SINGLE CORE OPTIMIZATION OF IMAGE QUILTING FOR TEXTURE SYNTHESIS

Christof Leutenegger, Lukas Nüesch, Marco Heiniger, Piero Neri

Department of Computer Science
ETH Zurich, Switzerland

ABSTRACT

This paper introduces an optimized version of the image quilting algorithm for texture synthesis, employing the precomputation technique of integral images and leveraging SIMD instructions. The runtime and performance improvements of the various optimization steps are evaluated through three comprehensive benchmark tests, encompassing all variable input parameters of the algorithm. Moreover, an analysis is conducted using roofline plots to visualize the algorithm's efficiency and resource utilization. In the best case, our final optimization achieves a 45x improvement in runtime compared to a straightforward base implementation.

1. INTRODUCTION

Texture synthesis is the process of generating larger textures by seamlessly combining smaller texture elements. Image quilting is a widely used technique for texture synthesis, whereby blocks from the input image are stitched together to create visually appealing and realistic textures [1].

Motivation. In computer graphics, virtual terrain generation is a significant topic with numerous applications [2]. A rapid image quilting algorithm for generating terrains, particularly in gaming applications, significantly enhances the user experience by enabling instant feedback and ensuring a seamless interaction environment. Additionally, image quilting can be used as a valuable technique for data augmentation in machine learning [3]. A faster image quilting algorithm accelerates augmented dataset generation, enabling quicker iterations, increased productivity, and efficient exploration of data variations, benefiting the machine learning workflow.

Contribution. In this paper we present an optimized implementation of the image quilting algorithm tailored for single-core processing.

Related work. In their seminal work, Efros et al. introduced the concept of image quilting as a technique for texture synthesis [1]. However, their approach did not consider the utilization of SIMD (Single Instruction, Multiple Data) instructions for performance optimization. Related subjects, such as template matching, often employ the concept of integral images as a form of precomputation to reduce the number of instructions and improve runtime efficiency. This technique has been shown to be effective in various computer vision tasks, including object detection and recognition. One reference that discusses the use of integral images in template matching is the work by Viola and Jones [4]. Utilizing SIMD instructions and integral images, we present an optimized implementation of the image quilting algorithm.

2. BACKGROUND ON THE ALGORITHM

This section presents the background for the image quilting algorithm and integral images that are used as an optimization.

Image quilting for texture synthesis. Image quilting is an algorithm that aims to generate larger textures by stitching together smaller overlapping blocks from a given input texture, creating seamless and visually appealing textures that closely resemble the input texture. The algorithm described by Efros et al. [1] is depicted in Algorithm 1. Blocks are inserted in row order overlapping by a fix amount.

Let's define M as the block side length. Then every block B_i within the input image of length $M \times M$ is considered a possible block to stitch to the previous blocks. To compare and calculate the error of two overlapping blocks B_i and B_j, we calculate the sum of squared differences (SSD) of their overlapping region B_i^{ov}, B_j^{ov}. Let Q and R denote width and height of the overlapping region.

$$SSD = \underbrace{\sum_{q=1}^{Q}\sum_{r=1}^{R}(B_i^{ov}[q,r] - B_j^{ov}[q,r])^2}_{\substack{\text{For each [q, r]: } 1_{\mathrm{sub}} + 1_{\mathrm{mul}} + 1_{\mathrm{add}} \\ \text{in total: Q * R * 3 ops}}} \qquad (1)$$

The minimum error boundary cut that stitches the blocks together is then performed using dynamic programming. It involves filling a cumulative minimum error matrix using squared differences and backtracking to determine the best path to get a seamless texture.

Integral images. Integral images are a computational technique that efficiently calculates the sum of pixel inten-

Algorithm 1: Image quilting

```
1  for every row of blocks do
2      for every column of blocks do
3          if first block in output image then
4              choose a random block from input image
5          else if in leftmost column then
6              for each possible block in input image do
7                  calculate top overlap error
8          else if in topmost row then
9              for each possible block in input image do
10                 calculate left overlap error
11         else
12             for each possible block in input image do
13                 calculate top and left overlap errors
14         pick random block out of the best candidates
15         perform minimum error boundary cut
16         copy the block into the output image
```

sities within rectangular regions of an image. Integral images are constructed by calculating the cumulative sum of pixel intensities over rows and columns of an image. Using integral images provides a computational benefit by allowing the sum of pixel intensities within rectangular regions to be computed using only $four$ operations, regardless of the region's size [4].

Cost analysis. For the cost measure we count the number of additions, multiplications and comparisons. As the unit of measurement changes we either use flops (floating point operations) or intops (integer operations) to count the number of operations.

We define the parameters of the cost measure for image quilting as follows:

$$\begin{aligned} n &= \text{number of blocks in a row/column} \\ h &= \text{input image height} \\ w &= \text{input image width} \\ ov &= \text{overlap size} \\ b &= \text{side length of a block} \end{aligned}$$

With this we can define the following formula for the cost measure:

$$C(n, h, w, ov, b) = (C_{add}, C_{mul}, C_{cmp})(n, h, w, ov, b)$$

Filled in with the actual cost of the base implementation we get the following calculation:

$$\begin{aligned} &C_{base}(n, h, w, ov, b) \\ &\leq C_{add} \cdot n^2 \cdot (12 \cdot h \cdot w \cdot ov \cdot b + 14 \cdot ov \cdot b + ov^2 \cdot 6) \\ &+ C_{mult} \cdot n^2 \cdot (6 \cdot h \cdot w \cdot ov \cdot b + 6 \cdot ov \cdot b + ov^2 \cdot 3) \\ &+ C_{cmp} \cdot n^2 \cdot (10 \cdot ov \cdot b + 2 \cdot ov + 4 \cdot b + 2 \cdot h \cdot w) \end{aligned}$$

For the performance measurements we aggregate the operation counts into one number and weigh adds, multiplications and comparisons equally. Although the operations count might change depending on the optimization step, the asymptotic complexity stays the same.

Liang et al. [5] discuss in a follow-up paper to Efros et al. [1], that they optimize the block search, which dominates the number of operations, to achieve better performance. The algorithmic complexity of the block search is dominated by the SSD calculations. For all n^2 blocks in the output image, we must compare $O(h \cdot w)$ input blocks at a cost of $O(b \cdot ov)$. Finally, we arrive at a complexity of $O(n^2 \cdot h \cdot w \cdot b \cdot ov)$.

3. PROPOSED METHOD

This section covers the base implementation of the image quilting algorithm and the subsequent optimizations that were applied.

3.1. Base implementation

As a starting point, we transpiled a MATLAB implementation of image quilting [6] into C, adhering to Algorithm 1, prioritizing correctness and conducting thorough testing. To avoid copying redundant pixel values and improve efficiency, we heavily utilized structs in our implementation. These structs contain a pointer into image, the stride between rows, the number of color channels, and the width and height of the overlap area (see Listing 1). This approach allows us to access the necessary overlap regions for calculating the overlap error and performing the minimum cut directly from the input image.

```
typedef struct {
    pixel_t *data;
    int width;
    int height;
    int channels;
    int stride;
} slice_t;
```

Listing 1. Definition of the slice_t type

calc_errors. The calc_errors function plays a crucial role in the base implementation, specifically covering lines

6&7, 9&10, 12&13 of the provided Algorithm 1. It is invoked at various points in the code to handle the three possible cases that arise when evaluating blocks (see figure 1). It populates the error matrix E, which stores all errors for

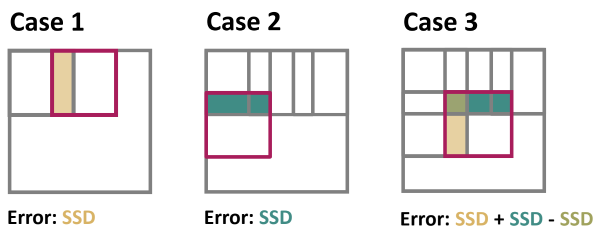

Fig. 1. The three cases when calculating the overlap error

every block in the input image. We note that in the third case, the error matrix E is traversed three times in the base implementation. The first two traversals add the errors of the left and top overlapping region. The third traversal subtracts the double computed top left square from the error matrix.

3.2. Optimization 1: Simple optimization

In this optimization step, our focus was on optimizing the parts of the algorithm that were easily identifiable optimization blockers.

Strength reduction. A first optimization we applied was strength reduction, where we replaced power functions with more efficient multiplication operations in several parts of the code.

Elimination of redundant computations. As mentioned in subsection 3.1, for the sake of simplicity our base implementation calculated the top left square error three times. To optimize this, we modified the code to process the entire overlap area only once, reducing redundant computations.

Efficient memory access. Alongside removing redundant computations we adapted the memory access pattern of the error matrix. Initially, we iterated three times over the error matrix, updating the error for each overlapping area. We then modified the access pattern so that we calculate the final error in one go, resulting in a single iteration over the error matrix.

Instruction level parallelism (ILP) generation. To improve ILP, we employed separate accumulators for each color channel of the pixel (R, G, B). This allowed for independent accumulation of squared differences for each channel, which were then summed together at the end to obtain the final SSD value for an overlap.

Profiling. After applying these improvements, we utilized Intel's VTune profiler to identify performance bottlenecks in our program and discovered that approximately 99.8% of the execution time was spent within the calc_errors function. This finding prompted us to proceed with the next optimization step.

3.3. Optimization 2: Integral images

This section covers the inclusion of integral images as optimization.

Integral images. Recognizing the significance of optimizing the calc_errors function, we then utilize integral images to precompute certain calculations. For that let us expand the SSD formula 1 as follows:

$$SSD = \underbrace{\sum_{q=1}^{Q}\sum_{r=1}^{R} B_{in}^{ov}[q,r]^2}_{\text{Integral: } 4_{\text{adds}}} \tag{2}$$

$$- 2 * \underbrace{\sum_{q=1}^{Q}\sum_{r=1}^{R} (B_{in}^{ov}[q,r] * B_{out}^{ov}[q,r])}_{\substack{\text{For each [q,r]: } 1_{\text{mul}} + 1_{\text{add}} \\ \text{In total: Q * R * 2 ops}}} \tag{3}$$

$$+ \underbrace{\sum_{q=1}^{Q}\sum_{r=1}^{R} B_{out}^{ov}[q,r]^2}_{\substack{\text{negligible, as this integral is calculated} \\ \text{once for all overlaps in the input image}}} \tag{4}$$

The first part of the equation needs only four additions due to the utilization of the integral image. The middle part needs in total $Q * R * 2$ operations. The third part is negligible, as it calculates the integral of the squared values of the output regions, which is performed only once for all overlaps in the input image. In total, we can estimate that approximately $Q * R * 2$ operations are required, resulting in a reduction of $\frac{1}{3}$ in the number of operations compared to the original SSD calculation for that overlap region. It is important to note that the SSD calculation with integral images can only be used if the region of the output image can still be found in the input image. There are four possible cases which are depicted in figure 2. The bigger the ratio of integral SSD is, the bigger the advantage we can draw from this technique.

During this optimization step, we add the concept of integral images to the code from optimization 1, while also moving if-else clauses out of the innermost loop. We renamed the calc_errors function to fill_errors_matrix, to reflect the change and avoid confusion. Additionally, we made a transition from using double data type to unsigned integers for the pixel values and error terms. This change was implemented to enable certain optimizations that we discuss in section 3.4 and ensuring the absence of numerical errors.

Roofline plots. Using Intel Advisor, we generated a cache-aware roofline plot to gain insights into the performance behavior of the image quilting algorithm based on

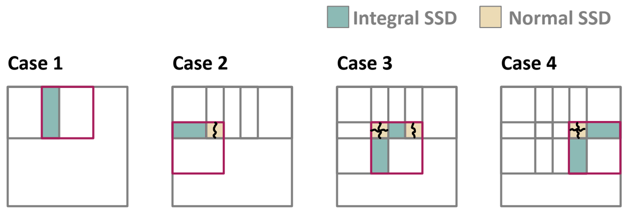

Fig. 2. There are four cases where integral images are utilized. The green areas represent regions that benefit from using the SSD with integral images. On the other hand, the orange areas have undergone modifications through the minimum cut algorithm and therefore require calculating the SSD using the standard approach.

typical input size, overlap size, and block size. Intel Advisor's profiling capabilities allowed us to analyze individual loops within our program. Among these loops, a significant workload was identified within the fill_errors_matrix function. Intel Advisor indicated that this particular loop reached the integer scalar add peak, indicating that this portion of the code was primarily constrained by computational capabilities. Consequently, this finding highlighted the importance of considering vectorization, which is elaborated in the following subsection.

3.4. Optimization 3: AVX2 vectorization

In this subsection, we present our vectorization strategy. We vectorize the calculation of a single entry of the error matrix by simultaneously calculating 16 entries of the SSD. To increase the ILP we unroll the inner loop 8 times to calculate 8 entries of the error matrix in a vectorized fashion and use separate accumulators for each of the eight entries and a separate accumulator for each of the 4 areas as discussed in section 3.3 leading to a total of 32 accumulators.

Integer precision. For vector instructions a smaller memory footprint leads to a larger speed improvement as more elements fit into a vector. Thus we selected the smallest integer sizes that still allow for correct results. Further we split the data types into `uint16` for all input values and `uint32` for all calculated error values (SSD, integral image). `uint16` for input values allow us to read the images with `uint8` color depth without precision loss. Moreover, it enables us to use AVX2 instructions that operate on `uint16` and produce `uint32` values without using any casting operations in the inner loop. `uint32` for the error terms is generally too small to hold all possible values of the integral image, but as `uint32` in C implements modular arithmetic we can make use of the following property (where R denotes the remainder).

$$R(m+n) = R(R(m)+R(n))$$

This property implies that for computations of additions that are only interested in the remainder of the result, calculating the remainders at any intermediate step (thus keeping it inside `uint32` range) does not change the result. The final size of the error is dependent on the overlap between blocks and independent of the input size and entirely fits into a `uint32`. Thus we can use the additional speedup from using the smaller size of `uint32` vs `uint64`.

Vector instructions. As described in section 3.4, there are areas that benefit from integral images and areas that do not. This leads to two different calculations in the innermost loop:

```
diff  = _mm256_sub_epi16(vec_0, vec_1);
mul   = _mm256_madd_epi16(diff, diff);
error = _mm256_add_epi32(error, mul);
```

These are the instructions used for the normal SSD as seen in formula 1.

```
mul   = _mm256_madd_epi16(vec_0, vec_1);
error = _mm256_add_epi32(error, mul);
```

These are the instructions used for the middle part of integral SSD as seen in formula 3.
For both calculations the `madd` instruction does most of the work. It multiplies 16 signed 16-bit integers into intermediate signed 32-bit integers and horizontally adds adjacent pairs together, thus doing 24 INTOPS with a throughput of 2 instructions per cycle and a latency of 5 cycles. If an overlapping region is not divisible by 16 then further cleanup code is needed.

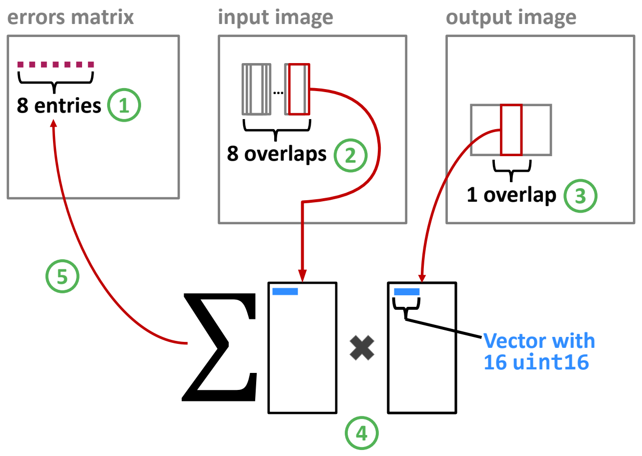

Fig. 3. Effect of unrolling and memory optimization in five steps

Sufficient ILP and memory optimization. In this section we describe our improvement done for ILP and memory access for the vectorized code. For this we describe the steps as seen in figure 3. As already mentioned, we unroll the inner loop 8 times to calculate 8 entries in the error matrix in one loop iteration (step 1). The errors of these 8 entries are gathered in 32 accumulators. In Step 2, the 8 overlapping regions in the input image that are used in the calculation are loaded. As all entries neighbor each other, we have an overall small working set with improved spatial locality. Furthermore, as all 8 overlaps compare to the same overlap in the output image, we only need to load it once (step 3) leading to improved temporal locality. Step 4 then performs the calculation as described in the previous section. Step 5 then stores the 8 entries into the errors matrix in a vectorized fashion.

4. EXPERIMENTAL RESULTS

In this section, we describe our experimental setup and then discuss the experimental results regarding runtime and performance.

Experimental setup. The experiments were conducted on a MacBook Pro featuring an Intel(R) Core(TM) i9-9880H CPU @ 2.30GHz processor with the Coffee Lake microarchitecture, an optimized version of the Skylake microarchitecture. The cache sizes are as follows: L1 data cache 32 KB, L1 instruction cache 32 KB, L2 unified cache 256 KB, L3 unified cache 16 MB. Furthermore, Clang version 13.0.0 was employed for compilation. For compilation the flags `-O3` and `-mfma` were used. We observed that different flags had minimal impact on the runtime.

We vary the following parameters for our measurements:

- input size
- block size
- overlap size
- number of blocks

To measure the impact of these parameters, we created 3 different test cases. For the tests, we vary only one or two parameters while the rest remain at the default values. The default values are: input size of 239 × 239 pixels, block size of 48 pixels, overlap of 8 pixels, and number of blocks of 8 for the output image. The first test varies input sizes from 63 × 63 pixels to 239 × 239 pixels with a step size of 16 pixels. For the second test we varied the block size and overlap from 48/8 to 144/56 and a step size of 16 respectively 8 pixels. The third and last test varies the number of blocks from 5 to 30 with a step size of 5.

Results. Figure 4 presents the results regarding the impact of input size on runtime. It is evident that as the input size increases, the difference in runtime also diverges. This indicates that the optimizations continuously reduce runtime with larger inputs, rather than providing only a constant improvement.

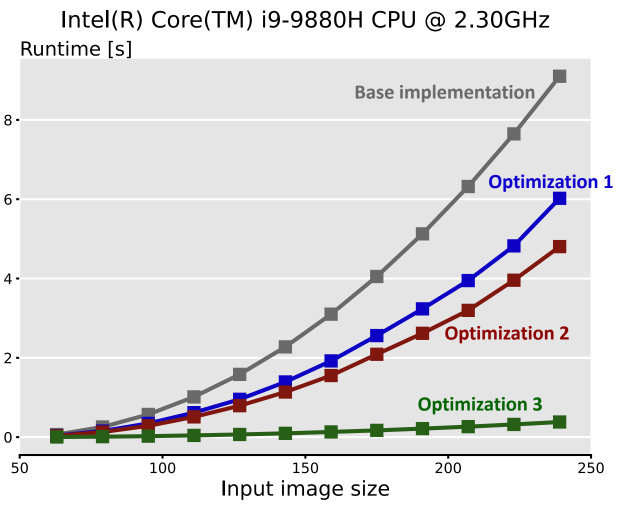

Fig. 4. Runtime measurements with varying size of the input image from 63 × 63 pixels to 239 × 239 pixels with a step size of 16 pixels. Measured for all optimizations. All runtimes increase quadratically, with each optimization reducing the runtime.

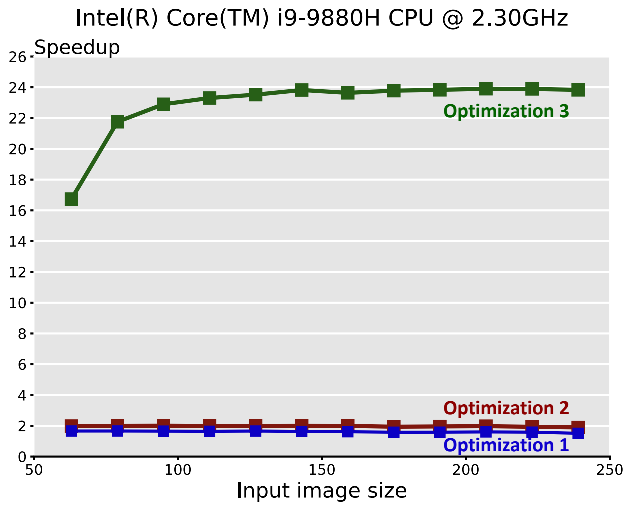

Fig. 5. Speedup plot with varying input size. All input sizes show a constant speedup over all input sizes. The settings are the same as with figure 4.

Looking at the speedups of the optimizations over the base implementation (figure 5), we get a speedup close to 2x for both, the first, and second optimization. With the second optimization being slightly faster. The vectorized version achieves a speedup of 24. After input sizes of 100 these speedups remain constant. As our working set is not dependent on the input size this constant speedup should also be maintained for bigger sizes. Notably, the compiler was not able to vectorize any optimization on its own, even though the `-O3` flag already enables vectorization.

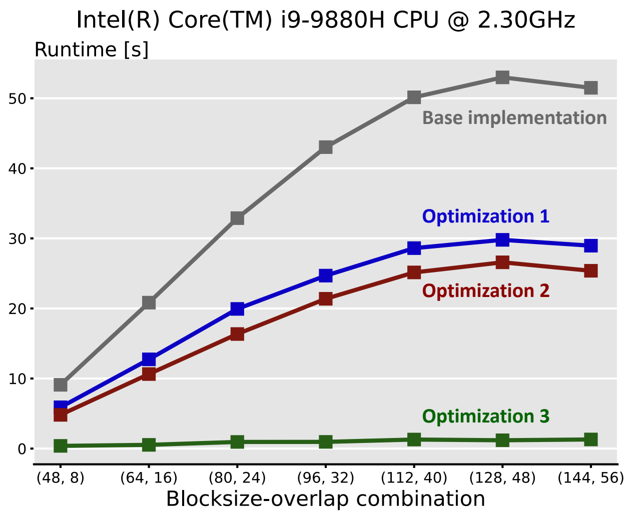

Fig. 6. Runtime plot of the measurement of varying block sizes and overlaps. In the plot a peak is reached at a block size of 128 and an overlap of 48. Afterwards, it decreases due to the reduced number of computations.

In the test where we vary block size and the overlap (figure 6), we can initially observe similarly increasing differences in runtime as mentioned for figure 4. Figure 6 shows, that after reaching the peak at a block size of 128 and an overlap of 48, the runtime declines for all versions. This is to be expected because as the block size approaches the input size, the size of the error matrix becomes smaller and smaller. Having a smaller error matrix can reduce the total number of operations if the added operations due to the larger block size and overlap are overcompensated by the reduced number of candidate blocks. Another remark to include is that the difference in runtime between the base implementation and optimization 1 is considerably larger than between optimization 1 and 2. This shows that removing optimization blockers such as removing function calls, introducing some ILP and changing the memory access pattern significantly speed up the computation. We do not see a diverging runtime between optimization 1 and optimization 2 as in figure 4, this is due to the fact that the ratio of computations that benefit from the integral image is decreasing alongside the ratio of block size to overlap. For example, for block size and overlap (48, 4) we get around 25% less computations due to the integral SSD, while for block size and overlap (144, 56) we get around 17% less computations, since the areas in figure 2, which are computed with the normal SSD, become more prevalent.

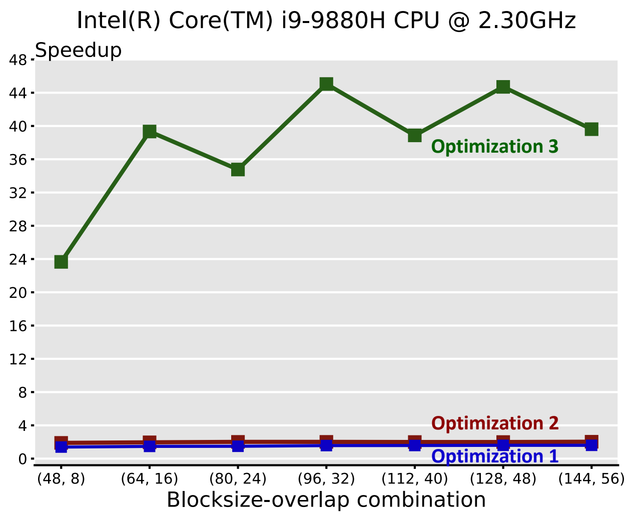

Fig. 7. Speedup plot of the block size and overlap test with the base implementation as reference. The zigzag line of optimization 3 is due to the fact that additional cleanup code is required if the overlap size is not divisible by 16.

In the speedup plot (figure 7) optimization 1 and 2 maintain an almost constant speedup in comparison to the base implementation. The most notable thing is that we are following a zigzag line for the speedup of optimization 3 with increasing block size and overlap. The reason behind this pattern is, that we are working with vectors of length 16. This means, that for every overlap divisible by 16 we can use the full potential of the vectorization. For all other overlap sizes, we have to utilize clean-up code involving blend intrinsics. Moreover, the zigzag line also follows an increasing trend. We think, that with increasing block size and overlap in combination with vectorization, more work can be done with vectorized instructions with less overhead from loading and storing the values.

For the last test, we varied the number of blocks (figure 8). The different implementations follow a similar curve as in figure 4.

Following the study of the runtime and speedup plots, we analyze various performance plots. Figure 9 is the performance plot with varying input size. The plot exclusively showcases the performance of optimization 2 and optimization 3. The reason for this selection is that only optimization 2 and optimization 3 share an identical operations count, making them directly comparable. Within the figure, Optimization 2 demonstrates a relatively consistent optimization value, maintaining stability throughout. For Optimization 3 performance increases until an input size of 100 is reached. Beyond this point, it converges towards a value of 10 [intops/cycle].

The performance of optimization 2 and optimization 3

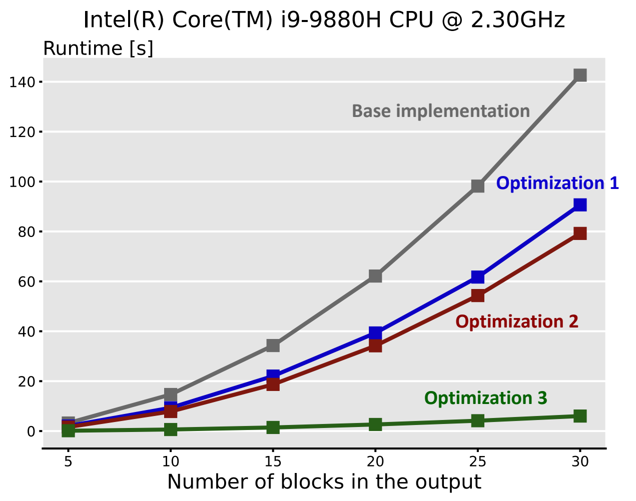

Fig. 8. Runtime plot of the measurement of varying number of blocks. All runtimes increase quadratically, with each optimization reducing the runtime.

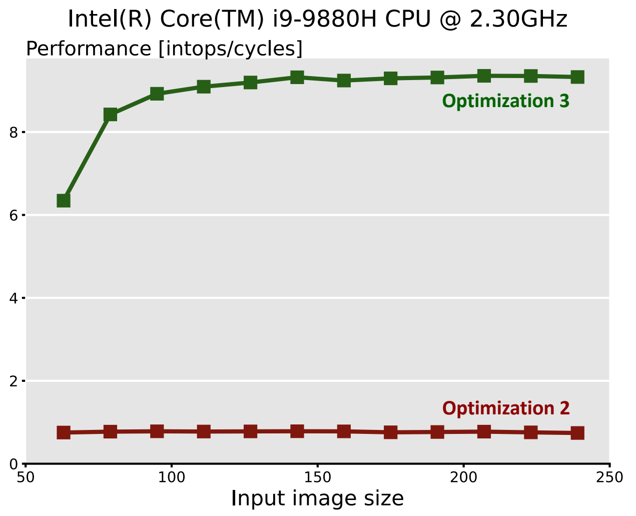

Fig. 9. Performance plot of optimization 2 and 3 for the input size test. Optimization 3 converges to 10 [intops/cycle], whereas optimization 2 remains stable at a performance of 1 [intops/cycle].

for the block size - overlap test is illustrated in figure 10. The plot again reveals the recurring zigzag pattern in optimization 3, attributed to the presence of cleanup code. On average, this pattern exhibits a slightly decreasing trend. One potential explanation could be that the increasing share of normal SSD calculation vs. the more efficient integral SSD hinders performance.

The performance curves of optimization 2 and optimization 3 for the number of blocks test, as depicted in figure 11, exhibit a noteworthy characteristic. They start at a higher

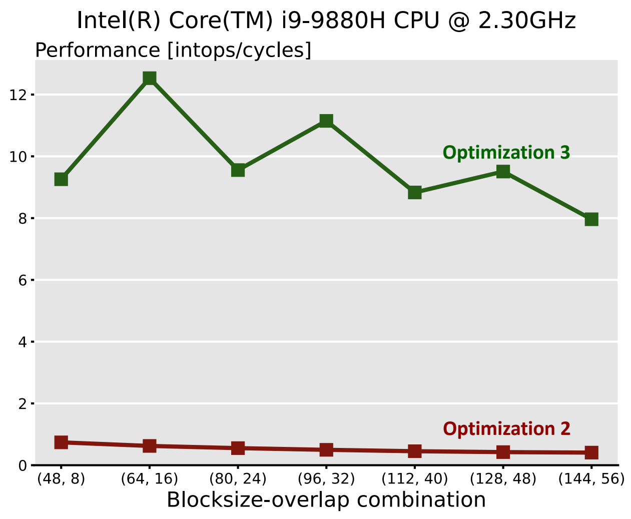

Fig. 10. Performance plot of optimization 2 and 3 for the test with varying block sizes and overlaps. The zigzag line from figure 7 is visible with a decreasing trend.

performance performance values and subsequently decline in performance. As in the setup the number of blocks in the output is varied, it is plausible that an increase in the output image size results in a corresponding expansion of the working set. Consequently, a larger working set lead to more cache evictions in L1 and L2 caches, thereby contributing to the observed decline in performance.

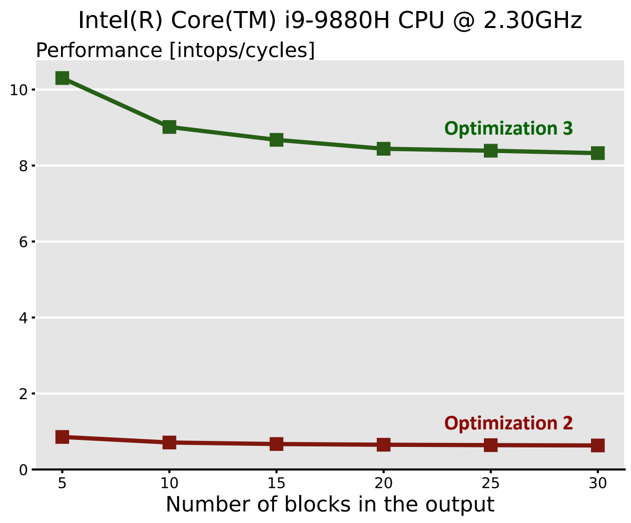

Fig. 11. Performance plot of optimization 2 and 3 for the number of blocks test. A decrease of performance from optimization 3 is due to increased working set which results in more cache evictions.

To put these performance measurements into perspective, we visualized the performance of our algorithm for

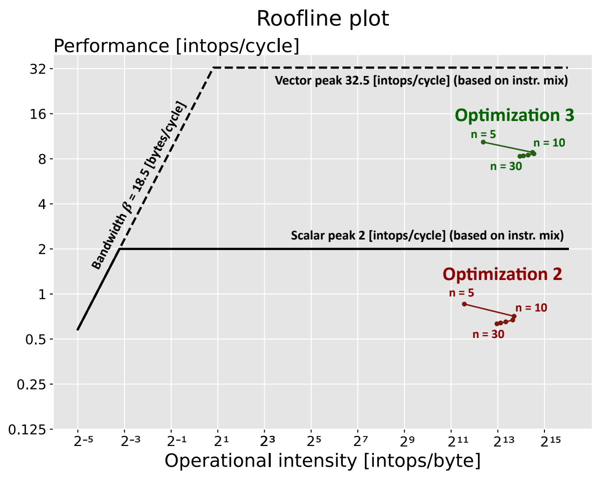

Fig. 12. Roofline plot for number of blocks test. The plot is created manually with the results of the measurements. The number of blocks used in the output is denoted here by n.

varying number of blocks in a roofline plot (figure 12). The platform we use operates with a memory bandwidth of about 18.5 [bytes/cycle]. Furthermore, we assumed that all the data used fits into the cache. Therefore, we only calculated the compulsory misses of all matrices used. For both measurement series we determined the peak performance based on the instruction mix. For the (scalar) optimization 2 we have a maximum peak performance of 2 [intops/cycle] and for optimization 3 a maximum peak performance of 32.5 [intops/cycle]. It is apparent that neither optimization is memory bound. Nevertheless, our estimated upper bound for the scalar peak performance is not reached. Which is true in some cases and not in others. Interestingly, according to the roofline plot generated by the intel advisor we do in fact hit the peak performance for optimization 2. On the one hand we think that dependencies between instructions might further restrict our calculated upper bound of the peak performance. On the other hand we only count integer operations that are relevant to calculate the output of the algorithm. Other instructions, e.g., index calculations, might still have an effect on the performance. Given that the performance curves (figure 11) for both optimizations follow the same pattern, it shouldn't come as a surprise that we see a similar pattern for optimization 2 and 3 in the roofline plot, when the number of blocks is varied. The operational intensity first increases rapidly and then decreases steadily with larger outputs. We can conclude that the best code achieves approximately 38% of the peak performance.

5. CONCLUSIONS

In this project, we successfully implemented and optimized the image quilting algorithm. Throughout the process, we employed various optimization techniques such as strength reduction, altering memory access patterns, and utilizing integral images, among others. However, it was the AVX2 vectorization that yielded the most significant improvement. With a peak performance of 12.5 [intops/cycle] and a remarkable speedup of 45 compared to our baseline implementation, it proved to be the most impactful enhancement. Future work could involve incorporating additional accumulators to fully optimize the pipeline and maximize overall performance.

6. CONTRIBUTIONS OF TEAM MEMBERS

6.1. Christof

Added data structures for the base implementation. Worked with Lukas to implement optimization 1 and optimization 2, especially adding ILP and removing memory calls. Implemented the vectorization of optimization 3 with all the integer precision handling on the basis of the work of Marco & Piero.

6.2. Lukas

Worked on the minimum error boundary cut in the base implementation. Contributed to optimizations in optimization 1. Worked on implementation of integral images. Generated roofline plots using Intel Advisor. Collected flop count information for performance plots.

6.3. Marco

Worked on the base implementation. Created the benchmark infrastructure including the plotting. Regarding the optimization, helped with the integral images and created the foundation of the vectorized optimization.

6.4. Piero

Contributed to the project by implementing the base implementation, excluding the minimum boundary cut part. Moreover, worked on optimizing parts of the integral images and assisted in vectorizing the code. Analyzed the performance by utilizing the Intel VTune Profiler for profiling the code. Additionally, created informative plots using the data generated from the benchmarks.

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*, 2001, pp. 341–346.

[2] Leandro Cruz, Luiz Velho, Eric Galin, Adrien Peytavie, and Eric Guérin, "Patch-based terrain synthesis," in *International Conference on Computer Graphics Theory and Applications*, 2015, pp. 6–pages.

[3] Daniel Bug, Gregor Nickel, Anne Grote, Friedrich Feuerhake, Eva Oswald, Julia Schüler, and Dorit Merhof, "Image quilting for histological image synthesis," in *Bildverarbeitung für die Medizin 2020: Algorithmen–Systeme–Anwendungen. Proceedings des Workshops vom 15. bis 17. März 2020 in Berlin*. Springer, 2020, pp. 322–327.

[4] Paul Viola and Michael Jones, "Rapid object detection using a boosted cascade of simple features," in *Proceedings of the 2001 IEEE computer society conference on computer vision and pattern recognition. CVPR 2001*. Ieee, 2001, vol. 1, pp. I–I.

[5] Lin Liang, Ce Liu, Ying-Qing Xu, Baining Guo, and Heung-Yeung Shum, "Real-time texture synthesis by patch-based sampling," *ACM Transactions on Graphics (ToG)*, vol. 20, no. 3, pp. 127–150, 2001.

[6] Aeris Jordan Mecom, "Implementation of image quilting by efros and freeman," https://github.com/jmecom/image-quilting/, 2016.