OPTIMIZING TEXTURE SYNTHESIS

Alessandro Cravioglio, Sofia d'Atri, Arseni Loika, Alessandro Zanzi

Department of Computer Science
ETH Zurich, Switzerland

ABSTRACT

The following report describes the optimization of a texture synthesis algorithm based on neighborhood similarity. First, the baseline implementation is described, with the relevant cost and profiling. Then follows an in-depth exploration of the applied optimizations, split into scalar and vectorized stages.

The key result is the achievement of significant performance gain relative to the hardware on which the experiments were run. Specifically, the vectorized optimizations proved to be the most potent to gain performance, while the scalar optimizations enabled alternative approaches that have equally proven to be of high significance.

1. INTRODUCTION

Motivation. Texture synthesis based on neighborhood similarity is an important texture generation algorithm used in computer vision, graphics, and image processing. It has many valuable applications, including image and video reconstruction, texture filling and geometric modeling. For example, the process can be applied to old damaged digitized images to procedurally fill damaged or missing parts, using the information from the rest of the image.

The goal of this algorithm is to process the input texture and synthesize a new one, of greater dimensions, without visible artifacts or visual repetition. However, the algorithm is computationally expensive, especially for larger images with sizes closer to the real world use cases. Fast implementations of the algorithm are difficult to obtain, mainly due to the underlying nature of the filling method, which involves multiple calculations for each output pixel, possibly several times.

Our work is based on the texture synthesis algorithm described in Li-Yi Wei [1], which outlined its overall structure with pseudocode. We were also able to analyze a publicly available Python implementation [2], which served as a starting point for our baseline C implementation.

The Python version proposes an implementation of the algorithm for input with dimensions of the form $2^l \cdot m$, where l is the number of Gaussian pyramids and m is an arbitrary integer. Overall, this version was not used for performance comparison, since it is implemented in Python and therefore is significantly harder to compare to any C version.

Contribution. In this paper, we propose an implementation of the texture synthesis problem for square images of dimensions that are powers of 2. Specifically, we measure performance of several distinct image sizes that are multiples of 8. Those sizes are optimized for cache locality, with higher potential for benefit from vectorization.

2. BACKGROUND ON THE ALGORITHM

The algorithm. The goal of Texture Synthesis is to generate a new image, of larger dimensions, such that it is visually similar to the input image I_i. Together with the input texture, the algorithm needs a random noise image I_o, with desired output size *S*, that is used as a ''canvas" to be filled to resemble I_i. For convenience, the key symbols are summarized in Table 1.

Symbol	Meaning
I_i	Input sample image
I_o	Output image
G_i	Gaussian pyramid built from I_i
G_o	Gaussian pyramid built from I_o
p_i	An input pixel from I_i or G_i
p_o	An input pixel from I_o or G_o

Fig. 1. Symbols Reference

The synthesis is done pixel by pixel, in a roster scan ordering (top-to-bottom, left-to-right). The value of a pixel p_o is chosen by comparing its neighborhood with all possible neighborhoods in I_i.

The input pixel p_i with the most similar neighborhood is then assigned to p_o. This is a single-level approach that works well with textures that present little visual structures or patterns. However, the bigger those structures become, the larger the neighborhood needs to be, requiring significantly more computation. This is the reason our approach

revolves around the multiresolution synthesis, which includes the following key steps:

1. Generate random noise of the output size.
2. Compute Gaussian pyramids G_i and G_o.
3. At each pyramid level (from coarsest to finest), compute output pixels by matching neighborhoods against those in G_i, using information from the current and all coarser levels (if available).
4. Fill pixels in raster scan order, minimizing the distance between neighborhoods [1].

The whole process is fully deterministic: given the same input image I_i, noise image I_o, and fixed pixel traversal order, the output remains identical. Since I_o is not used during computation, except for edge case handling [1], reproducibility is guaranteed by using a fixed random seed. This determinism has simplified the validation of optimizations, as outputs can be visually compared to a baseline to confirm that no synthesis artifacts were introduced.

Key Components. The algorithm described above can be divided into four principal components that will be briefly explained, since they are a fundamental part of the implementation and roughly correspond to the modules that were optimized the most.

Building Gaussian pyramid. This module constructs the Gaussian pyramid of an image with a specified number of levels. The original image is level 0. Each subsequent level is created by applying a Gaussian filter to the previous level and then downsampling by a factor of 2.

Building neighborhood. For a given input pixel p_i and a level of Gaussian pyramid G_n, this module returns its surrounding pixels, according to the neighborhood mask. Given that the neighborhood can be of any size, we decided to keep it at a constant 12 pixels, L-shaped.

Matching neighborhood. This submodule is responsible for calculating the sum of squared differences between two provided pixel neighborhoods.

Finding best match. Combines the neighborhood modules, iterating over each pixel of the input texture Gaussian pyramid level, and returning the pixel with the most similar neighborhood according to the similarity measure.

Cost Measure. In this project, the most appropriate cost measure is integer operations. The choice is driven by the fact that this is the only type of arithmetical operations in our code. It is important to note that it was decided not to consider the index operations, since they remain constant between different optimizations when operating on same size images.

Asymptotic Cost. The overall asymptotic cost was estimated as $O(L \cdot w^4 \cdot N)$, where w refers to the width of the input image, N is the number of neighborhood pixels, L is the number of levels in the Gaussian pyramid (i.e. the most asymptotically expensice elements of the code).

Cost Analysis. The table of costs (in Gops) for each optimization, as well as the baseline, is found below. It was used to plot the performance of each version against the below inputs sizes.

Size	Baseline	V1	V2	V4	V5
32	1.89	1.84	1.49	1.71	2.41
64	32.29	29.43	23.84	27.35	38.52
128	484.47	470.47	381.40	437.52	616.18
256	7,751.16	7,531.27	6,102.09	6,999.99	9,858.40

Fig. 2. Cost summary in giga-integer operations

The observed growth follows an exponential trend, which is not surprising given the asymptotic cost. Notably, the cost of version 3 is equal to that of version 2, similarly for versions 5 and 6, which is why they are omitted from the table. This is because V3 and V6 were the inlined versions of their corresponding predecessors, so the number of measured operations did not change. Another notable part is the increase in the number of operations for version 5, which was caused by additional shuffles and horizontal sums from vectorized optimizations.

3. OPTIMIZATIONS PERFORMED

3.1. Overwiev and Profiling

Overview. This section focuses on the optimizations that were performed on the texture aynthesis algorithm. First, the analysis of the baseline implementation that guided the optimization decisions is explained, after which first scalar and then vectorized optimizations are evaluated.

Profiling. An important step to guide our optimizations was to profile the base implementation. Intel VTune was used for that purpose, since the experimental setup was composed of Intel hardware. As a profiling tool, it was considerably successful in driving the optimization decisions.

Function	**CPU Time**	**% of CPU Time**
build_neighborhood	41.099s	71.4%
match_neighborhood	15.260s	26.5%
GI_libc_free	0.540s	0.9%
GI_libc_malloc	0.400s	0.7%
find_best_match	0.230s	0.4%

Fig. 3. Baseline profiling results of 64×64 input synthesized to 128×128. **No compiler optimization flags.**

As visible on 3, the initial analysis of the baseline for all input sizes clearly demonstrated that the hot spots of the program, taking over 95% of CPU time, were the build_neighborhood and match_neighborhood functions. Thus, the majority of the optimizations focused specifically on them. It was aligned

with our expectations of the initial performance, since those two functions are called the most number of times by the main function. Importantly, through roofline model analysis, it was determined that the program is compute-bound, as is demonstrated in the plot on Fig 4.

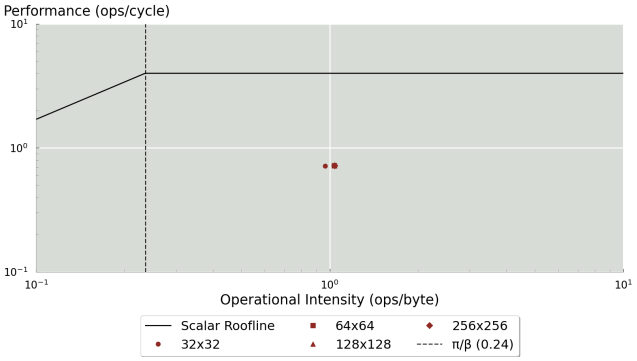

Fig. 4. Roofiline plot

The roofline analysis was done with the value of π (peak theoretical performance) equal to 4 IOPs/cycle [3] and β (memory bandwidth) of 7, which was calculated using the profiled bandwidth in GB/s, and converted to bytes/cycle using the CPU base frequency as reported in Table 5. It was preferred to use the profiled estimate rather than Intel specifications for our processor (45.7 GB/s = 17 bytes/-cycle [4]), since it was more realistic with the given program at the moment of experimentation. Also, it did not change the fact that the program is compute-bound. Therefore, it was known that program would benefit the most from computation-related optimizations, rather than memory-related.

Meanwhile, the estimate for memory accesses was measured with VTune, given as the number of loads and stores. Since it was impossible to obtain the data in bytes, it was decided that this approach still gives a reasonable estimate for the behavior of the program and its optimizations with respect to operations and data movement.

3.2. Optimizations Context

Algorithm adaptations. Certain assumptions and restrictions were made in order to better facilitate the optimizations. Firstly, the input image was restricted to be a square, with dimensions being a power of 2. This enabled the substitution of expensive arithmetic operations, such as modulo and multiplication, with bitwise operations. Meanwhile, strictly in order to standardize the performance measurements, the output image was agreed to have twice the dimensions of the input image (though it is possible to synthesize a texture to other sizes).

Also, the input sizes were multiples of 8 (32, 64, 128, 256), which means that they had much better cache locality given the L1 cache having sets with a multiple-of-8 capacity on each set - as reported on Fig. 5. Furthermore, such sizes fit better in `__m256i` vectors, which enabled us to take better advantage of vectorized optimizations.

Overall approach. The first round of optimizations focused on scalar speedup techniques. The following round focused instead on vector optimizations, specifically by using Intel intrinsics. Before exploring the optimizations, the baseline implementation of the most important routines is described.

Build neighborhood structure. Begins by initializing neighborhood sizes and allocating an integer array to store pixel data, minding RGB-channels. It then iterates over the current level *L* neighborhood, applying toroidal boundary conditions to compute valid pixel indices. Each pixel's channel is accessed via 1D indexing (e.g. `M[(i*k)+j]`) and appended to the output array.

If the level *L* is not the topmost, an `if` block triggers a second loop over the upper-level neighborhood. Upon any allocation or bounds violation, the function frees allocated memory and returns a `null` value. The final output is a contiguous array of RGB values representing the combined neighborhoods across the levels of the Gaussian pyramid.

The function has limited spatial locality, as neighborhood pixels are accessed with non-contiguous, toroidally wrapped offsets, often spanning different cache lines. Temporal locality is minimal, since the pixel values are not reused across calls or iterations.

Match neighborhood structure. This function computes the squared Euclidean distance between two neighborhoods, *Na* and *Ns*, each of total size *l*, iterating through their elements with a `for` loop and using an accumulator to store the result value. Here, spatial locality is ensured since elements of the two neighborhoods are accessed sequentially, while there is no temporal locality since each element is accessed only once (without counting the immediate second access). Importantly, the number of iterations through the loop is already known, since it is the total number of pixels in the neighborhood, thus leading to the possibility of more efficient unrolling.

3.3. Scalar Optimizations

Standard optimizations for build_neighborhood. In the first round of scalar optimizations, the focus lay on standard strategies which were deemed appropriate given the result of code analysis and profiling. They included strength reduction, scalar replacement, pointer arithmetic handling, and loop unrolling.

As stated above, the baseline implementation makes extensive use of the modulo (`%`) operator for the calculation of toroidal coordinates that wrap around the edges of each Gaussian pyramid level:

```
(x + dx + G[L].width) % G[L].width;
```

This arithmetic operator is computationally expensive, since it is implemented using division plus remainder. Our aforementioned restriction on the size of the input image has enabled an alternative calculation - with a bit-wise AND (`&`), which executes in a single CPU cycle:

```
(x + dx + L_width) & (L_width - 1)
```

This transformation preserves the correctness of the toroidal wrapping while improving computational efficiency, particularly in performance-critical loops that access neighborhoods for each pixel.

The above snippets also demonstrate the scalar replacement strategy, which in this case reduces the number of memory accesses to retrieve the width or height of the Gaussian pyramid levels `G[L].width` (or `G[L].height`). In the baseline implementation, these fields were accessed multiple times within the loops, potentially incurring unnecessary memory latency. Therefore, temporary local variables were introduced, such as `L_width` and `L_height`.

Then, a series of substitutions were performed using pointer arithmetic wherever appropriate, aiming to reduce indexing overhead and improve low-level memory access. For example, expressions of the form:

```
pixels[idx++] = G[L].data[index];
```

were replaced with equivalent pointer-based operations:

```
*p++ = *(next_data + index0);
```

However, as evaluated through profiling, this strategy resulted in little to no real impact on performance. That is likely due to the fact that compiler already performs this optimization by itself, especially when using `-O3` flag.

As stated before, the number of iterations through each loop is already determined by the size of the neighborhood, i.e. 12 pixels in the current neighborhood and 9 pixels in the upper neighborhood. That suits very well to a total unrolling of the two loops, totally removing the branch prediction cost and making it easier for the compiler to perform memory optimizations. Moreover, each iteration is structurally identical, and the offsets are known ahead of time. In the unrolled version, all of these operations are replicated inline with constants replacing the loop index. This is particularly effective with those deterministic access pixel-wise operations with high spatial locality.

Memory reuse. As could be seen from the profiling information (including Fig. **??**), the use of `malloc` and `free` incurred additional overhead, albeit small. However, upon closer analysis, it was derived that their impact could be much greater - since the overhead of allocating and freeing memory was likely included in the profiling for build and match neighborhood functions.

The reason for this likely is that `malloc` and `free` are called from there the most - at the beginning the memory is allocated for the neighborhood pixels, and at the end it is freed.

Therefore, it was decided to pre-allocate the memory for the neighborhood pixels only once in the main function and reuse it for each subsequent call to match_neighborhood(). It did not incur any potential segmentation faults, as the neighborhood remains the same size throughout the execution.

match_neighborhood. The simple structure of the match_neighborhood() function, seen below, enabled straightforward optimizations.

```
for (int i = 0; i < l; i++)
   res += (Na[i]-Ns[i])*(Na[i]-Ns[i]);
return res;
```

The first optimization was total loop unrolling. It was possible since the dimensions of the neighborhood were agreed to be constant, so the number of iterations `l` was known to stay unchanged, specifically at $(12 + 9) * 3 = 63$. This part provided certain speedup by removing the loop control and test instructions.

Then, the differences for each `Na[i] - Ns[i]` were precomputed and assigned to a variable, in order to avoid redundant computations. Those optimizations resulted in the new function structured as follows:

```
int diff_0 = Na[0] - Ns[0];
...
int diff_62 = Na[62] - Ns[62];
return
    (diff_0*diff_0)+...+(diff_62*diff_62);
```

Another strategy was attempted, which was to assign each `Na[i]` and `Ns[i]` to a variable to avoid redundant indexing. However, it was found that it provided no measurable speedup, likely since the compiler was able to cache the entire contents of `Na` and `Ns` in L1 cache (details in Fig. 5), and retrieving those was fast enough as a result.

Scalar inlining. This was the last scalar optimization performed to avoid interference with previous modular optimizations and to maximize the compiler's ability to optimize fully transformed code, and simply involved the process of exposing the internal logic of previously encapsulated routines. As a result, the routines were not longer treated as a ''black box" by the compiler, thus enabling further optimizations from its side.

This was performed by extending the declaration of each call to build_neighborhood() and match_neighborhood() from their main caller function (find_best_match()) starting from

the main file. The inlining was not applied to the Gaussian pyramid building function since by itself it already had negligible impact on CPU time, and provided no measurable speed up.

3.4. Vectorized Optimizations

Given the structure of the code after the scalar optimizations, it was evident that, above all in match neighborhood and build neighborhood functions, vectorization of the code using AVX intrinsics was a natural next step. We opted for AVX 256-bit instead of the 512-bit due to both flexibility and the structure of our data.

Vectorization of build_neighborhood. Specifically, in this function, the neighborhood data must be logically separated into low and high index regions; AVX 256 provided more granularity for operating on these regions without introducing excessive control logic or data shuffling. Thus, an eight-lane structure of AVX2 56 was used to split the neighborhood offsets between two logical groups - high and low. Below an example of such structure for the high regions can be seen - similar logic follows for low regions.

In contrast, using AVX 512, many lanes would have remained unused or would have required masking and reshuffling, compromising part of the performance gains.

```
__m256i indices_hi = _mm256_mullo_epi32(
      _mm256_add_epi32(_mm256_mullo_epi32
      (x_offsets_hi, width_vec),
      y_offsets_hi), three_vec);
```

In the snippet above it can also be noted that instead of FMA, the addition and multiplication had to be used separately in our case. This is because those instructions operate exclusively on floating-point data, whereas our indices are represented as 32-bit integers. Therefore, we relied on a combination of `mullo_epi32` and `add_epi32` to compute the address offsets in a vectorized form. Overall, the process of transforming the code from a scalar to a vectorized version was made easier by the previous unrolling and the limited number of elements to work with, and it was easy to perform the mapping without requiring complex data rearrangement.

match_neighborhood. Here, the strategy focused on processing 8 pixel at a time, using `__m256i` vectors. It was done by loading 8 elements at a time from `Na` and `Ns` arrays, performing subtraction and multiplication with `_mm256_sub_epi32` and `_mm256_mullo_epi32`, and storing the result in a vector of 8 intermediate sums. Finally, a horizontal sum of the vector was calculated to return the result.

There was a separate effort to unroll the loop entirely using the same core logic. However, it was concluded through profiling that the total unrolling was marginally slower, therefore the non-unrolled version was kept. This was likely due to the fact that the loop control and test instructions offered minimally less overhead than having to process comparatively more AVX shuffles.

Function inlining. As in the scalar version, the function inlining was left to be implemented last, to add a final speedup after all major transformations were applied. This made the full data flow available to the compiler, enabling more aggressive register allocation and loop instruction coalescing around the vectorized inner loops.

While this optimization contributed to tighter instruction packing and measurable performance gains, it was less pronounced than in the scalar version. This is mainly because 1) function overhead is already reduced by vectorization which reduced its relative cost, 2) most of the optimizations now possible by the compiler are already done manually during the previous vectorization, and 3) the low level structure is already clear and the logic is already exposed enough to the compiler.

Summary. Overall, our strategy was exhaustive in tackling the most significant bottlenecks up to a considerable extent. The choice for further manual optimizations was limited without the possibility of changing the structure of the code, e.g. through using more complex data structures.

4. EXPERIMENTAL RESULTS

Overview. Here the results of the optimizations are presented and discussed - first scalar, then vectorized. Before the results are evaluated, the experimental setup is described, specifically, the hardware on which the program was run and optimizations performed.

Experimental setup. The hardware on which the optimizations were measured are as follows:

Name	Intel Core i7-10850H
Microarchitecture	Comet Lake
CPU Base Freq	2.7GHz
L1 Cache	384 kB, 8-way SA
L2 Cache	1536 kB, 4-way SA
L3 Cache	12 MB, 16-way SA

Fig. 5. Hardware specifications

Compiler. Several combinations of compiler flags were experimented with during optimizations as well as accurately measuring the performance of each optimizations. The breakdown is as follows:

`-O3 -fno-tree-vectorize`. Those flags were used to measure the performance of the top scalar optimizations of the compiler - to set a target with our own scalar optimizations. They were therefore also used to measure the

effect of our scalar optimizations.

`-march=native`. This flag was was added to the above to see how our code performs at its highest compatibility with our chosen processor. While it offered limited speedup in some cases, it was used cautiously so as not to invoke vectorization of the instructions at scalar stage. Another flag that was experimented with was **`-ffast-math`** but it did not provide any measurable benefit, likely since our algorithm does not use any complex mathematical operations to have speedup from it.

`-O3 -mavx2`. Those flags were used to test compiler's own vectorized optimizations, as well as our own. Note that **`-mavx`** was not used over compatibility issues of certain intrinsics instructions.

Input. The input that was used to measure performance was standardized into 4 sizes: $32 \times 32, 64 \times 64, 128 \times 128$, and 256×256. In the smallest case, the computation, even at baseline, took less than a second, while in the largest case the baseline took more than an hour. This means the input size spread was sufficient to demonstrate adequately how any given optimization behaved.

Scalar results. The results of the scalar optimizations are split into several versions, seen in Fig. 6. Following the plot there is a more informative description of each version as well as which optimizations it contains.

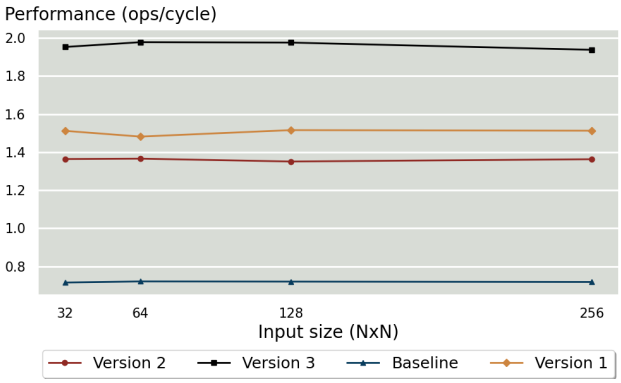

Fig. 6. Performance of scalar optimizations

V1: build neighborhood: scalar replacement, strength reduction, loop unrolling; match neighborhood: loop unrolling + pre-computation + scalar replacement

V2: build neighborhood: memory reuse, partial loop unrolling, strength reduction; match neighborhood: same as version 1

V3: build neighborhood and match neighborhood functions inlined inside their caller function

Version 1. This version took the most advantage from the standard optimizations described above. The expected speed-up was approximately $\times 2$ since those optimizations addressed a considerable part of the bottlenecks and inefficiencies that come from using comparatively expensive instructions.

The average speed-up achieved was indeed $\times 2.1$ on average between different input sizes, with the new performance averaging at 1.5 ops/cycle as compared to 0.7 at baseline.

Version 2. By itself, this version was not expected to give a boost higher than version 1. Our estimates were up to $\times 2$ speedup due to the possible resolution of the significant overhead caused by repeated dynamic memory de-/allocation. Indeed, the measured speedup was $\times 1.9$ which aligned with our reasoning.

The performance itself was naturally not as high as that in version 1 - but it was not far: 1.4 as compared to 1.6 ops/cycle on average. This is likely due to the other optimizations being implemented alongside memory reuse. Still, the key point of the experiment was to determine the impact of memory reuse, which confirmed that it was indeed a significant bottleneck.

Version 3. In this version the main expected benefit was to allow the compiler to not treat certain procedure calls as black boxes. The expected speedup was anywhere in the neighborhood of $\times 3$ - a conservative estimate from the combination of previous two speedups. The obtained speedup was $\times 2.7$, likely since our own optimizations were closer to what compiler would have done had it not treated the function calls as black boxes.

The new performance averaged at just under 2 ops/cycle - approximately half way to the peak theoretical (scalar) performance. Despite the completed optimizations, the synthesis is still done one pixel at a time, which was likely causing significant pipeline stalls. This could be resolved by processing many pixels at a time, possibly through vectorization or through modifying the function itself. In the next subsection, the former is implemented and analyzed.

Vectorized results. The results of the vectorized optimizations are seen in Fig. 7.

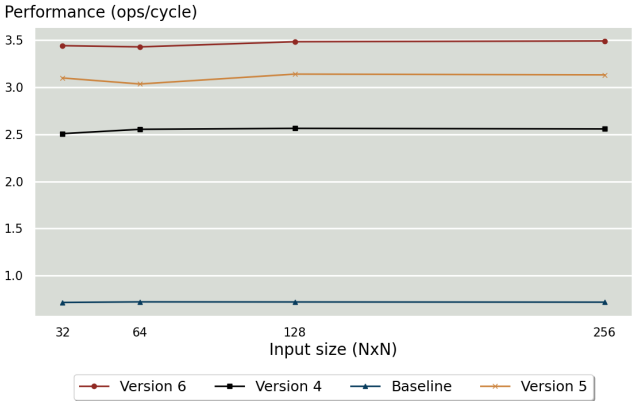

Fig. 7. Performance of vectorized optimizations

There are three different implementations, that can be summarized as follows.

V4: build neighborhood: same as version 2; match neighborhood: vectorized loop, 8 integers at a time

V5: build neighborhood: vectorized calculation with coordinates of each pixel's neighborhood; match neighborhood: loop unrolling by 8, calculating 8 intermediate sums at a time

V6: vectorized build neighborhood and match neighborhood inlined inside their caller function (find best match)

The key findings from the three different vectorized implementations are listed below.

Version 4. For this version the expected speedup was not very high, possibly up to $\times 3$, since only neighborhood matching function was vectorized. It should be noted that this was as an intermediate step to better understand the impact of vectorizing each function, and how it accelerates the program with reusing the memory.

The obtained performance was just above 2.5 ops/cycle on average, giving $\times 3.5$ speed-up on average. This has exceeded our expectations, possibly due to the fact that the compiler's scalar optimizations of build neighborhood function did not act as a bottleneck combined with the vectorized match neighborhood function.

Version 5. Here, the expected speed-up was approximately $\times 4$ from the baseline, since we were now utilizing more ports with integer operations each cycle.

The observed speed-up was $\times 4.3$, which is likely a result not only of the parallel execution of instructions but also from improved compiler visibility that enabled better register allocation and reduced branching overhead.

The performance of this version was already notably higher than the best performance of the scalar optimizations: 3 ops/cycle as compared to just under 2 at version 3. This already shows a much more efficient resource utilization given the vectorized code, which is understandable since the program has been determined to be compute-bound.

The observed gain was $\times 4.3$, likely from both instruction-level parallelism and improved compiler optimizations such as better register allocation and loop coalescing. Performance reached just above 3 ops/cycle—significantly higher than the scalar peak of just under 2, already showing a much more efficient resource utilization given the vectorized code, which is understandable since the program has been determined to be compute-bound.

Version 6. On top of parallel calculations, this version benefited from the compiler's own optimizations since it did not treat procedure calls as black boxes.

The expected speed-up was approximately $\times 4$ from the parallel execution of instructions, and a further $\times 1.5$ from the inlining, as observed in the scalar version. The actual observed speed-up was $\times 4.8$, suggesting that while inlining provided some speedup, its impact was partially overlapped by the optimizations already achieved through vectorization. Furthermore, it is possible our manual optimizations were already performing a part of what compiler would have done by itself from function inlining.

The performance of this version is the biggest as compared to the rest: approximately 3.5 ops/cycle on average, meaning it took the most advantage from ILP as well as efficient port utilization.

Roofline analysis. In order to examine the effect of our strongest optimizations, a roofline plot was made, found in Fig. 8.

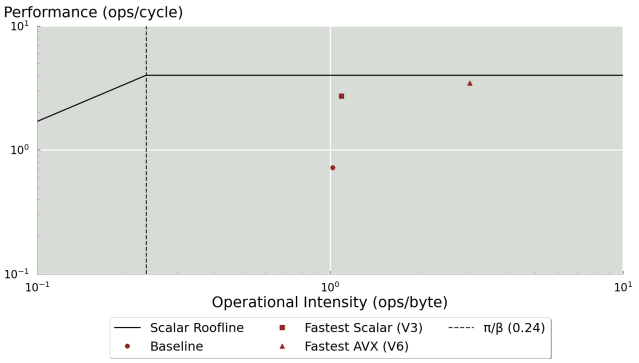

Fig. 8. Roofline plot of fastest scalar and vectorized versions

As can be seen from the plot, our optimizations have almost reached the scalar roofline. While it shows the quality of scalar optimizations, it means the vectorized optimizations still had unfulfilled potential as they did not close in on the vectorized roofline (omitted). This is likely due to the fact that our vectorized approach still contained much overhead from the use of rudimentary shuffles and other grouping operations that is less efficient than compiler's own approach.

5. CONCLUSIONS

In summary, the team was able to optimize the texture synthesis program to a significant extent, achieving a gain in performance $\times 4.8$ at peak, with 3.5 operations per cycle with vectorized optimizations.

The scalar optimizations, meanwhile, were able to achieve consequential speed-up by not only implementing the standard optimizations, but also experimenting with the dynamic memory management of the program.

In conclusion, the work on optimizing this program is of notable importance since it shows that it is possible to achieve a considerably fast working version of texture synthesis without changing its simple structure, relying on advanced data structures or more complex synthesis techniques.

6. CONTRIBUTIONS OF TEAM MEMBERS

Alessandro Zanzi. Implemented a vectorized version of match neighborhood function (which was its most performant version). Implemented improved memory management and optimizations in both scalar and vectorized versions.

Alessandro Cravioglio. Focused on optimizations on build_neighborhood(), both scalar and and vectorized, in version 1 and 4. Also implemented the inlined version of the best vectorized optimizations (version 6).

Sofia d'Atri. Focused on the inline_scalar branch (version 3 of optimizations) and cooperated for version 1 with Alessandro and Arseni. Produced benchmarking infrastructure, ran benchmarks and created performance plots.

Arseni Loika. Focused on both scalar and vectorized optimization of the match_neighborhood() function, in versions 1 and 4. Profiled the code for hotspots and memory access analysis. Created the roofline plot.

References

[1] Marc Levoy Li-Yi Wei. Fast texture synthesis using tree-structured vector quantization. *ACM*, 2000.

[2] Dionysios Spiliopoulos. texture_synthesis. https://github.com/Dspil/texture_synthesis, 2025.

[3] WikiChip chips and semi. Skylake (client) - microarchitectures - intel. https://en.wikichip.org/wiki/intel/microarchitectures/skylake, 2024.

[4] Intel. Intel core i7-10850h processor specifications. https://www.intel.com/content/www/us/en/products/sku/201897/intel-core-i710850h-processor-12m-cache-up-to-5-10-ghz/specifications.html, 2021.