Implementation of Depth Map Filtering Algorithms on Mobile-Specific Platforms

Olli Suominen, Sumeet Sen, Sergey Smirnov, Atanas Gotchev

***Abstract*—The paper addresses the problem of implementing depth map filtering algorithms optimized for mobile platforms. Main algorithm being targeted is the bilateral filter and its implementation on a mobile platform[1] has been studied. Furthermore, an alternative approach of using OpenCL to control a graphics accelerator[2] is explored. Experimental results of the latter look quite positive.**

I. Introduction

Modern modern video enhancement algorithms are considerably complex, and therefore the limitations of mobile devices have to be taken into consideration when balancing the needed computational power and the quality requirements. Bilateral filter has been considered in the Mobile 3DTV project both for denoising and deblocking of stereo video and post-filtering of depth maps [1]. The aim of research reported in this paper has been to explore, what kind of possibilities for implementation is offered by a platform corresponding to the level of current devices on the market.

As an alternative, the possibility of using OpenCL in future mobile products has also been considered. Such devices are not yet commonly available on the market, but some[3,4] have already been published. A GPU of a netbook[5], considered as a reasonable substitute for currently missing GPUs supporting OpenCL on mobile platforms, has been used for performance testing of the OpenCL implementation.

II. Bilateral filter

The bilateral filter takes into consideration not only the spatial domain, but also the color information (range) of the neighborhood when calculating the weights for any given pixel. This gives it the ability to preserve edges, which would be smoothed out by a traditional spatial filter such as a Gaussian [2]. The bilateral filter needs two functions to map the differences in value to weight values, the spatial coefficients $f(\boldsymbol{k})$ and range coefficients $g(\boldsymbol{x},\boldsymbol{k})$. Those functions can be arbitrary, but a suitable candidate is for instance the Gaussian. For a rectangular 2-dimensional window W of size $2r+1\times 2r+1$ around image point $\boldsymbol{x}$, the discreet response of the bilateral filter can be expressed as

$$y(\boldsymbol{x}) = \frac{1}{\kappa}\sum_{\boldsymbol{k}\in W} f(\boldsymbol{k})I(\boldsymbol{x}+\boldsymbol{k})g\big(I(\boldsymbol{x})-I(\boldsymbol{x}+\boldsymbol{k})\big), \tag{1}$$

where $\kappa = \sum_{\boldsymbol{k}\in W} f(\boldsymbol{k})g\big(I(\boldsymbol{x})-I(\boldsymbol{x}+\boldsymbol{k})\big)$ is a normalization term. The normalization term has to be computed for each point the filter is applied to, unlike in a purely spatial filter [3].

1) *Constant spatial filter*

Even though the filter is eventually applied like a regular convolution, the varying weights of the filter make it considerably heavy to compute. Three ways to speed up the processing were introduced by Porikli [3]. The simplest and fastest of these is that by setting the spatial filter coefficients $f(\boldsymbol{k})$ to a constant value c (i.e. a box filter), the response of the filter can be modified into

$$y_b(\boldsymbol{x}) = c\,\frac{1}{\kappa}\sum_{i} i h_{\boldsymbol{x}}(i) g\big(I(\boldsymbol{x})-i\big), \tag{2}$$

where $h_{\boldsymbol{x}}(i)$ is the histogram of the windowed image around the point $\boldsymbol{x}$. As all spatial locations have the same weight, the spatial location of the range values no longer matters. Therefore it is sufficient to form the response by accumulating over the bins of the histogram, not over single pixels. This approach has the added benefit of not being sensitive to the size of the window in terms of performance, assuming the computation of the histograms is not either. However, the performance is O(n) in terms of the amount of histogram bins used.

2) *Distributive histograms*

Porikli originally suggested integral histograms as the tool for computing the histograms needed for Eq.(2) [3]. Another method for this is the distributive histogram proposed by Sizintsev et al., which has a lower computational cost and memory requirement. The distributive histogram is based on the idea that the histograms of two consecutive sliding windows differ only by the histograms of the $1\times(2r+1)$ column segments that are included in one but not the other. If assuming the sliding window is traveling along the rows, the new local histogram can be computed from the previous by adding the histogram of the new column segment and subtracting the histogram of the column segment leaving the window. Similarly, computing the histogram of a column segment is done by respectively subtracting and adding the value of the two individual pixels that leave and enter the window from the histogram of the segment above it [4].

[1] TI OMAP 3530
[2] Nvidia ION GPU
[3] TI OMAP5432
[4] TI OMAP5430
[5] Asus 1201PN

3) *Hypothesis filtering*

In our application, we used the bilateral filter as the main building block of the hypothesis filter for depth map deblocking [1]. The bilateral filter is applied to filter a cost volume in order to find the optimal depth value for each pixel. The color and spatial weights are computed only once and then applied once for each of the slices of the cost volume. Winner-takes-all procedure across the slices selects the depth value.

III. IMPLEMENTATION

A. *Implementation environment*

The implementation environment included two platforms. The first platform consisted of a pair of ARM and DSP cores, where we paid special attention on the distribution of the tasks between the cores, the communications between then and the memory use. The second platform included a GPU and the focus was put on the proper parallelization of the algorithms.

The GPU is accessed via OpenCL. Maintained by the same consortium as, among other things, its graphics counterpart OpenGL, OpenCL is an open standard for utilizing the computing power of graphics processors beyond their original purpose of graphics rendering. It allows the programmer to access the capabilities more freely in the form of *kernels*, special programs written in a modified C-language that are executed on the graphics hardware in parallel.

While the following comparison is highly volatile in the sense that GPU performance cannot be simplified to just counting theoretical operations, it will give some impression of scale. The GPU used in these experiments has two multiprocessors running at 1092 MHz each. The SIMD width of those multiprocessors is 8, so as a rough generalization, it is capable of executing 17.5 billion effective operations per second. In comparison, some of the more modern mobile GPUs mentioned before are ranging from 1 to 16 cores with SIMD width of 4, running at 200-400 MHz. Depending on configuration, this would result in anywhere between 0.8 and 25.6 billion operations per second. This extremely coarse estimate hints that our choice of a mobile GPU substitute is justified.

For a detailed description of the implementation environment we refer to [6].

B. *Bilateral filter on mobile platform*

The basic algorithm was implemented as reported in [1]. First, a pure C implementation was run on the ARM core with enabled NEON support for simultaneous operations showing a performance of 20 seconds per frame.

The constant spatial filter implemented on the DSP performed at 16 seconds per frame with 64 bin histograms. By modifying the code to allow the compiler to perform some of the loops in parallel, most importantly the summation described in Eq. (2), the performance increased to 6 seconds per frame.

The earlier attempts had only used the ARM core to initialize the program and for file I/O. While the DSP was working on the filtering, the ARM was doing nothing. When building distributive histograms was moved to the ARM, the simplification of DSP side control structures gave a larger performance boost than could have been expected from moving the less computationally expensive histogram operations to be done in parallel. Some compromises regarding the accuracy on computation had to be made to further increase performance. The use of the fixed point arithmetic library was removed altogether, and all DSP side computation was converted into half-word (16 bits) integer operations with simulated decimal accuracy up to 3 decimals. This in addition with reducing the amount of histogram bins from 64 to 16 brought the time down to 0.66 seconds per frame.

C. *Bilateral and hypothesis filter on OpenCL*

An alternative approach was considered: implementing the bilateral filter on a GPU. Even though not yet commonly available on the mobile market, OpenCL can be expected to be a viable method of exploiting the hardware of next generation mobile devices. This will give an idea how well the algorithm deploys to that kind of platform.

The distributive histogram method used in the implementation in Sub-section III.B is an iterative process. It would be very poorly suited on a heavily parallelized platform such as the GPU. Therefore the algorithm implemented in OpenCL is the direct bilateral filter with both the distance and range weight functions being arbitrary. The actual shape of the distribution is irrelevant in terms of speed, as both functions can be precomputed once at the start of the processing for necessary values, and used as a lookup table from the OpenCL *constant memory*, which is cached to allow fast access.

An OpenCL application was constructed in a way that there is a thread for each pixel, which computes Eq.(1) for a window centered at that pixel. Each thread has to access the surrounding $(2r+1)^2$ pixels around its assigned pixel in order to compute the output of the filter. As the amount of memory accesses is large and several threads must access the same pixels, it is reasonable to use an *image object* to store the input image instead of a traditional buffer. Image objects are spatially cached, i.e. memory accesses to the same area of the image do not have to be retrieved from expensive global memory, but are available in the device cache. Each thread loops through the sum in Eq. (1) reading and processing all the channels for a single pixel on each iteration.

After processing, the image can either be returned to the CPU for further processing, buffering, saving etc, or using OpenCL/OpenGL interoperability, sent directly to the display.

IV. EXPERIMENTAL RESULTS

The test material used was a 480x272 YUV420 video sequence from the Mobile 3DTV project's video library [5], which has been impaired with a low quality level DCT compression, resulting in very clear block boundaries, although the performance of the used techniques is not dependant on the amount of degradation in the video.

The choice of color model between RGB and YUV (or CIE-

Lab, as suggested by Tomasi [2]) has an effect on the computational performance of the filtering, depending on from which channels the weights are computed and to which channels they are applied. The ARM+DSP version was not extended to cover more than the Y channel as it was already too slow for the purpose with only a single channel.

The OpenCL version uses all three RGB channels for calculating the weights and applies them to all three channels, so it gives in a sense the worst-case performance in terms of channel selection.

A. *Bilateral filter on DSP+ARM*

The final version of the ARM+DSP implementation of the bilateral filter processes one frame in 0.66 seconds. The computation of histograms for all of the window positions (centered on each pixel) takes approximately 250 ms on the ARM processor and the application of weights into the image 600 ms. The ARM side does not take advantage of NEON optimization, but as the bottleneck is the application of weights on the image and the operations are done in parallel, it has no effect to the overall processing time. The rest of the time, approx. 60ms is spent on moving content in memory etc. The total time can be expected to increase significantly if more than one channel is taken into account. As this is clearly too low to even consider a real time application, we did not pursue a DSP implementation of the hypothesis filter, which would require several passes of the bilateral on each frame.

In comparison, Porikli achieved 0.06 seconds per frame on a desktop computer on a 1MB grayscale image with 16 bins [3]. This performance was roughly matched by running practically the same implementation on a PC as on the DSP, with the exception of using floating point arithmetic instead of fixed point. Further reducing the number of histogram bins from 16 was found visually disturbing due to the “comic book effect” caused by reduced color depth.

B. *Bilateral and hypothesis filter on OpenCL*

The key factor in the performance of the OpenCL application is the time it takes to transfer the data to the device, process it, and retrieve it back. Using an OpenCL specific profiler, it was determined that the 522kB of 32 bits per pixel image data takes 2.5ms to be transferred to or from the device, making the memory transfer take 5ms total. Processing time of the kernel is dependent on the window size.

The cache miss/hit rate decreases steadily when the window size is increased, going down to 0.2% on the largest tested window, i.e. only 0.2% of memory accesses have to be done from the memory, and all the rest come from the fast cache. This means that the bigger the window, the more use the GPU gets from the caching properties of the image object. The results in Figure 1 are given for processing all three color channels.

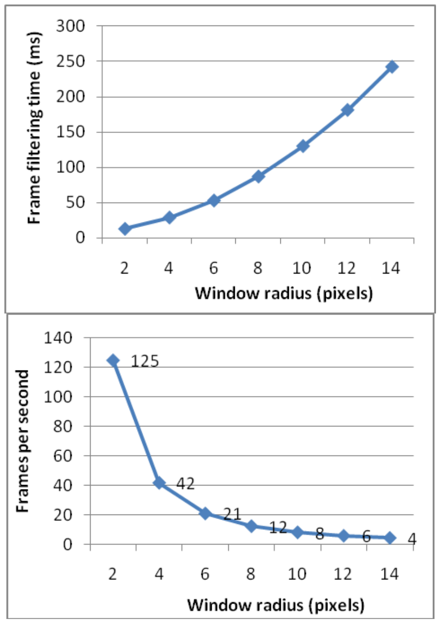

Figure 1 – Top: time in milliseconds it takes to process one frame of 480*272 color video in RGB space on the GPU. Bottom: corresponding single video stream FPS resulting from those times. Window radius is the distance from the middle pixel to the edge of a square window.

In hypothesis filtering, the weights are applied to a single channel (depth), but several times per frame. Speed of the hypothesis filter was evaluated by a visual profiler as equal to 43 ms/frame for filtering with block size of 9x9 pixels. This accounts for approximately 23 fps which is a good compromise between spatial improvement of visual quality and sufficient speed. Figure 2 illustrates the performance in terms of FPS for different filter sizes for the ‘Bullinger’ sequence. An illustration of filtering results for the ‘Car’ sequence is given in Figure 3.

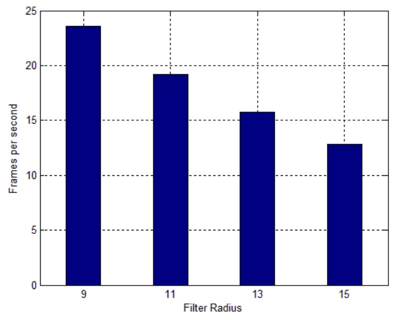

Figure 2 – Performance of Hypothesis filter for Bullinger (320 x 192) sequence for varying filter sizes.

V. CONCLUSIONS

The bilateral filter is well suitable for parallel processing, as there are no dependencies between generating outputs of neighboring pixel values. The DSP approach gained significant speed improvement from the utilization of the device’s SIMD instructions.

However, the performance of the bilateral filter on the ARM+DSP platform did not reach the objective of processing view plus depth video in real time. While the implementation is likely not perfectly optimized, the key factors have been taken into account. Still, the speed is at least an order of magnitude slower than needed for a real time application. Furthermore, it would also not be possible to allocate all the resources of the platform to video post-processing, as the decoding and color space conversion also run at the same time. One can resort to implementing the filter in specialized hardware, for which the current tests are quite instructive. It would seem that for the current generation mobile hardware, this kind of processing is too intensive.

The OpenCL version looks very promising in terms of computational performance. Even when performing the filtering on all RGB channels, reasonable speeds are achieved. With this implementation, a window radius of approximately 4, i.e. 81 pixels in total would be feasible in a real time, full-color stereo video application. Further improvements are possible by code-level optimization and taking advantage of all the properties the GPU hardware is offering. It is also only using the GPU, leaving the CPU free for other tasks. Information on those OpenCL compliant, future mobile platforms is still scarce. Further studies are needed when samples and development tools are available, but at the moment, utilizing the GPU seems to be a potential branch of mobile related research and development in video processing.

REFERENCES

[1] S. Smirnov, S. Sen, A. Gotchev, H. Burst, G. Tech, '3D Video processing algorithms, Part I', Mobile3DTV tech. report D5.4, Feb. 2010.

[2] C. Tomasi, R. Manduchi "Bilateral Filtering for Gray and Color Images"**.** *Sixth International Conference on Computer Vision*, 1998.

[3] Porikli, Fatih. "Constant time O(1) bilateral filtering", *IEEE Conference on Computer Vision and Pattern Recognition*, 2008. pp. 1-8.

[4] M. Sizintsev, K.G. Derpanis, A. Hogue, "Histogram-based search: A comparative study," *IEEE Conference on Computer Vision and Pattern Recognition*, 2008

[5] Mobile 3DTV 3D video database, [Online] 2009. http://sp.cs.tut.fi/mobile3dtv/impaired-videos/

[6] L. Azzari, O. Suominen, S. Sen, A. Gotchev, D. Bugdaici, G. B. Akar, '3D Video processing algorithms, Part II', Mobile3DTV tech. report D5.6, May 2011. p. 65.

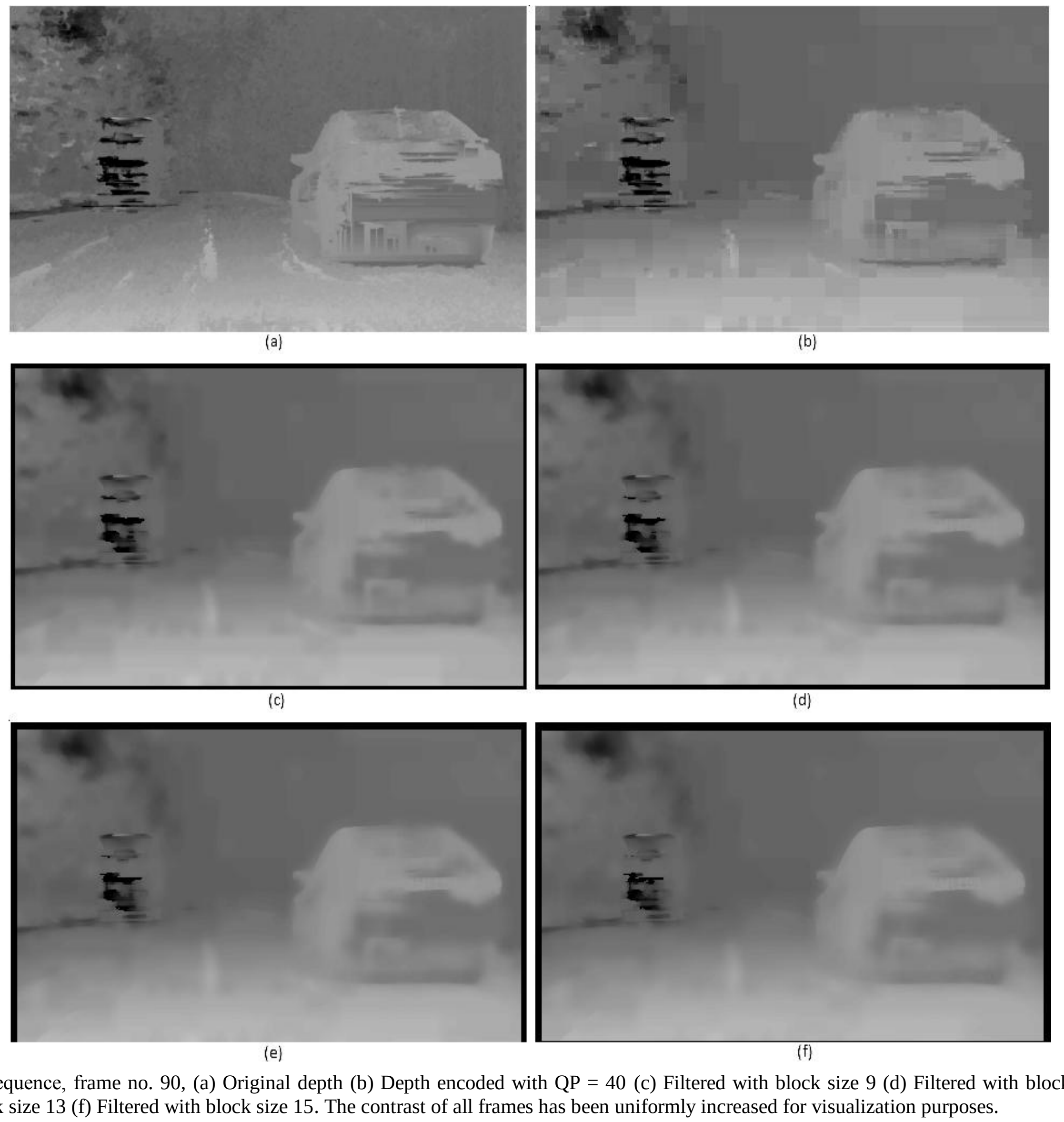

Figure 3 – Car' sequence, frame no. 90, (a) Original depth (b) Depth encoded with QP = 40 (c) Filtered with block size 9 (d) Filtered with block size 11 (e) Filtered with block size 13 (f) Filtered with block size 15. The contrast of all frames has been uniformly increased for visualization purposes.