by a factor of four. Consequently, under optimal circumstances, it becomes feasible to parallelize slightly over half

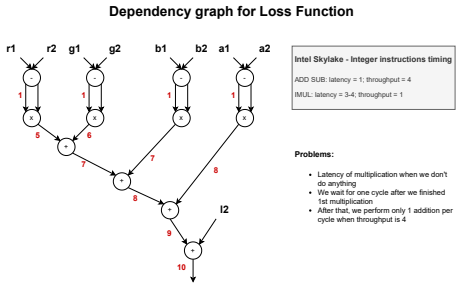

Fig. 1. Dependency graph of the ℓ^2-loss function between two pixels, considering Intel Skylake microarchitecture. Channels $r_1, r_2, g_1, g_2, ...$ correspond to the respective pixel channels.

of all multiplications. Considering this, an unroll factor of 4 should suffice since it allows for parallelization of half of the multiplications. Specifically, for the first set of pixel pairs, the multiplication of differences can be executed in parallel with the subtractions of RGBA channel values from the subsequent 3 pairs. Similarly, for the final pixel pair, the multiplication can occur concurrently with the final additions. This assertion can be substantiated by examining the dependency graph provided in Fig. 2. By employing this unrolling technique, we successfully minimized the number of cycles required per pixel to a mere 5.25. This value is only slightly larger than the best possible scenario for this algorithm. It serves as a favorable compromise since higher unrolling factors lead to increased code complexity and potential memory-related challenges arising from a larger number of accumulators.

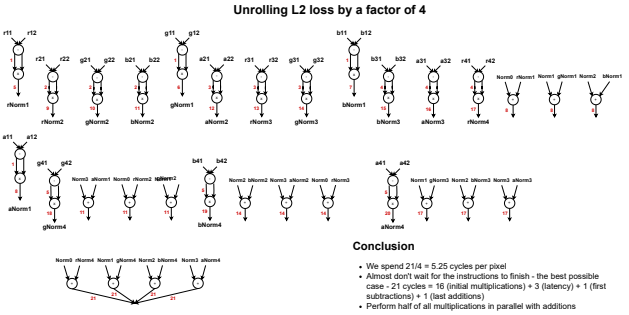

Fig. 2. Dependency graph of the ℓ^2-loss function between two pixels with an unroll factor of 4, considering Intel Skylake microarchitecture.

Locality and Blocking.

Understanding the temporal and spatial locality of memory access patterns is a crucial step in improving performance and efficiency by reducing the number of cache misses outside of the last level cache (LLC). We perform a temporal and spatial locality analysis, informing our optimization

plan of (1) selecting a new block and (2) performing the minimum cut.

(1) Selecting a Block.

Recall that selecting a block consists of iterating over every potential source block in S_b, computing its ℓ^2-norm with respect to the overlap region in the output texture, and randomly sampling a block within a threshold of the lowest ℓ^2-norm. In our baseline implementation, we created a function called `ComputeOverlap` whose sole purpose was to compute the ℓ^2-norm between the overlap region of the output texture and of an individual source block. A single call to this function exhibits no temporal locality because each pixel in the output and input overlap regions is accessed exactly once. However, because our images are stored as two-dimensional arrays, this function exhibits spatial locality, since pixels of the same row are stored contiguously in memory. In the baseline implementation we already took advantage of the spatial locality since we iterate through the overlap region row-by-row. Therefore, because the CPU reads contiguous cache blocks into the L1 and L2 caches, for a cache block size of B we will have on average 1 compulsory cache miss to $B-1$ cache hits.

We implemented selecting a block by calling the `ComputeOverlap` function once for each potential source block in S_b. Although this approach results in readable code, it leads to an enourmous amount of temporal locality. Firstly, each call to the `ComputeOverlap` function iterates over every pixel of the the overlap region in the output texture, and so the overlap region in the output texture is read $|S_b|$ times. Secondly, although each call to the `ComputeOverlap` function iterates over the overlap region of a different source block, consecutive source blocks often differ by a one-pixel-shift to the right, and hence their overlap regions admit a considerable amount of the same pixels. Non-consecutive source blocks that differ by horizontal or vertical shifts similarly admit overlap regions that share a substantial number of pixels.

We can address the temporal locality of iterating over the overlap region in the output texture $|S_b|$ times with a loop reordering. The baseline implementation calls the `ComputeOverlap` function once for each potential source block in S_b, which gives us the following pseudo code:

```
1  for i = 0 to maxBlockY:
2    for j = 0 to maxBlockX:
3      for k = 0 to overlapHeight:
4        for l = 0 to overlapWidth:
5          op = output(k, l)
6          ip = input(i, j, k, l)
7          l2(i, j) += |op - ip|^2
```

Here $|S_b| = \text{maxBlockY} * \text{maxBlockX}$; hence the outer two loops iterate over every potential source block and the inner two loops iterate over the pixels of the overlap region. Fixing `k` and `l`, the output pixel `op = output(k, l)` is read $|S_b|$ times, and the input pixel `ip = input(i, j, k, l)` iterates over almost every pixel of the input texture. Letting `k` and `l` vary we have that the input pixel `ip = input(i, j, k, l)` iterates over almost every pixel of the input texture multiple times.

The above pseudocode analysis tells us that (1) the output overlap region is read multiple times and (2) the input texture is read multiple times. Ideally, both the output overlap region and input texture need to fit into cache. However, by reordering the inner two and outer two, we can iterate over the pixels of the output overlap region exactly once:

```
1  for k = 0 to overlapHeight:
2    for l = 0 to overlapWidth:
3      op = output(k, l)
4      for i = 0 to maxBlockY:
5        for j = 0 to maxBlockX:
6          ip = input(i, j, k, l)
7          l2(i, j) += |op - ip|^2
```

After this loop reordering, the algorithm (1) reads the output overlap region exactly once and (2) still reads the input texture multiple times. It is important to note that performing this loop reordering requires us to now keep track of an array of the $|S_b|$ ℓ^2-norms, since we no longer compute the norms one at a time; this is an array of integers and recall the size of an integer is 4 bytes. This means that ideally only a single cache block of the output overlap region, the entirety of the input texture, and the entire array of $|S_b|$ ℓ^2-norms should fit into cache. The size of the array of $|S_b|$ ℓ^2-norms is the same of the size of the input texture since the ℓ^2-norms are stored with integers and each pixel of the input texture uses 4 bytes. It is unlikely that the input texture and array of ℓ^2-norms will fit into the L1 cache; with 4 bytes-per-pixel, the 64 KB L1 cache of the Intel Skylake microarchitecture can only fit a square input texture with width just under 2^7 pixels.

Hence, the next logical step is to block the inner two loops so that a smaller subset of the input texture and array of ℓ^2-norms remains in the L1 cache. We perform a blocking that aims to keep one row of the output overlap region and a rectangular sub-region of the input texture in cache. We compromised to keep an entire row of the output overlap region as opposed to a single cache block since keeping a row of the overlap region's pixels in cache is feasible and it allowed us to focus on blocking the inner two loops. Let B be our block size; we then block the our pseudocode as follows:

```
1  for k = 0 to overlapHeight:
2    for m = 0 to maxBlockY by B:
3      for n = 0 to maxBlockX by B:
4        for l = 0 to overlapWidth:
5          op = output(k, l)
6          for i = m to m + B:
7            for j = n to n + B:
```

```
8               ip = input(i, j, k, l)
9               l2(i, j) += |op - ip|^2
```

The algorithm block width is at most the input texture width, and so only a single row of the input texture, a B by B block of the input texture, and a B by B subset of the corresponding array of ℓ^2-norms should fit into cache. A single row of the input texture will be considerably smaller than the $4B^2 + 4B^2$ bytes required for the block of the input texture and the subset of the corresponding array of ℓ^2-norms. A block size of $B = 90$ just barely fits the $4B^2 + 4B^2$ bytes into the 64 KB L1 cache of the Intel Skylake microarchitecture. This suggests experimenting with block sizes greater than zero to a bit above $B = 90$.

(2) Performing the Minimum Cut.

While we did not spend much time optimizing performing the minimum cut, we still provide a brief locality analysis.

In the case of the vertical minimum cut, we have spatial locality since we are iterating over pixels of the overlap region; the baseline implementation already takes advantage of the spatial locality since we iterate through the overlap region row-by-row. We also have a small amount of temporal locality since we construct the vertical path by traversing the dynamic programming table from the last row to the first row.

In the case of the horizontal minimum cut, we have spatial since we are iterating over pixels of the overlap region; however, the baseline implementation does not take advantage of the spatial locality since we iterate through the overlap region column-by-column. We also have a small amount of temporal locality since we construct the horizontal path by traversing the dynamic programming table from the last column to the first column.

Vectorization. The majority of our vectorization efforts are dedicated to computing the ℓ^2-loss function, as it is the most computationally expensive operation in our code. By optimizing this function, we can achieve significant performance improvements.

When utilizing SIMD (Single Instruction, Multiple Data) vectorization, there are several considerations to keep in mind to maximize performance and efficiency:

Data alignment: SIMD operations typically require data to be aligned on specific boundaries. We ensure that data is aligned by using `malloc`, which is guaranteed to be properly aligned for objects of any type. Maximum alignment size of `malloc` is determined by `max_align_t` which usually equals to size of `long double` (16 bytes). Maximum alignment will influence vector width selection.

Data dependency: Minimizing data dependencies within vectorized code is crucial for enabling parallel execution and fully utilizing SIMD capabilities. To achieve this, we apply insights discussed in the loop unrolling section, reducing idle CPU time.

Vector width: In our case, we represent each pixel value as a 16-bit value with four components, each occupying one byte. To perform operations correctly, we load 16 8-bit values into a 128-bit integer variable and convert it into a 256-bit variable of 16-bit integers. This approach prevents overflow during multiplication and addition and maximizes performance by utilizing the available memory in the vector.

Through our experiments, we found that the "madd" (multiply-add) function is well-suited for our specific purpose of calculating the norm and performing horizontal component addition. Furthermore, to mitigate data dependencies, we employed a loop unroll with a factor of x2 of the vector width. This approach allowed us to effectively reorder operations and minimize periods of idle CPU time.

4. EXPERIMENTAL RESULTS

This section presents the methodology employed in the experiments, including the hardware and software. Additionally, it delves into the specific results obtained from the experiments, highlighting any significant observations or trends. The interpretation and analysis of these results will contribute to a deeper understanding of the results of our work.

Experimental setup. We employed various platforms and compiler flags to assess the outcomes of diverse configurations. To ensure consistency and comparability across different platforms, we opted for the GCC 12.2.0 compiler, with three sets of flags: `-O3-ffast-math-march=native`, `-O3-fno-tree-vectorize` and `-O1`. We used three different platforms to perform the experiments.

(1) An AMD Ryzen 7 5800H @3.2GHz processor; L3: 16 MB. For most of the experiments.

(2) An Intel(R) Core i7-10510U @1.8GHz processor; L3: 8 MB. To compare to an benchmark implementation.

(3) An Intel(R) Core i7-1068NG7 @2.3GHz; L1: 64 KB, L2: 256 KB, L3: 8 MB. For different block sizes.

For input we used square images of varying sizes, ranging from 2^{11} to 2^{18} in terms of pixel count. We always use a square Image Quilting block with width 1/4 of the input width, unless specified otherwise.

Results.

Basic optimizations and algorithmic improvements: In Fig. 3, we present a performance comparison of the basic optimizations and algorithmic improvements with respect to our baseline implementation for different input sizes. The program was compiled using the flags `-O3 -fno-tree-vectorize`. We will address a significant decline in performance observed across all implementations for larger image sizes in the further sections. Nevertheless, it is evident that we achieved a substantial increase in program performance. The specified machine has a peak performance of 4 ops/cycle for integer operations, and our basic optimiza-

tions, combined with algorithmic improvements, achieved over 3.5 ops/cycle for several input sizes. However, the achieved improvement is not solely attributable to the specified techniques. Rather, it stems from the successful simplification of the code and the resolution of certain dependencies that improved the compiler's ability to analyze and enhance performance through advanced optimizations that were applied automatically.

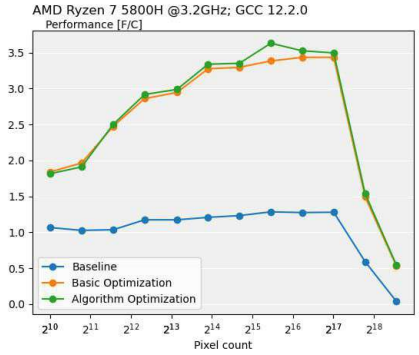

Fig. 3. Performance of the basic optimizations, algorithmic improvements, and baseline implementation.

Benchmark: How does the performance of our standard C and algorithm optimizations compare to the performance of a benchmark implementation across the three sets of compiler flags? Fig. 4 demonstrates that across the three sets of compiler flags, our standard C and algorithm optimizations consistently outperforms the benchmark implementation.

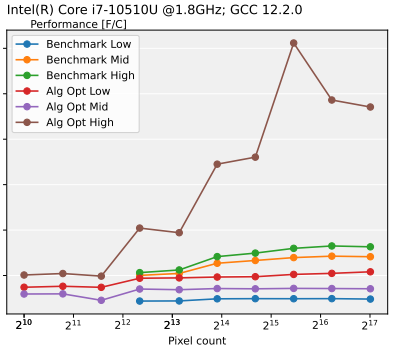

Fig. 4. Performance plot of the standard C and algorithm optimizations compared to a benchmark implementation.

Loop Unrolling: In Fig. 5, we compared different unrolling factors. While a theoretical analysis suggested that unrolling by a factor of 4 would provide the best performance, in practice, unrolling by a factor of 2 performed better in most cases. This could be due to increased memory traffic complications caused by a higher number of accumulators or the complexities introduced by larger unrolling factors, making it challenging for compiler optimizations.

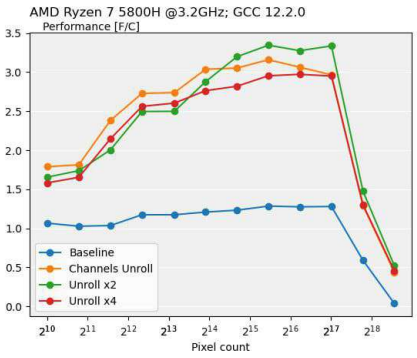

Fig. 5. Performance of different unrolling factors comparing to the baseline implementation.

Despite the decrease in runtime achieved through our unrolling technique, its performance is slightly inferior to that of the previous improvements. This observation, combined with the significant performance drop for larger input sizes, led us to recognize that our program is memory-bound. To substantiate this, we constructed a roofline plot, depicted in Fig. 6. As we optimize our program's computations further, we observe a decrease in operational intensity and an increasing dependency on memory access, indicating a more memory-bound nature.

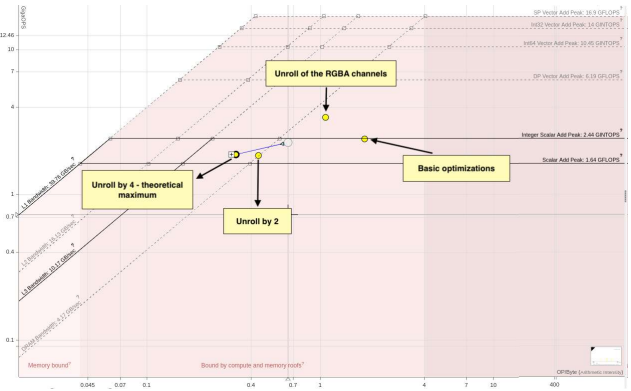

Fig. 6. Roofline plot for different unrolling factors

Blocking: Does blocking help us become less memory bound and improve our performance? Fig. 7 illustrates that blocking does in fact increase the operational intensity of the loop unrolling by 2 and loop unrolling by 4 optimization variants.

Vectorization: In Fig. 8 we show our vectorization results, as anticipated, this approach yielded superior results compared to simple unrolling. However, we observed that

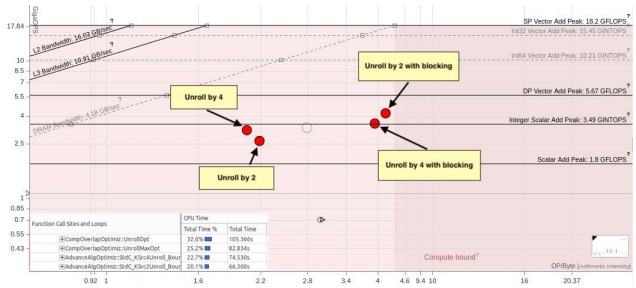

Fig. 7. Roofline plot illustrating the result of blocking the loop unrolling by 2 and 4 optimization variants.

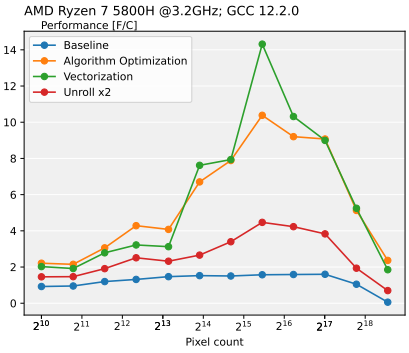

Fig. 8. Performance of vectorization optimization compared to unrolling and algorithm optimization

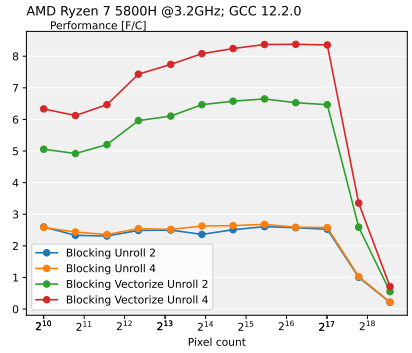

Fig. 9. Performance of blocking vectorization compared to regular blocking.

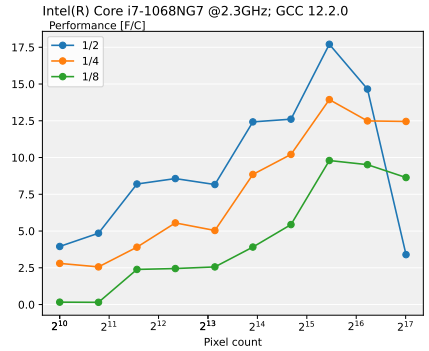

Fig. 10. Performance of the standard C and algorithm optmizations with Image Quilting block sizes of 1/2, 1/4, and 1/8.

algorithm optimization performed comparably to manual vectorization. Compiler exhibited an ability to optimize the code as efficiently as our manual efforts, and in some cases, even surpassing our optimizations.

Transitioning to blocking vectorization, the notable improvement in blocking can be observed in Fig. 9. This improvement can be primarily attributed to the fact that the compiler encounters more difficulties in optimizing non-straightforward algorithms such as blocking.

Block Size: How does the Image Quilting algorithm block size affect the standard C and algorithm optimizations implementation Performance? Fig. 10 illustrates that the performance increases with a larger Image Quilting block size. This makes sense since a larger Image Quilting block sizes leads to larger contiguous regions of memory for the algorithm to access. Note that the performance of the 1/2 block size plummets on the largest input size, potentially because the blocks are no longer fitting into LLC.

5. CONCLUSIONS

Our results illustrate that a combination of standard C optimizations, algorithm improvements, vectorization, and blocking, are all needed to produce a highly performant and efficient implementation of the Image Quilting algorithm. An important result is that applying standard C optimizations and algorithm improvements along with using the highest level of compiler optimization flags produced some of our most performant and efficient optimizations. Another key result is that blocking was mostly beneficial when paired with vectorization; otherwise, the performance and runtime gains were not significant.

6. CONTRIBUTIONS OF TEAM MEMBERS

Svitlana. Focused on the set of basic optimizations and the algorithmic improvement that improved spatial locality (for ℓ^2-norm function) that were discussed in Section 3 in the Basic optimizations block. Implemented unrolling with different unrolling factors for 2 cases of the algorithm - before and after the blocking and performed the corresponding analysis. Implemented loop reordering to reuse the data in the output texture and worked on the analysis of the spatial locality and blocking with Tal. Divided functions into case-specific types to reduce unnecessary computations -

second point of the advanced optimization plan. Prepared the roofline plots.

Ivan. Implemented SIMD vectorization optimizations, analyzed the influence of unroll factor on vectorization performance. Worked on initial cost analysis, INTOPs and FLOPs calculation and identifying bottlenecks. Created performance and runtime plots.

Tal. Focused on the blocking optimizations. Implemented basic optimizations and loop unrolling for performing the minimum cut with Baptiste. Implemented loop reordering to reuse the data in the output texture and worked on the analysis of the spatial locality and blocking with Svitlana. Implemented a first unsuccessful blocking attempt; implemented a second successful loop blocking attempt with Svitlana. Used Ivan's code to create performance and runtime plots.

Baptiste. Implemented basic optimizations and loop unrolling for performing the minimum cut with Tal. Did benchmarking analysis and benchmarking plots.

7. REFERENCES

[1] Alexei A. Efros and William T. Freeman, "Image quilting for texture synthesis and transfer," in *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, New York, NY, USA, 2001, SIGGRAPH '01, p. 341–346, Association for Computing Machinery.