FAST EXPOSURE FUSION

Jérémie Miserez

Department of Computer Science
ETH Zürich
Zürich, Switzerland

ABSTRACT

In this project, the exposure fusion algorithm [1] was partially optimized. The optimized part for generating the weight maps shows increased scalar performance (in flops/cycle) of 1.27 times the baseline value. Using AVX vectorization, this could be increased to 4.0 times the baseline value. The resulting performance for the whole algorithm (including unoptimized parts) is then compared to another similar implementation [2]. The results are very similar to the best intermediate result in their implementation, which contains only the same optimized parts of the code.

1. INTRODUCTION

Motivation.

When capturing photos using a standard digital camera, the dynamic range captured in a single image is often not sufficient to encode both very dark areas as well as bright areas with the same detail. This is most often the case when taking photos against the sun, or in dark rooms with bright windows. It is however often possible to take multiple pictures from the same angle, but with different exposures. Then, using exposure fusion these images can be combined to a single image combining each of the regions of each image that are correctly exposed.

In this report, I will show that the exposure fusion algorithm can be sped up as a whole (in terms of cycles) by removing redundant operations, and that individual parts of the algorithm can be optimized for performance (in terms of floating point operations per cycle). As time was limited, just the worst performing parts of the algorithm were optimized in this fashion.

Related work. Mertens et al. [1] published a novel way to automatically generate properly exposed composite pictures from multiple source images. The set of input images must all be taken from the same perspective, but the individual exposure levels of the images can be different. The exposure fusion algorithm then weights the pixels in the images according to a quality measures. The images are then blended according to these weights, the result is a well-exposed picture. The authors aim to prevent halos and other image artifacts by doing multiresolution blending instead of blending the whole image at once. What this means in practice is that there are no unnatural sharp edges, as the low-frequency parts of the image are blended first, with successive refinement in blending smaller and smaller details. This ensures that there are no high-frequency artifacts in the resulting output image. HDR (high dynamic range) is similar, although there are strong differences: While exposure fusion aims to create a single image with normal dynamic range that is viewable on standard monitor, HDR primarily creates single images with a higher dynamic range than is displayable on a standard monitor. In order to actually view an HDR image, it must then be tone-mapped and converted into an image suitable for viewing on a standard monitor. This step is not necessary with exposure fusion.

A group last year [2] implemented the same project, and there will be comparisons to their works in later sections.

2. BACKGROUND

Per pixel quality measures. The exposure fusion algorithm measures the quality of a pixel in an input image in three different ways:

- Saturation: In this implementation, the saturation is defined as the standard deviation within the three color channels red, green, and blue. This is a per pixel measure.
- Well-exposedness: The well-exposedness defined as the distance from the a perfectly exposed pixel with a grey value of 0.5. The distance is weighted by the form of an S curve, thus very light and very dark pixels are rejected. This is also a per pixel measure.
- Contrast: This is defined as the absolute response of a Laplacian filter in a 3x3 neighborhood of the pixel in question. This weights pixels with high local contrast higher than pixels in other areas.

Laplacian 3x3 filter. In contrast to the other filters applied during later steps, the Laplacian 3x3 kernel is not lin-

early separable. Thus, it is necessary to actually look at the neighbors of each pixel in question while processing it.

Gaussian pyramids. A Gaussian pyramid is a stack of images created from a single input image. Each image further down in the stack has half the width and height of the previous one. Downscaled images are generated by filtering the image with a 5x5 Gaussian filter, and then storing the values of each 2nd pixel value in each dimension. The original input image is always preserved as the topmost image. A Gaussian pyramid is created from each weight map, one for each input image.

Laplacian pyramid. A Laplacian pyramid is similar to a Gaussian pyramid. However, after the downscaling step the image is upscaled again, and only the difference between the upscaled and current image is stored in each layer. Thus, the original image is not stored in any one layer, but must be reconstructed by reversing the process: Upscaling the smallest image and adding the stored differences at each level. One Gaussian pyramid is created for each input image, and an additional empty pyramid as a template for the output image.

Upscaling and Downscaling. Downscaling is analogous to the downscaling step in the Gaussian pyramid, while upscaling involves upsampling the smaller image by a factor of 2 and then filtering with a 5x5 Gaussian filter. Afterwards, the stored difference is added. Both upscaling and downscaling operations are linearly separable, thus they can be done in two passes: A per-row pass followed by a per-column pass.

Multiresolution blending. The last step involves blending all input images together. This happens at each resolution step and blends a layer from each Laplacian pyramid weighted by the weight map stored in the corresponding layers at each Gaussian pyramid. The end result is a single Laplacian pyramid, which can then be reconstructed to produce a single output image.

Cost analysis. The cost analysis is done by measuring the exact number of operations executed. A macro is inserted into the source code after every operation, counting the number of executions for each type of floating point op as well as loads and stores. An analysis of the algorithm and runtime plots shows that the algorithm runs in $O(w*h*n)$, where n is the number of input images and w and h the dimensions of these images. This gives the following cost measure:

$$C_{fusion}(w,h,n) = \begin{pmatrix} 75*w*h*n \text{ loads} \\ 26*w*h*n \text{ stores} \\ 54*w*h*n \text{ adds} \\ 57*w*h*n \text{ mults} \\ 3*w*h*n \text{ divs} \\ 3*w*h*n \text{ abs} \\ 1*w*h*n \text{ sqrts} \\ 3*w*h*n \text{ exp} \end{pmatrix}$$

And just for the weight map generation that would be:

$$C_{weightmap}(w,h,n) = \begin{pmatrix} 15*w*h*n \text{ loads} \\ 5*w*h*n \text{ stores} \\ 19*w*h*n \text{ adds} \\ 17*w*h*n \text{ mults} \\ 3*w*h*n \text{ divs} \\ 3*w*h*n \text{ abs} \\ 1*w*h*n \text{ sqrts} \\ 3*w*h*n \text{ exp} \end{pmatrix}$$

For all performance calculations, the exponential function is weighted as 29 flops. This is the exact amount of flops it uses in the freely available (zlib license) sse_math library by Julien Pommier [3]. This library was used solely for its exp() function in SSE, as native SSE or AVX does not support it.

The authors in [2] weighted the exponential function as 50 flops, the abs() function as 2 flops, and the sqrt() function as 5 flops.

3. IMPLEMENTATIONS

An initial reference implementation was created that was 1:1 identical to the Matlab implementation provided by the original authors of the paper.

Baseline: Minimizing the number of operations. The first step was to remove all operations that produce an intermediate result that is never read again. Also, intermediate storage variables were removed when possible and operations were fused. Most of these reductions were done in the area of pyramid generation and reconstruction, as they contained the most possibilities for this kind of optimization. The most prominent example is the Gaussian filtering applied before downsampling and after upsampling. Implementing the filtering together with the downsampling allows us to reduce the number of pixels calculated by a factor of 4 for downscaling, and by a factor of 2 for upscaling. The original algorithm allows the user to weight the importance of each of the pixel quality measures relative to each other. The choice of the weights has no effect on the runtime of the algorithm, but the weighting itself needs floating point

pow() functionality. As it was foreseen that this functionality is not available in AVX, the decision was made to hardcode the importance of each measure as equal. All these reductions in together reduced the total number of flops by a factor of 2.03.

Weightmap: Two-pass approach. The 3x3 Laplace filter operates on the greyscale versions of the input images. In the baseline approach, the image data is read as RGB values and then converted on the fly every time a pixel is read from memory. An initial idea was to implement a two-pass approach: In the first pass, all greyscale values are calculated and stored. In this step, the saturation and well-exposedness measures are also stored. In the second pass, the 3x3 convolution for determining the contrast values is then run on the stored greyscale values. With no other optimizations, this reduces the number of loads and stores from 15 loads and 1 store per pixel to 9 loads and 3 stores per pixel.

Weight map: All-in-one approach. The next approach was to calculate all the measures at the same time. This removes the need for an intermediate array to store the greyscale values. For each pixel we calculate the saturation, well-exposedness and the local contrast. The idea is that it is not necessary to store the greyscale values in a separate array if we can later introduce blocking and store the greyscale values that will be reused directly in registers.

Weight map: Blocking and Inlining. The next step was then to introduce blocking for caches, using a block size of 64x64. This block size ensures that the block will fit into the L2 cache: At 256KiB, the L2 cache can hold 104x104 RGB pixels. Inside each block, additional inlining or blocking was used: Three different versions were implemented with block sizes of 1x1 (no inlining), 2x1, 2x2, and 2x4. The inlined versions are able to reuse the greyscale values inside a block, by storing them in registers.

Weight map: AVX versions. AVX versions were implemented for three of the previous versions. The simplest AVX implementations are for the all-in-one and the blocking without inlining version: These versions simply uses AVX for operations that work on each of the channels R,G, and B, thus effectively wasting a fourth of all possible operations. They also use AVX to sum up the values of the 4 neighbors, by simultaneously adding all four neighbors horizontally. Thus they still process only 1 pixel per iteration. In order to make real use of the AVX instructions, the AVX implementation for the blocked and inlined 2x4 version is more involved: This version calculates a total of 8 pixel values per iteration (4 values per row), and does not waste any operations. The values are shuffled so that all operations for each of the 4 output pixels can be done in parallel. In the next iteration, 4 out of 12 greyscale values can be reused. All in all, per iteration there are 48 loads and 8 stores, which translates to 6 double loads per output pixel.

As AVX does not have support for the exp() function, sse_math library [3] was used here.

Downscaling/Upscaling. Both the downscaling as well as the upscaling functions could profit from blocking (in order to improve spatial locality) and an AVX implementation. However, this is currently not yet done.

4. EXPERIMENTAL RESULTS

Experimental setup. All experiments were run on a Intel i7-2720QM with 256Kb of L2 cache per core and 6MB of L3 cache. GCC 4.8.1 was used, with the relevant flags "-O3 -m64 -march=corei7-avx". All versions were run for input sizes 128x128 through 4096x4096, always with a set of 4 differently exposed photographs of the same dimension. Using C macros, each version was instrumented to count the exact number of loads, stores, and flops in the run. Then it was recompiled without the macros, and the runtime (cycles) as well as cache miss percentages were measured using the libpfm4 library after a short warmup phase.

Results only for weight map generation. Figure 1 shows the performance when only measuring the weight map generation. The best version "inline2x4_avx" has an exact speedup of 4.0 over the baseline version, which is to be expected due to vectorization. However, the best scalar version "blocking" is only 1.27 times faster than the baseline. The measly performance of the other two avx versions "onestep_avx" and "blocking_avx" is due to the incomplete use of the AVX registers by using only 3 out of 4 values. It is then only a speedup of 3.07 compared to the baseline version. Figure 2 shows the operational intensity when only measuring the weight map generation. Considering that other parts of the code have an operational intensity of below 0.25, the high values of between 0.812 for the baseline up to 1.674 for the "inline2x2" versions suggest that the weight map generation is compute-bound, rather than memory-bound. A large contributor is the exp() function, as it consists of 29 different floating point operations.

Results for overall program. Figure 3 shows the performance for the full algorithm, including any unoptimized code such as the pyramid generation code. The best version is 1.67 faster than the baseline implementation. These results can be directly compared to the results from [2]: Although the final achieved performance of the full algorithm is not the same, the best AVX version from this implementation has almost exactly the same performance as their last SSE intermediate result. This makes sense, as the authors took the same approach as this implementation: They first optimized the weight map generation completely (including SSE) and only then went on the optimize the pyramid code as well. The plot in the paper [2] uses the same image sizes.

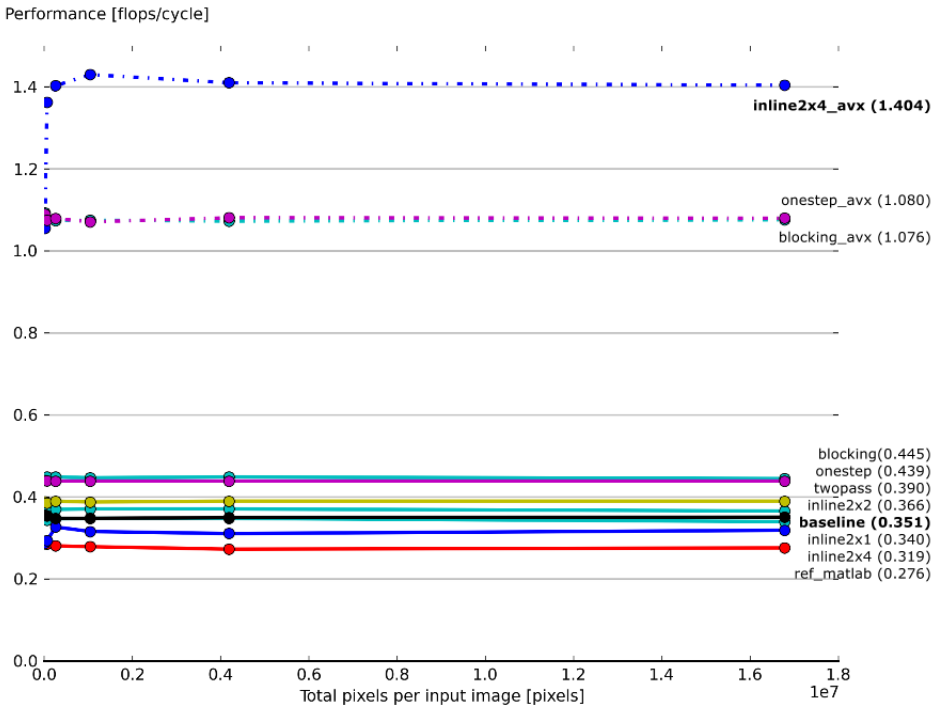

Fig. 1. Performance of the weights() function. Dashed lines are AVX.

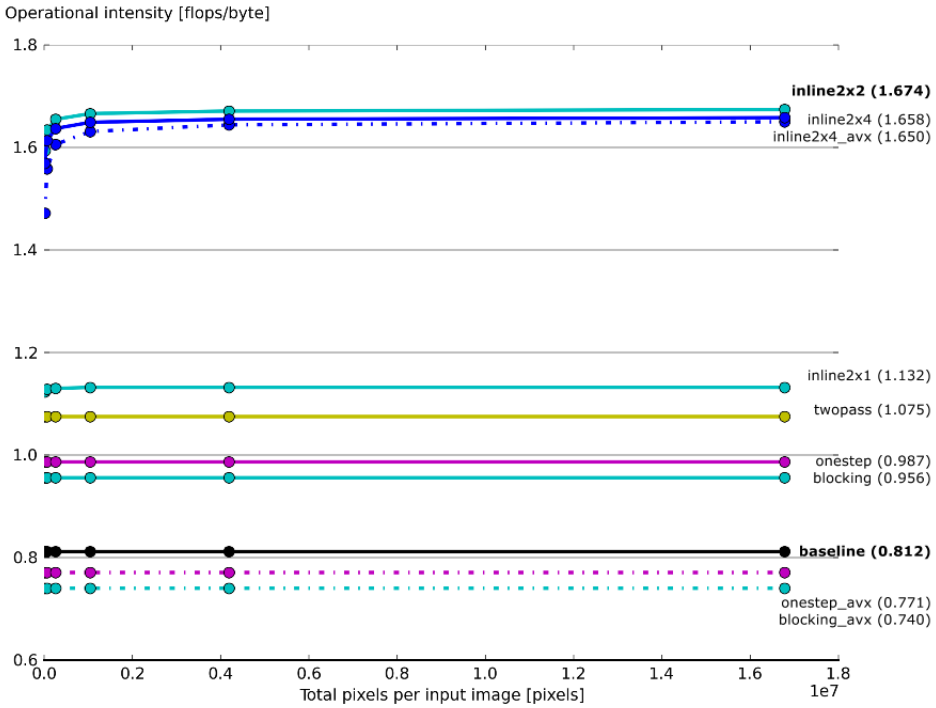

Fig. 2. Operational intensity of weights() function. Dashed lines are AVX.

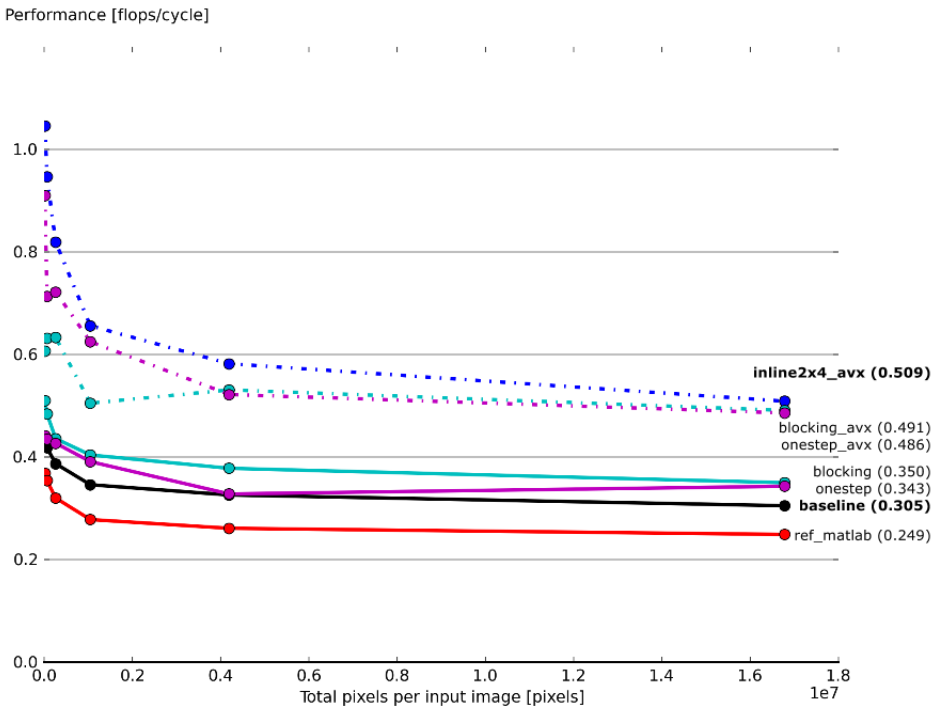

Fig. 3. Performance of the full algorithm, including unoptimized pyramid function. Dashed lines are versions with AVX for the weights() function.

5. CONCLUSIONS

Main results. In this project, the exposure fusion algorithm was implemented and partially optimized. The weight map generation was sped up and the scalar performance was increased by a factor of 1.27 from the baseline. The vectorized version has a performance 4.0 times higher than the baseline. **Further improvements.** In hindsight, the two-pass approach would probably have been a better candidate for blocking/vectorization compared to the all-in-one approach, as there would have been no need to recompute the greyscale values each time a pixel was read. Even though recomputing greyscale values may increase the performance (in terms of flops/cycles), possibly more could be gained from the reduction of overall runtime (cycles), memory accesses, and cache misses. Unfortunately, most of the work had already done for the all-in-one approach and changing all of the implementations was not feasible anymore. In addition, the downscaling and upscaling functions should have also been implemented with blocking and vectorization.

6. REFERENCES

[1] Tom Mertens, Jan Kautz, and Frank Van Reeth, "Exposure fusion: A simple and practical alternative to high dynamic range photography," *Comput. Graph. Forum*, vol. 28, no. 1, pp. 161–171, 2009.

[2] Fabian Hahn and Alexandre Chapiro, "Fast exposure fusion on the cpu," 2013.

[3] Julien Pommier, "Simple sse and sse2 (and now neon) optimized sin, cos, log and exp," `http://gruntthepeon.free.fr/ssemath/`.