OPTIMIZING DOMAIN TRANSFORM FOR EDGE-AWARE IMAGE PROCESSING

Adrian Blumer, Pascal Spörri, Julia Pečerska

Department of Computer Science
ETH Zürich
Zürich, Switzerland

ABSTRACT

Edge-aware filtering can be used as a basic building block for many image processing tasks. In this field of research the recent publication *Domain Transform for Edge-Aware Image and Video Processing* [1] introduces a new and promising algorithm.

At the time of writing of this report, the only publicly avaiable implementation of this method is a Matlab version provided by the papers authors.

The goal of this project is to provide a performant C/C++ implementation of the aforementioned method, while documenting and discussing both successfull and failed optimisation approaches. Starting with a direct port of the Matlab code, we apply various optimisations to achieve an approximate 2 times speedup with regard to our base version. Optimisation potential still exists as we are far from both memory and computation bounds, however neither of our further optimisation attempts got us significantly closer to these bounds.

1. INTRODUCTION

Filtering operations are often a crucial step of image processing and a particular sub-group of these are edge-aware smoothing filters. They are used as a basic building block for multiple other applications. For example, such filters can be used to extract or remove features from an image, to colorize the image in a intuitive fashion by colouring specific items on an image, or to perform other tasks such as noise removal and image compression [1].

As in most cases with image (and in particular with video) processing, it is highly required that the filters have a low computational cost and therefore can produce fast results. This is desireable to allow for real-time transformation of video streams as well as immediate feedback during image editing and in general when working on large datasets. It is also true that image recording technologies advance and therefore can create images of greater resolution, which means an increase in the amount of computation needed for filtering.

Fig. 1. Sample input and output of our Normalized convolution implementation.

There exist multiple algorithms that perform this type of filtering, most of which fall into the categories of *Bilateral filtering* and *Anisotropic diffusion*. However most of these suffer either from quality, performance or scalability drawbacks. Moreover, some of them only natively support grayscale images.

Recently, a new method that promises to solve these problems was prososed by Eduardo S. L. Gastal and Manuel M. Oliveira in their paper *Domain Transform for Edge-Aware Image and Video Processing* [1].

This publication gives an in depth explanation of the theoretical foundations of the technique as well as presenting three different implementation approaches. These three approaches are called *Normalized convolution* (NC), *Interpolated convolution* (IC) and *Recursive filtering* (RF).

So far the only publicly available implementation of these methods is done in MATLAB, published on the original papers homepage. While useful for verification and demonstration purposes, its runtime performance is far from op-

timal. The work presented here aims at providing an optimized implementation of the *Normalized convolution* variant of the algorithm on the CPU while documenting the optimization process. We have implemented the algorithm in C/C++ and proceeded to use the techniques described in the *How to Write Fast Numerical Code* course [2] to improve its performance. Example input and output of our implementation can be seen in Figure 1.

2. BACKGROUND

In this section we will present the necessary background information to follow our implementation of the *Normalized convolution* domain transform filtering algorithm. For a more detailed explanation of the method and its theoretical background, please refer to the original paper [1].

The general outline of the algorithm is given in Figure 2. We distinguish two algorithm steps. First is the initialisation step, when the domain transform data of the image is computed and stored. Second is the filtering step, when multiple filter iterations are applied to the image data using the domain transform information.

2.1. Domain transform

The straightforward approach to image blurring is to compute a new value for each image pixel using the weighted average of its neighbourhood. To decide whether a pixel lies in this neighbourhood, some distance measure is necessary. In the case of simple edge unaware blurring (such as applying a Gaussian filter) this distance measure is the spatial distance between image pixels. However to achieve edge-aware image filtering the radiometric distance between pixels (their difference in color) has to be taken into account as well. In general this increases the dimensionality of the problem by adding a dimension for each colour channel.

For a RGB color image this would lead to a 5-dimensional space (3 radiometric + 2 spatial dimensions). The domain transform method reduces this dimensionality as follows. Firstly, it treats the two spatial dimensions separately, which means that the image is successively filtered along its rows and columns separately. Secondly, it merges the three radiometric dimensions and the remaining one spatial dimension into the one-dimensional space of the domain transform. This transformation retains the necessary distance properties from the original space.

Looking at a single image row, the pixels can be enumerated from left to right as $p_0 \dots p_n$. Each pixel p_i is associated with a domain transform value dt_i, and it holds that $\forall k : dt_k \leq dt_{k+1}$. In other words, the series of domain transform values is monotonically increasing. The exact difference between two consecutive domain transform values depends on the spatial and radiometric distance between pixels.

This data is computed and stored for all image rows and all image columns.

2.2. Filtering iterations

For the filtering step, multiple (usually three) filtering iterations are applied to the image. Each iteration consists of filtering along all rows and then filtering along all columns.

Filtering happens using a box filter. This means that a pixels new value is the uniform average of the pixels in its neighbourhood. For a single pixel p_i this neighbourhood consists of all pixels p_k for which it both holds that $dt_k < dt_i + r$ and $dt_k > dt_i - r$, where r is the filtering radius that changes from iteration to iteration. The size of this neighbourhood can vary significantly between pixels and strongly depends on the input image.

The use of a box filter might first look like a severe restriction as usually a Gaussian filter response is preferable. However the multiple consecutive iterations of box filtering in fact produce a Gaussian like filtering response.

2.3. Cost Analysis

In asymptotic terms, the algorithm has a complexity of $\mathrm{O}(n)$, which makes it already very competitive with other approaches to edge-aware image filtering. For optimization purposes we are however interested in a more thorough cost analysis of the method. Therefore we have counted the number of different floating point operations executed by the implementation, the numbers can be seen in Table 2.3. These numbers were used for performance computations and analysis in the course of optimisation.

The following variables are used in Table 2.3: W - width of the processed image, H - height of the processed image, N - number of filtering iteration.
`TransformedDomainBoxFilter` is the filtering function which executes one pass through the image in one direction. `Filter` is the main method called on the whole image. It includes all necessary precomputation and multiple iterations of the filter itself.

Operation	TransformedDomain BoxFilter	Filter
add	$2 \cdot N \cdot 4HW$	$(8 + 8N)HW$
subtract	$2 \cdot N \cdot 4HW$	$(6 + 8N)HW$
compare	$2 \cdot N \cdot 4HW$	$(8N)HW$
multiply	$2 \cdot N \cdot 3HW$	$(2 + 6N)HW + 3N$
divide	$2 \cdot N \cdot HW$	$(2HW + 1)N + 1$
fabs	0	$6HW$
sqrt	0	$3N$
powf	0	$2N$

Table 1. Floating point operations performed in the code

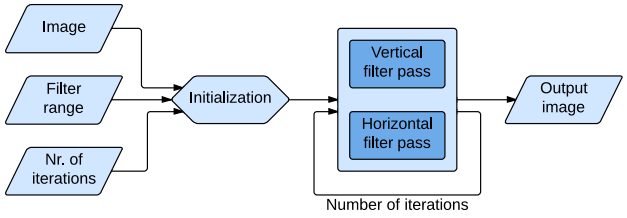

Fig. 2. Algorithm flowchart

3. OPTIMIZING DOMAIN TRANSFORM

This section describes the baseline implementation of the algorithm and the optimisation steps we executed that made a significant impact on the performance of the code. Finally the section also describes and discusses further optimisation approaches that were tried out and discarded because they didn't lead to a performance improvement.

3.1. Baseline version

Our baseline implementation is a straightforward port to C++ of the Matlab code provided on the homepage[1] of the authors of the paper describing the algorithm [1].

In order to preserve the precision of calculations all images are store using a `float3` struct, which uses $3 \times 4 = 12$ bytes per pixel. Hence a 1M pixel image is stored in $3 \times 4 \times 10^6$ B $= 12$ MB.

The algorithm starts with some initialisation procedures. During initialisation two matrices - $dIdx$ and $dIdy$ - are precomputed, which contain the differentials of the neighbouring image pixels in both x and y directions. Afterwards a prefix sum is computed for both $dIdx$ and $dIdy^T$, creating matrices which in fact contain for each pixel a mapping into 1D coordinate space along each direction.

During the filtering step (Listing 1) we compute `compute Bound` and `boxFilter` twice. After each computation step we transpose the images.

```
for (i=0; i<iterations; i++)
    bounds = computeBound(img, dIdX, r);
    img = boxFilter(img, bounds);
    img = transpose(img);
    bounds = computeBound(img, dIdY, r);
    img = boxFilter(img, dIdY, bounds);
    img = transpose(img);
```

Listing 1. Filterstep

The `computeBound` method computes the 1D upper and lower bound for the box filter of each pixel, which is dependent on the predefined filter radius.

[1] http://www.inf.ufrgs.br/~eslgastal/DomainTransform/

Before the actual application of the box filter, a prefix sum over the image pixels is computed. This cannot be done once for the whole algorithm and needs to be recomputed, as the pixel values change between iterations. Then the `boxFilter` makes use of the previously computed prefix sum and filter bounds to compute the filtered value for each pixel (Listing 2).

```
void boxfilter(img, bounds)
    psum = prefixsum(img);
    for (i = 0; i < H; i++)
        for (j = 0; j < W; j++)
            l = lowerBound(i,j);
            r = upperBound(i,j);
            delta = r - l;
            img(i,j) = psum(l)-psum(r);
            img(i,j) = img(i,j)/delta;
```

Listing 2. Boxfilter step

3.2. Optimization Steps

3.2.1. Blocked Transpose

After benchmarking the initial version we recognised that the transpose function was one of our bottlenecks. Hence we blocked the transpose function. Doing a transpose directly into a second array proved to be better than doing the transpose inplace.

3.2.2. Inlining Bound Computation

In order to allow the compiler to optimise the code we inlined the bound computation, which also allowed us to spare an additional pass through the data and thus reduce memory reads and writes.

```
void boxfilter(img, dx, boxR)
    psum = prefixsum(img);
    for (i=0; i<H; i++)
        posL = 0;
        posR = 0;
        for (j=0; j<W; j++)
            // Bound computation
            dtL = dx(i,j) - boxR;
            dtR = dx(i,j) + boxR;
            while (dx(i,posL) < dtL
                    && posL < W-1)
                posL++;
            while (posR < W &&
                   dx(i,posR < dtR )
                posR++;
            // Boxfilter step
```

Listing 3. Inlining of the bound computation

3.2.3. Recombination

In this optimisation step we tried to combine as many computation operations as possible in as few passes through the data as possible. We rearranged the initialisation step to conform to that idea, and inlined the prefix sum computation into the `boxFilter` bound computation step, getting a sliding window approach (Listing 4). This changes renders the prefix sum computation unnecessary as the sum of the values in the box filter window are now computed on the fly incrementally.

```
    while (dx(i,posL) < dtL
            && posL < W-1)
        sum -= imgRow[posL];
        posL++;

    while (posR < W &&
           dx(i,posR < dtR )
        sum  += imgRow[posR];
        posR++;
    // Boxfilter step
    invD = 1.0f /(posR-posL);
    img(i,j) = sum * invD;
```

Listing 4. Recombination

3.2.4. Write transpose

Previously we explicitly transposed the data after each pass of the filter to keep the locality of the computation. However this requires an additional pass through the whole image data. By instead directly storing the data transposed we were able to further increase performance (Listing 5).

```
void boxfilter(img, dx, boxR)
    for (i=0; i<H; i++)
        for (j=0; j<W; j++)
        // Prefix Sum & Bounding Step
        // Boxfilter step
        invD = 1.0f /(posR-posL);
        img(j,i) = sum * invD;
```

Listing 5. Write transpose

3.2.5. Vectorisation

As a last optimisation step we added vectorisation. To ensure proper alignment we added a fourth floating point value to the `float3` type (Listing 3.2.5).

```
union float3{
  struct{ float r,g,b,a; };
  __m128 mmvalue;
};
```

The vectorisation of the individual operations proved straightforward. The operations on the individual colour values could be replaced with a single vector operation. This also reduced the number of instructions in the code and thus the number of branch mispredictions as measured in VTune. Using this approach we could theoretically increase performance by a factor of three (instead of the usual four times factor, because 1/4 of the operations is executed on padding data). In practice we however only observed a small speedup of a few percent. Our explanation for this is that we exchanged instruction level parallelism with vector instruction parallelism. Using the vector instructions we do not have enough independent computations anymore to fill the instruction pipeline.

3.3. Failed optimisation attempts

3.3.1. Precomputation

Benchmarking the program with Intel VTune showed a high number of floating point division operations executed. Since the box filter computes the inverse of values in the range $[1, \text{image width}-1]$ we precomputed the inverse during compile time. Thus the precomputed values could be looked up from memory and be used in faster multiplication operations.

Using precomputed values in the range of $[1, 100]$ returned the better performance than precomputing all values. Unfortunately the performance attempt didn't bring any overall performance improvements compared to the method discussed in section 3.2.4.

This might be due to several factors. Firstly, it might be that such precomputed values already exist, as these rather small factors which might be used a lot and the time needed to lookup large values outweigh the time needed for computing the divisions on the fly. Or, secondly, it is possible that there was some increase in performance, but this was unnoticeable because the additional array accesses caused more cache misses, and, consequently, more overhead.

3.3.2. Changing image storage datatype

We tried to store the per pixel colour information in three `unsigned char` variables instead of the same number of `float` fields. This reduced the precision of intermediate storage but also reduces memory requirements for image storage by 75%.

Unfortunately, this also introduced additional conversion overhead for each operation: after loading from memory the char needs to be cast to int and from there to float. Once all computations are completed the floats need to be cast back again.

We felt that the conversion steps proved to much overhead (5 cycles latency per conversion) to increase perfor-

mance next to memory and additional code required.

3.3.3. Blocking

Unfortunately, one of the most common optimisation approaches does not work for the algorithm in question. In general blocking allows to increase cache locality, especially when the block sizes are calculated in a way such that the working set fits in cache. However that also means that the computation for each image pixel should depend on a fixed number of adjacent pixels, which isn't the case for the edge-aware filter.

We have gathered statistics on the amount of pixels used for the test images, and these were numbers between 30 and 150 pixels. That would mean that our block sizes should be around 150 pixels, which is too much to be a reasonable size. Also this number may increase up to the whole dimension of the image as the filter bound depend on the image data (for example in the case of a monochrome line in the image with a wide filter window). Smaller block sizes also don't make sense as in that case we would need to re-read information if the filter bound goes outside the block which would mean more overhead.

3.3.4. Instruction interleaving

We tried to combine the while loops that compute the borders for the filter. Here the idea was to interleave the two iterations to get more independent computations to fill the instruction pipeline. This did not give any performance improvement, perhaps because the combined loop still had conditional statements in between the independent addition operations.

3.3.5. Loop unrolling and reordering

We write the filtered image data in a transposed manner with the consequence that these writes are not memory aligned. Thus, for every pixel written, a new cache line is accessed and invalidated till the next access to it. This causes an additional memory overhead as always the whole cache line will be loaded from memory.

To improve on this situation we unrolled and reordered the loop iterations, with the goal that the computations happen in a more blocked fashion, causing more memory aligned writing operations. However this slowed down our code, possibly due to the increased amount of temporary register variables necessary.

We also tried to tackle this problem by using streamwrite intrinsics (direct writes that bypass the cache), but again with no success.

Image	Size	
`bhudda_200.png`	0.02	Megapixel
`bhudda_320.png`	0.07	Megapixel
`bhudda_640.png`	0.26	Megapixel
`bhudda_720p.png`	0.92	Megapixel
`bhudda_1080p.png`	2.07	Megapixel

Table 2. Images used for benchmarking

3.3.6. Reshuffling of data

Most of the versions we implemented work on the interleaved colour arrays, storing colours as $rgbrgb\ldots$. We attempted to switch from an array of structs to a struct of arrays, each array containing one colour. However, this didn't lead to a smaller working set size and didn't bring any performance improvement.

4. EXPERIMENTAL RESULTS

4.1. Experimental Setup

In order to test and benchmark the code we stored each major implementation revision in a separate folder. We then ran a script that compiled the code and benchmarked it against different parameter sets including 5 different image sizes (Table 2). The image sizes were chosen in such a way that they represent real world applications from image and video processing. The images `bhudda_200.png` and `bhudda_300.png` have been chosen since both fit into the L3 cache.

4.1.1. Platform

Benchmarking was done on Ubuntu Linux 12.10 running on machine with a Intel Core i7-3610QM "Ivy Bridge" CPU and 16GB of RAM. In order to generate reproducible results the CPU frequency was fixed to 2.3 GHz with Turbo Boost disabled.

4.1.2. Tools and counters used

The code was both compiled and executed on GCC 4.6.4 and ICC 13.1.2.183 64bit. We found that the compiler flags `-O3 -march=corei7` produced optimal results in our case. All our presented results were obtained with GCC as ICC produced very similarly performing code.

To measure the performance we used a custom timing framework using RTDSC counters. This framework was used both to compare between different code sections (to detect bottlenecks) as well as between different code revisions. For additional checks we used Intel VTune 2013[2].

[2] http://software.intel.com/en-us/intel-vtune-amplifier-xe

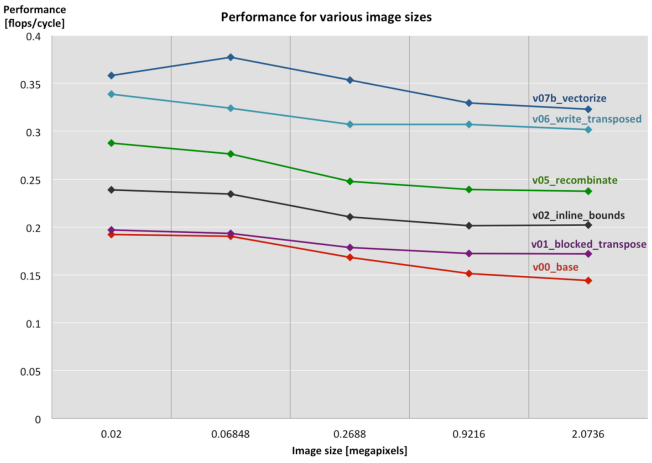

Fig. 3. Performance plot comparing our different code revisions. Compiler: GCC 4.6.4, flags: `-O3 -march=corei7`.

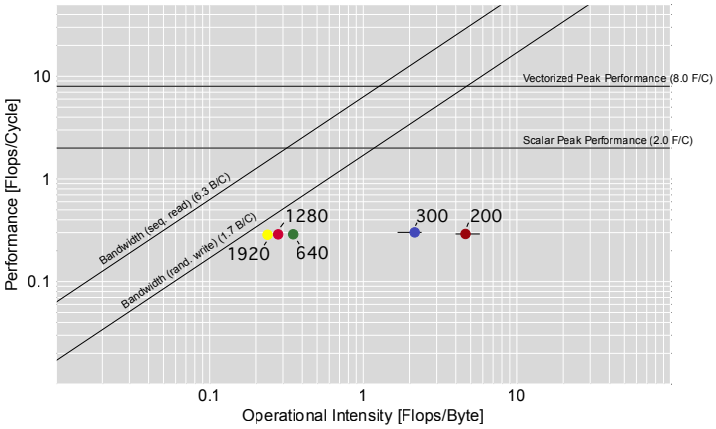

Fig. 4. Roofline plot showing the performance of our vectorised code for various image sizes (the labels indicate the image width).

In order to generate the roofline plots (Figure 4) we used the perfplot tool[3] provided by Georg Ofenbeck. To measure the memory bandwith of the reference computer we used an open source tool called Bandwith[4].

4.2. Results

We were able to achieve a speedup of 1.98 for the cache based image `bhudda_320.png` with a performance of 0.38 flops/cycle and a speedup 2.24 for our HD style image `bhudda_1080p.png` with a total performance of 0.32 flops/cycle (Figure 3). In our roofline plot [3] (Figure 4) we observe that we read/write bound for our `bhudda_1080p.png` image denoted as **1920** in the plot. As soon as we decrease the image size (**1280** denotes the image `bhudda_720p.png`, **640** denotes the image `bhudda_640.png`, **320** denotes the image `bhudda_320.png`, **200** denotes the image `bhudda_200.png`) we observe that the operational intensity changes but not the performance of the code (Figure 4). Hence there must be another limiting factor.

5. CONCLUSIONS

In this report we presented a summary of optimisation strategies for an edge preserving blurring filter. Even though the baseline implementation was already fast, we got an approximate $2x$ performance gain in our final code version.

During the course of optimisation we got rid of the primitive bottleneck which was the matrix transpose, by writing the data in transposed fashion directly. This allowed us to focus on the iterative part of the algorithm.

The way data is processed gives natural spatial locality and also instruction level parallelism, as 3 colour channels are stored contiguously but computed independently. However this can only be leveraged either by better instruction pipeline usage or vector instructions as discussed earlier.

We have attempted to analyze the performance bound of our code using roofline plots (see Figure 4). According to this, our fastest version is neither memory or compute bound. This probably means that we are bound by some other, unfortunately, unidentified factor.

[3] https://github.com/GeorgOfenbeck/perfplot
[4] http://zsmith.co/bandwidth.html

6. REFERENCES

[1] Eduardo S. L. Gastal and Manuel M. Oliveira, "Domain transform for edge-aware image and video processing," *ACM TOG*, vol. 30, no. 4, pp. 69:1–69:12, 2011, Proceedings of SIGGRAPH 2011.

[2] Markus Püschel, Daniele Spampinato, and Georg Ofenbeck, "How To Write Fast Numerical Code Webpage," http://www.inf.ethz.ch/personal/markusp/teaching/263-2300-ETH-spring13/course.html, Accessed: June 2013.

[3] Samuel Williams, Andrew Waterman, and David Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, no. 4, pp. 65–76, Apr. 2009.