OPTIMISING IMAGE QUILTING USING AUTO-TUNING

Boyko Borisov, Gergely Kulcsár, Audrey Leong, Oana Roşca

Department of Computer Science
ETH Zurich, Switzerland

ABSTRACT

Image Quilting is a technique for texture generation widely used in the areas of image and video editing. While the early 2000s witnessed a large surge of research into fast texture-generation algorithms, most efforts focused solely on the use of algorithmic optimisations and multi-threading. By contrast, we present a series of hardware-related optimisations to a single-threaded image quilting algorithm, accompanied by an analysis of their effect on performance and runtime, guided by insights from performance analysis tools and extensive raw measurements. Through the use of standard optimization techniques, algorithmic optimisations, code generation, auto-tuning, and AVX-2 intrinsics, we obtain a speedup of over 15x compared to our baseline implementation.

1. INTRODUCTION

Motivation. Image-based texture synthesis algorithms appeared as computer graphics researchers attempted to generate digital content based on (limited) real-world samples. These algorithms take an image as input and synthesise a larger texture which can be perceived to be the same texture as the one represented in the input image.

In particular, image quilting [1] splits the input texture into smaller blocks and stitches them together with some overlap. A minimum error boundary cut is performed on the overlapping area of every pair of adjacent blocks to ensure a more seamless finish.

Improving texture synthesis algorithms to the point of real-time performance is particularly important in the context of texture mapping – the process of applying a texture onto a graphical component in a scene. It enables the expansion of new modelling tools – such as texture paintbrushes – and techniques: for example, dynamically generated visual components can be textured at negligible costs.

Contribution. In this paper we present both algorithmic and hardware-related optimisations of the image quilting algorithm described in [1]. We empirically demonstrate that our optimisations outperform the base implementation compiled at the highest optimisation levels.

We lay the groundwork in Section 2 by presenting the larger context around the algorithm in [1]. This includes its pseudo-code, as well as our derived cost and asymptotic analyses. In Section 3 we describe the software and hardware infrastructure that was used to benchmark the base and optimised implementations. In Section 4 we profile and analyse the performance of the baseline implementation. Then, in Section 5, we propose optimisations based on the observations made in the previous section. We present the results of our optimisations in Section 6 as performance and roofline plots, and we discuss the impact of each optimisation. Lastly, in Section 7 we reflect on the optimisation results and draw conclusions.

Related work. Raad and Galerne discuss some algorithmic improvements to the original image quilting algorithm in their paper [2]. This was achieved through the usage of Fast Fourier Transforms to speed up error calculations and parallelisation to calculate multiple block candidates at once. We did not incorporate these methods as we were discouraged from optimizing algorithms for which the bottleneck are FFT or Gaussian elimination, and we concluded that refraining from this avenue would be more in the spirit of the assignment. We did not experiment with multi-threading related optimisations, as the focus of the course was strictly single threaded applications. Liang and Liu [3] describe optimisations to an algorithm based on the original image quilting algorithm and which suffers from the same performance bottlenecks. However, when picking the next block to be stitched onto the texture, they do not consider the entire search space – like in the original paper – but the set of some approximate nearest neighbours. As we wished to follow the original algorithm as closely as possible, we have not taken this approach into consideration.

2. BACKGROUND ON THE IMAGE QUILTING ALGORITHM

In this section we inspect the image quilting algorithm and its building blocks. First, we describe it briefly and present its pseudo-code. We then assess its cost in floating-point operations, as well as its asymptotic complexity.

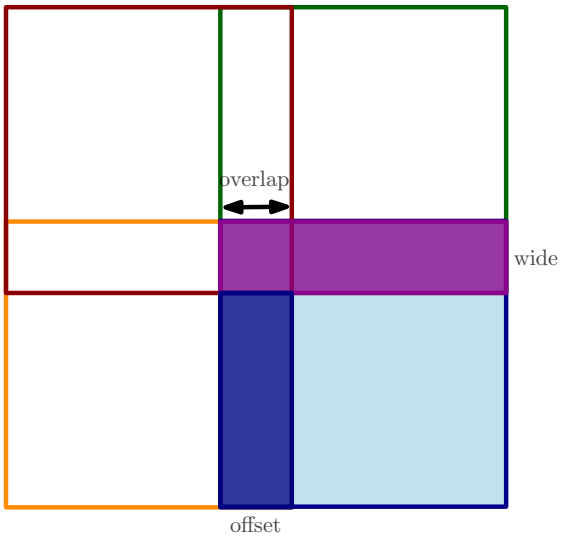

Fig. 1. Image showcasing the various areas of interest when evaluating the fitness of a candidate block (bottom-right)

Quilting using overlapping blocks. As introduced in the previous section, [1] considers *overlapping* blocks when constructing the output image. This is to allow for smoother 'blending' between neighbouring blocks. At every step, a random block is picked from the set of blocks that meet a certain error tolerance threshold. The error is computed over the overlap area (above and left) of the two already-chosen blocks and the current candidate.

We follow the conventions set out in [1]: the width of the overlap area is one sixth of the block size, the error is computed based on the L2 norm between pixels, and the tolerance factor is 0.1 times the lowest nonzero error encountered for a candidate block during a given iteration.

Minimum error boundary cut. Ragged edges between adjacent blocks are introduced to further enhance the consistency of the output texture. Consequently, a minimal cost path can be computed across the overlap area. Let $e_{i,j}$ be the error computed at coordinates i and j on the overlap patch, and $E_{i,j}$ the cumulative minimum error for all paths ending at coordinates i and j. Then:

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

can be used to trace the minimum cut across a vertical overlap. This can similarly be applied to a horizontal overlap patch. In this paper, we use 'tall' to refer to a vertical overlap of a block with its neighbour to the left, and 'wide' to describe the horizontal overlap with the block above. Additionally, 'corner' refers to the overlap between both the tall and wide overlaps.

Algorithm 1 describes the above in pseudo-code. Figure 1 offers a visual representation of the relevant overlap areas between blocks.

Algorithm 1: The image quilting algorithm

Input : source image, size of blocks in source image B, number of blocks per side in output image N

Output: image consisting of $N \times N$ square blocks of size $B \times B$

```
1 for row in 1..N do
2     for column in 1..N do
3         if first row and first column then
4             copy random block to output image
5         else
6             S ← source image blocks with overlap
                error meeting threshold
7             b ← random element of S
8             compute minimum error path on b's
                overlaps with above and left
                neighbours
9             copy b to output image based on
                minimum error path
```

Cost Analysis. We define the cost measure of the algorithm as the total number of floating-point operations when working with floats, and as the total number of integer operations – unrelated to indexing – for the integer implementations of the application [1]. This number is computed by inspecting the implementation and inserting counters where floating-point operations take place, in both the baseline and optimised implementations. The counters can be enabled at compile-time using a macro. Note that our cost analysis assigns a cost of 1 flop to all floating-point operations (ADD, MUL, DIV).

Asymptotic Analysis. Assume the following notation:

W	width of input image
H	height of input image
B	block size of blocks in input image
O	block overlap size
N	number of blocks in output image, per side

The loops on lines 1 and 2 of Algorithm 1 have an asymptotic complexity of $O(N^2)$. The other computationally heavy part of the implementation involves constructing the set of blocks with small overlap errors, described on line 6. The overlap error must be computed for every block in the input image – that is $(H - B) \cdot (W - B)$ blocks. The error for each block is computed across a surface of size $O \cdot B$. Ignoring the asymptotically lower-order terms, the expected

[1] For brevity, we will call the cost measure 'flops' even for integer implementations. In this case we count integer operations in reality.

asymptotic runtime of the algorithm is defined as:

$$O(N^2 \cdot (H - B) \cdot (W - B) \cdot O \cdot B).$$

Assuming $H = W$ and $B \ll H$, the above simplifies to:

$$O(N^2 \cdot H^2 \cdot O \cdot B).$$

3. EXPERIMENTAL SETUP

In this section we present the infrastructure setup that was put into place to perform experiments and collect results.

Platform setup. All experiments were performed using the following hardware, compilers and combinations of compiler flags:

- CPU: Intel Core i7-6700HQ (Skylake)
- Caches and memory:
  - L1 (per core): 32KiB, 8-way set associative, 64B line size, throughput of 24 floats/cycle
  - L2 (unified): 256KiB, 4-way set associative, 64B line size, throughput of 16 floats/cycle
  - L3 (shared across all cores): up to 2MiB per core, up to 16-way set associative, 64B line size, throughput of 8 floats/cycle
  - Memory throughput: 4 floats/cycle
- Compiler: MinGW/GCC 12.2
- Compiler flags: *-O3*, *-ffast-math*, *-march=native*, *-f(no)-tree-vectorize*

Compiler choice. In addition to GCC, we have also considered the Intel oneAPI DPC++/C++ compiler with flags *-O3*, *-fp-model=fast*, *-xHost*, and *-(no)-vec*, which mirror the flags chosen for the GCC compiler. As Figure 2 shows, the two compilers perform similarly on the baseline implementation. We have eventually decided against using the Intel oneAPI compiler as our default compiler given the results of the performance comparison experiment. This is in addition to the significantly higher overhead of porting this compiler across the heterogeneous hardware we have used for development (in particular, this compiler is not available on macOS).

Timing infrastucture. Both the RDTSC and the C clock timers were integrated into the implementation. A macro was added to allow for the timer to be selected at compile time. The RDTSC timer was eventually used for the final measurements because of its lower variance from run to run. Only the operations performed by the quilting algorithm were timed. The operations related to memory allocation/deallocation and image reading/writing were excluded from the measurements.

The reported time for each input size is the average across 15 runs of the program. To ensure cold-cache measurements, matrix multiplication was performed between consecutive runs of the algorithm. The matrices were sufficient in size to clear all three cache levels.

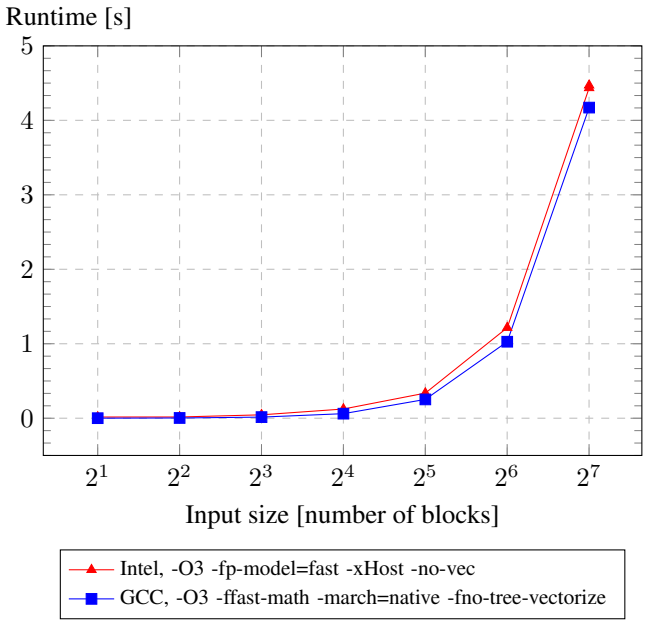

Fig. 2. Performance comparison between the GNU GCC compiler and the Intel oneAPI compiler, on the baseline implementation, using equivalent flags

Validation infrastructure. This was created to ensure no logical changes to the algorithm occurred while optimising it. A flag set during compilation may be used to switch off nondeterminism, thus make resulting images comparable with one another.

Resulting images are compared using mean squared error over the resulting pixels and channels. Validation fails if the error exceeds a small, handpicked threshold. We note that this validation step makes numerical stability quite important, as the imprecise calculation of errors may lead to branching into a different choice of block, which would have cascading effects across the entire output image. The authors, however, do not see a way to circumvent this issue, so care was taken to make sure that the optimisations implemented did not have a significant impact on numerical precision.

4. ANALYSIS OF THE BASELINE IMPLEMENTATION

In this section, we describe the implementation of the algorithm we assume as the baseline, the profiling tools used to

detect performance bottlenecks, and the observations concerning areas for performance improvement.

Baseline implementation. Our baseline implementation is largely a translation of the Matlab implementation provided in the project description [4] to C. The `stb_image` [5] library was used to handle image reading and writing.

The baseline implementation makes some simplifying assumptions regarding the input parameters. Namely, the input image size is fixed to 40×40 pixels and the block size is fixed to 24×24 pixels. This limits the input parameters' degrees of freedom to the number of blocks only, making benchmarking more straight-forward. We fixed the parameters to these particular values for two reasons:

1. We persist flexibility in terms of memory footprint. At $N \leftarrow 2^2$ the application's memory footprint fits in L1 cache. At $N \leftarrow 2^7$ the memory required by the program reaches roughly 1.2 GB.
2. The block and overlap size being multiples of 4 greatly reduces the need for additional operation for handling leftovers in the AVX-2 intrinsics implementations.

Performance data for the baseline is collected across various input sizes. As explained in the paragraph above, the block size is fixed. This leaves N, the number of blocks in the output image per side, as the only parameter which can be varied (as per Algorithm 1). We vary N between 2 and 128, in powers of 2. This is to ensure that several scenarios are captured, particularly related to the working set overflowing from different cache levels.

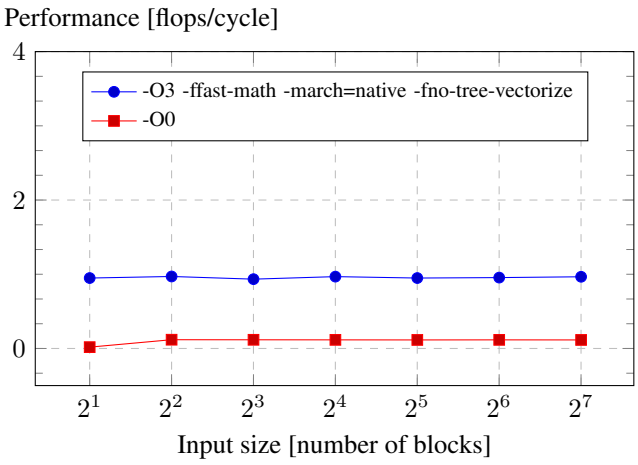

Fig. 3. Performance of the baseline implementation measured on Intel i7-6700HQ and compiled with GCC-12. Results averaged across 15 runs, cold cache

Profiling Approach. Intel VTune was used to analyse the baseline's performance and to obtain a more fine-grained breakdown of the time spent in the different sections of the code. Intel Advisor was used to produce the roofline analyses. It is important to note that the performance reported for Intel Advisor is for the entirety for `calc_errors` only, while the one in the performance plot is for all steps of the image quilting algorithm, which include image reading and writing. We have derived a number of observations based on the performance of the baseline implementation:

Observation 1. The vast majority of the running time is spent in the section of the code which is responsible for computing the errors on the overlapping patch between two blocks – `calc_errors` in the source code. The baseline implementation spends more than 99% of the running time in this function.

Observation 2. Both the input and output images are represented using three-dimensional arrays, which leads to unnecessary dereferencing. VTune measurements indicated that a significant portion of time was spent in these dereferencing instructions.

Observation 3. `calc_errors` inherently possesses good spatial and temporal locality. The function has spatial locality because all data structures are loaded in row-major order. Temporal locality comes from the fact that the entire input image, which fits in L1 cache with our specified block size, is read on every iteration.

Observation 4. The application's performance (Figure 3) does not significantly fluctuate based on the input size. This, together with the roofline measurements (Figure 8), indicate compute-boundness.

Observation 5. The baseline optimisation uses a single accumulator, which severely limits instruction-level parallelism.

Observation 6. 32-bit integers can be used for all computations without losing precision. Some of the operations even allow for the usage of 16-bit integers without the risk of overflow.

Observation 7. `calc_errors` is strongly suitable for vectorisation, because of the same computations being applied to the entire input.

5. PROPOSED OPTIMISATIONS

In this section we outline our optimisations and explain the motivation behind each one of them.

All of our optimisations focus on improving the error calculation section of the code. This is because a vast majority (over 99%) of the running time of the application was spent in this function in the baseline implementation, according to our profiling results presented in the previous section. This part of the algorithm continues to dominate the performance measurements in the optimised implementations.

As the profiling results indicate that the baseline is compute-bound, we focus our efforts on improving the overall Instruction-

Level Parallelism (ILP) of the implementation, rather than addressing more memory-centric possible issues.

Optimisation 1: Omission of redundant error calculation for corners. The baseline involves calculating the error for the corner separately and subtracting it from both the tall and wide overlapping rectangular regions. To avoid these redundant calculations, we have decided to offset the indices of the tall overlap, and only take the error calculation of the wide overlap into consideration for the corner. This ensures that the wide overlap is divisible by 8, which facilitates future vectorisation.

Optimisation 2: Flattening all data structures. To reduce the amount of de-referencing done, we decided to flatten the three-dimensional arrays to a one-dimensional representation of the input and output images. This has allowed us to further experiment with locality by changing the order in which the data is stored and the loops are ordered. We have chosen to test having channel as the outermost index, which provides us with more fine-grained control of the factors by which we unroll the entirety of `calc_errors`. For example, it allows us to unroll by 4 or 8 for better pipelining when executing parallel instructions. This is in contrast to having channel as the innermost index, which always gets fully unrolled, removing the overhead of an entire loop, but restricting unroll factors to multiples of 3.

Optimisation 3: Loop unrolling and usage of multiple accumulators. Since Observation 5 informs us of the baseline implementation's limited use of instruction-level parallelism, we have written a generator that unrolls and accumulates the error calculation by various factors. Unrolling is done for the row, column, and channel of the overlap areas. The loop order and data representation vary between having channel as the first or last index, allowing for experimentation with spatial and temporal locality. However, given that the overlap areas wide, tall, and tall-with-an-offset (from Optimisation 1) all have different dimensions, we have split `calc_errors` into three functions to accommodate each shape's separate unrolling factors. Because of this, as well as the changing loop order, we have also ensured that we can unroll by non-divisors of the loop lengths. This additional functionality gives us further flexibility in what factors we can unroll by and also enables the generation of fully unrolled versions of the code, in which instructions are grouped in blocks of desired size in Single Static Assignment style.

Optimisation 4: Usage of 32-bit integers. We ported the generator and loop unrolling functionality to a version of the code that represented data as 32-bit integers. This was not expected to perform better than the float version. It was done to enable future experimentation with integers of smaller size, which would enable further parallelism.

Optimisation 5: Usage of AVX2 vector intrinsics. Based on Observation 7, we expanded the existent generators to support vector intrinsics. In this optimisation step, we developed four separate generators, where each supports a single data type (single-precision float or 32-bit integer) and one flattened matrix representation type. All four generators support specifying the number of accumulators used, which in turn determines the unrolling factor. Using vector intrinsics came with the overhead of having to apply some additional operations related to data alignment to ensure that the entirety of the 256-bit registers was utilised.

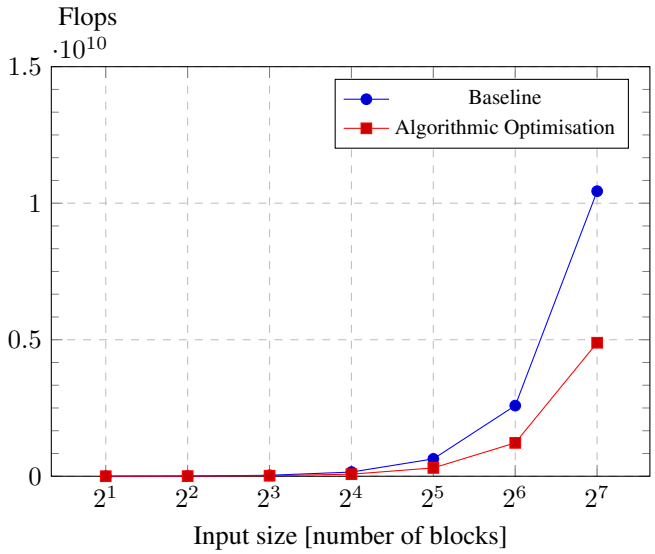

Fig. 4. Flop Count Analysis

Optimisation 6: Usage of 16-bit integers. According to Observation 6, we experimented with using 16-bit integer representations of the input and output image and performing the loading, subtraction, and multiplication. The error accumulation occurs using 32-bit integers to prevent overflows. The AVX-2 instruction set is well suited for obtaining a 32-bit product of 16-bit integers. This optimisation was interesting to explore because of introducing the trade-off between having higher bandwidth for part of the calculations and having the additional overhead of performing more operations to obtain a 32-bit product from 16-bit multiples.

6. EXPERIMENTAL RESULTS

In this section, we evaluate the performance of the baseline and optimised versions of the algorithm. We also compare the performance impact of the different optimisations.

Experiment 1: Exploring the speedup and reduction of the number of floating point operations with Optimisation 1. This algorithmic optimisation reduces the flop count by half for the larger input sizes, and at its peak introduces a speedup of 2.13x (Figure 5).

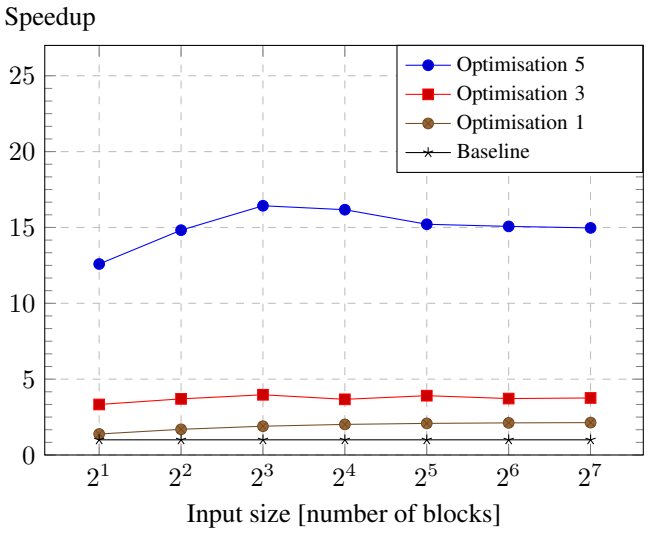

Fig. 5. Speedup plot. The reported speedup is for the best-performing parameters of the specified stage of optimisation. Measured on Intel i7-6700HQ and compiled with GCC-12. Results averaged across 15 runs, cold cache. Flags: -O3 -ffast-math -march=native -fno-tree-vectorize

Implementation	ψ^r_{wide}	ψ^c_{wide}	ψ^r_{tall}	ψ^c_{tall}	ψ^r_{offset}	ψ^c_{offset}	$\frac{\text{Runtime}}{N^2}$
Float, Ch. last	1	6	2	3	1	6	207779
Float, Ch. first	1	4	2	3	1	4	157363
32-Int, Ch. last	1	6	2	12	2	3	178111
32-Int, Ch. first	1	4	8	1	8	1	262032

Table 1. Generation parameters and normalised runtimes in cycles for the best-performing versions of each **scalar** implementation. The number of blocks N is 32; Measured on Intel i7-6700HQ and compiled with GCC-12; Results averaged across 15 runs, cold cache; Flags: -O3 -ffast-math -march=native -fno-tree-vectorize

Experiment 2: Effect of Optimisation 2. This optimisation in and of itself does not lead to an improvement in performance; it remains at 1 flop/cycle regardless of the input size or the flattened loop order, which is equivalent to the baseline's behavior. This is due to inter-loop dependencies being the bottleneck, despite VTune noting that much of the time spent in `calc_errors` was on de-referencing the original matrix representation, which we attribute to the imprecise nature of instruction-level measurements.

Experiment 3: Finding the best hyperparameters for the scalar generators. The main advantage of using generators as opposed to manually writing each optimisation is the opportunity to find the best unrolling factors for the different functions through auto-tuning. The generator accepts 2 parameters per `calc_errors` shape:

- ψ^r_{shape}: The number of rows processed in a single iteration.
- ψ^r_{shape}: The number of accumulators operating in a single row.

The optimal parameters for each generator were found by fixing the parameters for two of the `calc_error` shapes and iterating through a range of possible optimal parameters for the third. Table 1 presents the optimal parameters for each generator and the achieved normalised runtime at $N = 32$. The experimentation was extensive with more than 20 different sets of parameters tested for each generator[2]. A trend we noticed is that the optimal integer generator configurations make use of more accumulators than their floating-point counterparts. We speculate that this may be related to the lower bandwidth of scalar multiplication for integers (the throughput of the operation on the target machine is 1 per cycle). Having more accumulators implies that there are more operations that can be performed in the background while the multiplications are pipelined.

Experiment 4: Comparison of the best performing scalar generator configurations. The optimal unrolling factors yielded by auto-tuning were surprising at first. However, they can be explained by one port being responsible for the FMAs and another for the subtractions, with both being fully occupied in the case of an unrolling factor of 4. This leads to both ports being able to issue instructions every cycle: one of them can issue subtraction commands, while the other one can accumulate these with FMAs delayed by 4 cycles. This gives minimal overhead due to the number of accumulators, since both ports can not be kept busy with less accumulators.

It can be observed that channel-first outperforms channel-last in the case of floats, with this trend being reversed for integers. The significantly worse performance for integers can be explained by the lack of FMA operations for integers on the processor in question. We further hypothesise that for floats, channel-first performs better due to the more fine-grained control when unrolling. A plausible explanation for the reversal of the situation when employing integers is that, due to multiplication being the bottleneck – with only one port being able to issue integer multiplication operations – the overhead of the additional loop significantly impacts the measured performance.

The highest speedup compared to the baseline is achieved by the channel-first implementation with floats. The speedup of this implementation is in the range between 3x and 4x (Figure 5).

Experiment 5: Finding the best hyperparameters for the AVX-2 generators. We approached the optimisation of

[2] The results of the experiments (for both scalar and vectorised implementations) can be found in the project's GitLab repository in `scripts/generator.ipynb` in the branch for the respective optimisation.

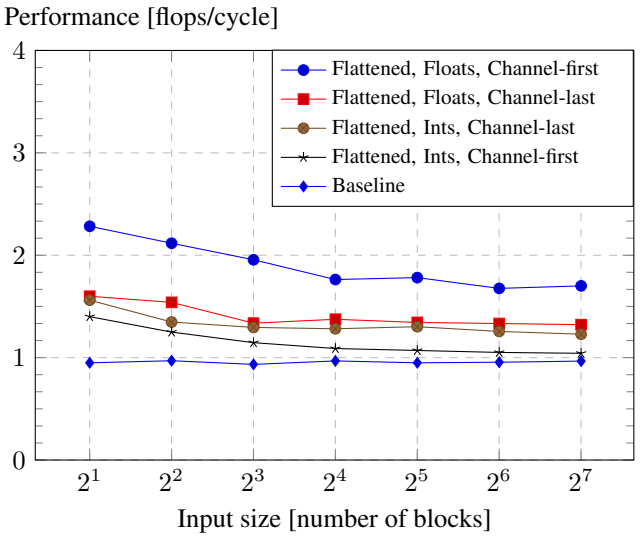

Fig. 6. Performance of implementations using scalar operations. Measured on Intel i7-6700HQ and compiled with GCC-12. Results averaged across 15 runs, cold cache. Flags: -O3 -ffast-math -march=native -fno-tree-vectorize

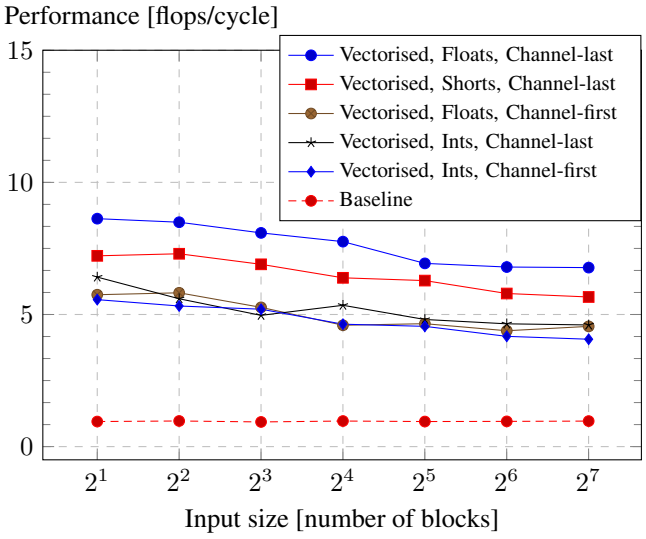

Fig. 7. Performance of implementations with AVX-2 intrinsics. Measured on Intel i7-6700HQ and compiled with GCC-12. Results averaged across 15 runs, cold cache. Flags: -O3 -ffast-math -march=native -fno-tree-vectorize

Implementation	ψ_{wide}^{r}	ψ_{wide}^{c}	ψ_{tall}	ψ_{offset}	$\frac{\text{Runtime}}{N^2}$
Float, Ch. last	1	3	12	6	40431
Float, Ch. first	2	1	4	2	60263
32-Int, Ch. last	1	3	6	6	58277
32-Int, Ch. first	1	1	4	2	61605

Table 2. Generation parameters and normalised runtimes in cycles for the best-performing versions of each **vectorised** implementation. The number of blocks N is 32; Measured on Intel i7-6700HQ and compiled with GCC-12. Results averaged across 15 runs, cold cache. Flags: -O3 -ffast-math -march=native -fno-tree-vectorize

our implementation with AVX-2 intrinsics by using generators once again. The generators accept 4 parameters:

- ψ_{wide}^{r}: The number of rows processed in a single iteration of `calc_errors_wide`.
- ψ_{wide}^{c}: The number of accumulator registers operating on one row when calculating `calc_errors_wide`.
- ψ_{tall}: The number of accumulator registers used in `calc_errors_tall`.
- ψ_{offset}: The number of accumulator registers used in `calc_errors_tall_offset`.

It is important to note that one accumulator register contains 8 scalar accumulators. The optimal parameters for each generator were found with the same approach as before: experimenting with only one `calc_error` shape at a time. Table 2 presents the optimal parameters for each generator and the achieved normalised runtime at $N = 32$. We have noticed that employing too many accumulators hinders performance. This can, at least partially, be explained by the overhead of summing up all of the values within each accumulator to a single scalar.

Experiment 6: Comparison of the AVX-2 intrinsic implementations. Figure 7 compares the performance of the different implementations of the algorithm with AVX-2 intrinsics. It is apparent that all methods lead to a significant performance increase compared to the baseline implementation. The best performing implementation (vectorised, floats, channel-last) has a performance ranging between 6.7 flops/cycle and 8.6 flops/cycle (Figure 7) and a peak speedup of 16.4x (Figure 5).

An observation we made is that the channel-last implementations generally outperform the channel-first implementations for the same data type. This can be explained by the fact that channel-last implementations benefit more from unrolling. In addition, there is less overhead when performing additional intrinsic operations related to aligning the data.

The 32-bit integer implementations perform the worst, while the 32-bit floating point implementations perform the

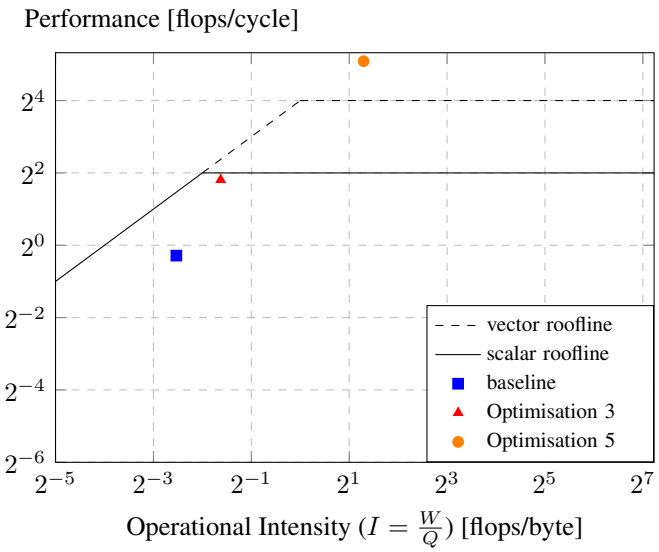

Fig. 8. Roofline plot for the best-performing version of the application from the specified stages of optimisation. The data was collected with Intel Advisor. The operational intensity and performance is for `calc_errors` only. The number of blocks N is 32. Measured on Intel i7-6700HQ and compiled with GCC-12. Results averaged across 15 runs, cold cache. Flags: -O3 -ffast-math -march=native -fno-tree-vectorize

best. The 16-bit integer implementation is the middle ground. The float implementations' performance can be explained by having FMAs available. The 16-bit integer implementation performs better than the 32-bit integer implementation thanks to the doubling of the bandwidth for loads, subtractions and multiplications. The performance gain from this improvement, however, is limited by the fact that the 16-bit integers have to be transformed into 32-bit integers for the error accumulation. This transformation leads to a reduction in the bandwidth available for that final operation.

Roofline Analysis: Discrepancies and hypotheses. In the roofline plot presented in Figure 8, it can be observed that the performance and operational intensity both improve at the different stages of optimisation. The increase in operational intensity can be explained by the improvement in locality attributed to the flattening of the data structures. The vectorised implementation further benefits from using each loaded cache block only once in the innermost loop. The increase in performance can be explained by the increase of instruction-level parallelism and the use of FMAs.

It is important to highlight that Intel Advisor exhibits some inaccuracy in its measurements, as demonstrated by the performance of the vectorised version of the code exceeding the theoretical maximum, despite Turbo Boost being disabled on the experiment machine. In spite of this, we can still speculate that the performance of `calc_errors` is likely close to the optimal. Another factor contributing to the performance measurement discrepancy between Intel Advisor and the measurement presented in Figure 7 is that Intel Advisor only measures the performance of `calc_errors`, as opposed to that of the entire application. This implies that if we would like to increase performance further, we might like to start focusing on sections of the code outside of `calc_errors`. This approach, however, might not lead to a significant runtime gain even for the most optimised version of the code, as at least 80% of the running time is spent in `calc_errors`.

7. CONCLUSIONS

In this paper, we examined different algorithmic and hardware-informed approaches to optimise the image quilting algorithm presented in [1] through extensive measurement and experimentation. Throughout the project we used standard hardware-related optimisation techniques, algorithmic optimisations and vector intrinsics to optimise the code for our machine.

Our improvements were achieved in an empirically exhaustive manner by developing a suite of code generators. This has allowed us to experiment with different data representations, data types, and unrolling factors. This extensive empirical approach not only allowed us to identify suitable optimisations and the optimal parameters for their applications, but also to discover unsuitable ones – such as the usage of integers. Eventually, we have achieved a 15x speedup compared to the baseline implementation.

8. CONTRIBUTIONS OF TEAM MEMBERS

Boyko. Implemented the baseline implementation without the minimum cut with Oana. Extended the generator to work with scalar integers. Implemented generators for floating point and integer vector intrinsics. Made performance measurements for all generator implementations. Investigated using the Intel Compiler with Oana.

Gergely. Implemented minimum cut part of baseline and implemented algorithmic optimisations with Audrey. Extended the generator to add additional functionality for unrolling by non-divisors and grouping. Added minor optimizations to accumulator usage in generated code. Analyzed code with VTune and Intel Advisor with Audrey to guide optimization efforts.

Audrey. Implemented minimum cut part of baseline and algorithmic optimisations with Gergely. Implemented all scalar versions of the generators, accommodating for both loop orders, and separating the function for each overlap shape. Integrated the generators into Boyko's measurement infrastructure with Gergely to facilitate autotuning.

Worked with Gergely to retrieve VTune measurements and Intel Advisor rooflines through each optimisation phase.

Oana. Implemented baseline implementation without minimum cut with Boyko. Focused on flattening the data structures and integrating the new data representation format with the omission of the redundant error calculation for corners. Performed measurements for vector intrinsics generators with Boyko. Investigated using the Intel Compiler with Boyko.

9. REFERENCES

[1] Alexei A. Efros and William T. Freeman, "Image Quilting for Texture Synthesis and Transfer," Association for Computing Machinery, 2001, https://doi.org/10.1145/383259.383296.

[2] Lara Raad and Bruno Galerne, "Efros and Freeman Image Quilting Algorithm for Texture Synthesis," *Image Processing On Line*, vol. 7, pp. 1–22, 2017, https://doi.org/10.5201/ipol.2017.171.

[3] Lin Liang, Ce Liu, Ying-Qing Xu, Baining Guo, and Heung-Yeung Shum, "Real-time texture synthesis by patch-based sampling," vol. 20, no. 3, 2001, https://doi.org/10.1145/501786.501787.

[4] Jordan Mecom and Alice Wang, "image-quilting," GitHub, 2016, https://github.com/jmecom/image-quilting.

[5] Sean Barrett, "stb: Single file libraries for c/c++," GitHub, 2014, https://github.com/nothings/stb.