In Proceedings of the Tenth International Conference on Information Visualisation. Los Alamitos: IEEE Computer Society, p. 609-614. 2006.

FFT and Convolution Performance in Image Filtering on GPU

Ondřej Fialka, Martin Čadík
Department of Computer Science and Engineering, Czech Technical University in Prague
Karlovo náměstí 13, 121 35 Prague, Czech Republic
fialko1@fel.cvut.cz, cadikm@fel.cvut.cz

Abstract

Many contemporary visualization tools comprise some image filtering approach. Since image filtering approaches are very computationally demanding, the acceleration using graphics-hardware (GPU) is very desirable to preserve interactivity of the main visualization tool itself. In this article we take a close look on GPU implementation of two basic approaches to image filtering – Fast Fourier Transform (frequency domain) and convolution (spatial domain). We evaluate these methods in terms of the performance in real time applications and suitability for GPU implementation. Convolution yields better performance than Fast Fourier Transform (FFT) in many cases; however, this observation cannot be generalized. In this article we identify conditions under which the FFT gives better performance than the corresponding convolution and we assess the different kernel sizes and issues of application of multiple filters on one image.

Keywords— **Fast Fourier Transformation (FFT), convolution, Graphics Processing Unit (GPU), image filtering**

1 Introduction

Many image processing techniques that until most recently required a considerable amount of CPU time are becoming available for real time applications thanks to modern graphics hardware (GPUs). These techniques range from simple blurring, sharpening or fog effects to sophisticated methods of tone mapping of high dynamic range (HDR) images. Many of these techniques apply various filters in the process of image transformation and advanced effects can be achieved by combining several filters. Therefore the application of filters is one of the key issues in modern computer graphics. There exist two basic approaches to filtering – filtering in the *spatial* and filtering in the *frequency domain*.

While in the spatial domain we apply filter by simple *convolution* of the image and the kernel of the filter function, in the frequency domain we must convert the image into the frequency domain using *Fourier transform*, then we must apply the filter by multiplication and finally we have to transform the result back to spatial domain using inverse transform. Both of these approaches have their pros and cons and are profitable under different conditions. CPU based implementations are well known and have been described thoroughly. However, GPU implementations bring many new issues that have to be resolved. This article takes a look at the implementation of filtering techniques on GPUs and evaluates their performance in real time applications. The focus is given on the usage in processing (filtering) of images, see figure 1.

2 Previous Work

Fast Fourier Transform is a well known technique and many slightly different FFT algorithms have been proposed in history. The problem is that most of them cannot be directly implemented on a GPU. Moreland and Angel [5] were first to describe how to implement FFT on a GPU, but their work is out of date now since they used older graphics cards (NVidia GeForce 5000 series) than those available nowadays (6000 and 7000 series) and therefore their possibilities were somewhat limited. A different and a very effective approach for medical images was developed by Sumamaweera and Liu [8]. Their solution adopts the decimation in time algorithm and enables the computational load to be split between the pixel and vertex shaders and the rasterizer. However, the performance speedup introduced by splitting the load is not very significant – about 10% according to their results.

Most of the proposed approaches consider just greyscale images when it comes to FFT. Quaternion Fourier Transform [1] is a hypercomplex (numbers with four components) version of the standard (complex) FT and can be used for processing of images with up to four channels. Much work has been done on applying FFT and convolution in signal processing [7]. Smith in his book [7] compares both methods in term of performance for CPU implementations. We make a similar comparison for GPU implementations, which differ in many aspects from traditional CPU algorithms.

Image filters have been extensively used in many HDR tone mapping techniques [3, 9]. Despite this fact, not much

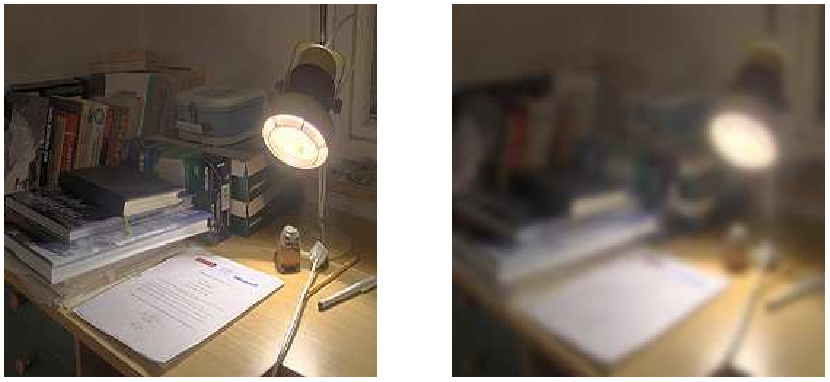
Figure 1: Image filtering: an example of the application of the low pass Gaussian filter using four-channel FFT (256x256). Left: original image, Right: result of the filtering process.

work has been done in comparing the FFT and convolution on GPU considering their pros and cons for HDR applications.

3 Theory and Implementation Issues

3.1 Discrete Fast Fourier Transform

One-dimensional Fourier transform of discrete function x(n) with N samples is defined as follows:

$$X(k) = \sum_{n=0}^{N-1} x(n) W_N^{kn}, \qquad 0 \le k \le N-1,$$

where

$$W_N^{kn} = e^{-i\frac{2\pi kn}{N}}. \tag{1}$$

The FT is separable and we can thus compute 2D FT by applying 1D FTs to all rows and then to all columns (or vice versa). A naive approach yields $O(N^2)$. So-called fast Fourier transform (FFT) algorithm reduces the complexity to $O(N \log N)$.

The algorithm for FFT we chose to implement is known as *decimation in frequency 2-radix* (DIF). It is easily mapped onto the GPU computational model. It can be described by a butterfly network shown in figures 2 and 3. Input values are on the left side, arrows indicate the distribution of values. Circles represents addition, -1's indicate sign reversal and W's indicate complex multiplication as in equation (1). The sequence of transformed values comes out in bit-reversed order and therefore reordering is necessary.

GPU implementation of FFT

As stated previously, the transform of a 2D image is done by applying 1D FFT's to all rows and columns consecutively. All 1D transforms can be applied in $\log_2 N$ steps (image N by N). In each step, pixels form pairs between two groups – G(i) and H(i) – as shown in figure 3. Instead of using branching and making different computations based on which group the current pixel belongs to, it is more efficient to execute the same instructions in both cases. The only difference between them is the sign and coefficients of the complex multiplication which are all fetched from a precomputed texture. The complex multiplication by the factor W from equation (1) can be decomposed as:

$$(a+bi)\ e^{-i\frac{2\pi k}{N}} = a\ \cos\left(\frac{2\pi k}{N}\right) - b\ \sin\left(\frac{2\pi k}{N}\right) +$$

$$+i\left[a\ \cos\left(\frac{2\pi k}{N}\right) + b\ \sin\left(\frac{2\pi k}{N}\right)\right]$$

Because there are only N distinct values of the sines and cosines, they can be easily precomputed and fetched from a texture.

After the last step of DIF, the data comes out in bit-reversed order. Reordering can be easily embedded into the last step of DIF, where we sample with bit reversed coordinates.

A single auxiliary data texture is used, see figure 4. Channel *x* contains the values +1 or -1 denoting addition or subtraction (as shown in figure 3). Channel *y* holds the coordinate of the other texel from the current pair. Channels *z* and *w* are used for the values of the corresponding sines and cosines. For the last step of DIF, no sines and cosines are needed and channel *z* holds the bit-reversed coordinate of the current pixel.

It is crucial for data persistence to map output pixels directly to input data texture texels. Figure 5 shows a naive mapping of a 4x4 texture into a 6x6 window. The origin of the pixel coordinates is different from the origin of texturing coordinates. Therefore, we need to shift the texturing quadriliteral in a corresponding way.

The algorithm described here operates with complex numbers and is suitable for monochromatic images or

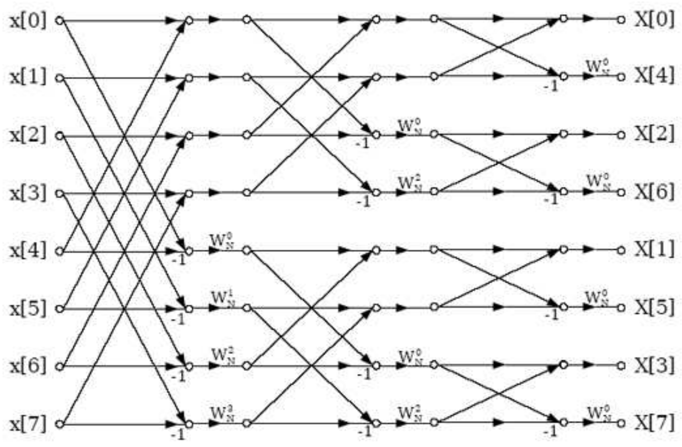

Figure 2: Decimation in frequency 2-radix butterfly network for 8 values

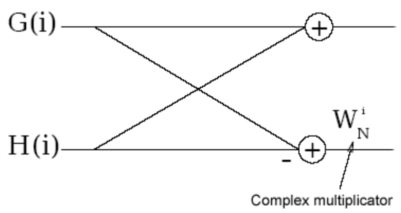

Figure 3: Detail of one "butterfly" from the DIF network

for images with two channels. Three (RGB) and four (RGBA) channels can be processed by the means of hyper-complex quaternion Fourier Transform (QFT). Quaterions are hyper-complex numbers with four dimensions – one real and three imaginary – i, j, k. The problem of QFT is described in [1].

Another and more straightforward approach to three- and four-channel images is representing the channels as real and imaginary parts of two images and performing two FFTs in parallel. Most of the instructions can operate on both FFTs at the same time thanks to the swizzle operator and therefore the slowdown is relatively small. We employed this method in our measurements.

3.2 Convolution in spatial domain

Convolution of two discrete functions over a finite interval is defined as follows:

$$y(k) = \sum_{n=0}^{N} x(k-n)h(n), \quad n \in \langle 0, N \rangle .$$

In the case of image processing, one function represents the image and the other is the kernel of a filter.

The implementation of convolution on a GPU must reflect the size and type of the filter kernel. The bigger the kernel, the more operations have to be executed per pixel. This can be easily implemented using a loop but we are limited by the maximal number of instructions the shader can execute per pixel. This limit is very strict with SM (Shader Model) 2.0 while SM 3.0 offers more freedom and allows much bigger kernels to be used.

Another aspect is the separability of the kernel. A separable kernel can be decomposed into two one-dimensional convolutions and can thus save many instructions – quadratic complexity becomes linear.

An optimized GPU implementation stores the kernel as well as the sampling coordinates in a uniform array which is precomputed on the CPU.

4 Comparison and Discussion

All implementations and measurements were done on a machine with 2.6GHz Intel Celeron D processor, 1GB RAM and NVIDIA GeForce 6600 GT 128MB at PCI-E. Programs were implemented using HLSL and DirextX 9.0c. Both the CPU and GPU are typical mainstream hardware of the beginning of 2005.

The GPU implementations of both the FFT and spatial convolution clearly outperform CPU versions. Figure 6 charts performance results for spatial convolution (for textures 256x256 and 512x512) – 32bit GPU version is 3 times faster than the CPU implementation and 16bit GPU version gives an average speedup of 7.4 over 32bit CPU implementation (16bit precision brings no advantage on a 32bit architecture).

Complex FFT of a 256x256 texture took 5.9ms on the GPU while CPU implementation took about 20ms – i.e. the speedup of 3.4. For the CPU FFT algorithm we used FFTW libraries by M. Frigo and S. G. Johnson [2]. It is a heavily-optimized algorithm that uses advanced features of today's CPUs. Multiplying the transformed data array with a filter took additional 20ms on the CPU while it added only about 0.1ms on the GPU.

Basic performance

In the basic performance test we compared the framerate of a Gaussian low pass filter applied by convolution and FFT to a simple testing scene. If not stated otherwise, we used the monochromatic version of FFT. Three textures of the sizes 256x256, 512x512 and 1024x1024 were used. Image 7 shows the framerates of FFT compared to separable and unseparable convolutions with different kernel sizes.

As we can observe in the chart, the ratio of framerates of convolution and FFT is not the same for different texture sizes. This can be explained by the number of elementary computation steps. The computational complexity of filtering a N by N texture with separable convolution is

$$O\left(2k \cdot N^2\right),$$

where k denotes the size of the kernel – every pixel gets convoluted with the whole kernel in x and y dimensions. With FFT the time increases to

$$O\left(4N^2 \cdot \log_2\left(N\right)\right)$$

– each pixel is processed in $\log_2 N$ steps that reapeat four times – for both dimensions and for forward and backward transforms.

Applying filters by *separable convolutions* is much faster than by the FFT for small filter kernels. The *size of the kernel* for which convolution is still faster than FFT depends on the size of the texture. In our tests it lies between 31 and 41 pixels for monochromatic FFT. The RGBA version of FFT is slightly slower. Due to optimizations in the GPU, the same instructions run faster when two components (z,w) of the processed four-vectors are zero, which is the case of monochromatic and two-channel FFT. Quaternion Fourier Transform was not tested, but its computational complexity should be comparable to our four-channel version of FFT.

The situation is different for *unseparable filter kernels* (e.g. edge detection in a given direction). The complexity is

$$O\left(k^2 \cdot N^2\right),$$

making it very slow. Another limitation is the number of instructions the pixel shader can execute. Due to this limitation, the largest odd-dimension filter we could implement was 9 by 9 (maximum number of samples per pixel was 112). Separable convolutions are limited by this feature as well.

Multiple filters

There are many applications, including tone mapping operators like [3, 4], that process the original scene by multiple filters, evaluate the results and compose the final image based on the results of individual filters. While convolution has to be repeated for every applied filter, with FFT there is only one forward transform, and multiple backward transforms. This phenomena is depicted in figure 8. It plots the framerates of multiple filters being applied to the same scene.

As we can see in the chart, the FFT becomes gradually more efficient with the increasing number of filters. While it is approximately as fast as convolution with a 33 pixels wide kernel for one filter, it reaches the performance of convolution with a 21 pixels wide kernel for eight filters.

Bit depth of textures

Another issue is the bit depth of textures that can be used with these techniques. Because the FFT requires many passes, 16 bits per channel are necessary even for non-HDR images. Convolution in spatial domain can be done with only 8 bits per channel in case the application does not require higher range. The bit depth has a major impact on the speed – the speed ratios for 8 bit integer and 16 and 32 bit floats per channel are approximately 4:2:1.

The overview of major pros and cons of the convolution and the FFT is recapitulated in table 1.

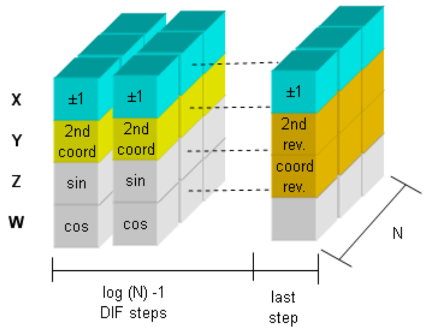

Figure 4: Scheme of the auxiliary texture for FFT

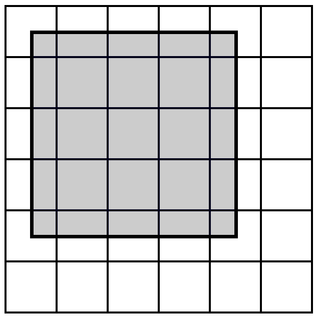

Figure 5: Mapping pixels to texels

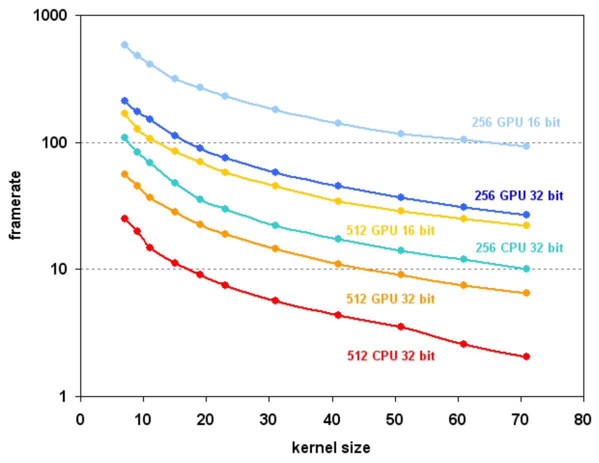

Figure 6: Comparison of GPU and CPU implementations of convolution (textures 256x256 and 512x512)

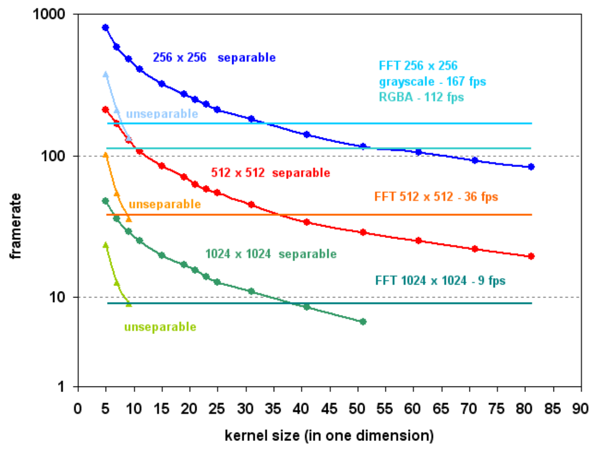

Figure 7: Comparison of FFT and convolution, 16 bit float textures

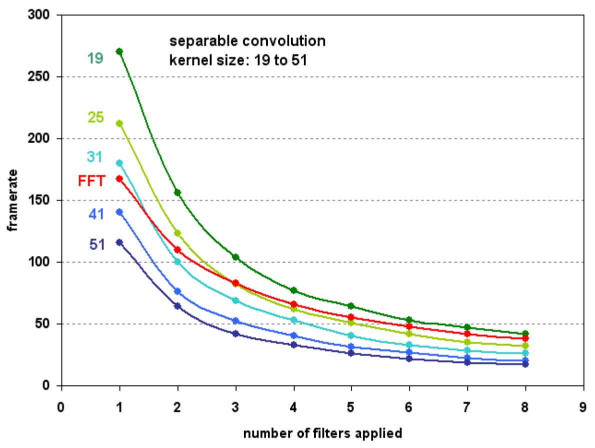

Figure 8: Comparison of FFT and convolution for multiple filters, 256x256, 16 bit float textures

5 Conclusions

In this article we have proposed new solutions of GPU-implementation issues for two main approaches to image filtering – the convolution and the fast Fourier transform. We have performed several measurements to asses the performance of these two approaches.

The results show that GPU implementations of both the convolution and the FFT outperform their CPU counterparts. Convolution on GPU performs better than the FFT on GPU when we want to apply simple and small filters. Separable convolution allows more optimizations, because for kernels up to approximately 100 pixels in size all the values and texturing coordinates can be precomputed and stored in a uniform data array. This offers a significant speedup in comparison with storing the values in textures or computing them directly.

However, the fast Fourier transform can outperform convolution in several cases:

1. When applying multiple filters on the same image – with FFT, we only need one forward transformation.

2. Unseparable filter kernels – i.e. the edge detection filters in chosen directions. Convolution of unseparable filters has quadratic complexity and therefore becomes very slow when increasing the kernel size. Furthermore, it is strictly limited by the number of instructions a GPU can execute.

3. Very large filter kernels – more than 33 pixels for one or two channel images, more than 51 pixels for three or four channel images.

FFT	Convolution
+ Takes the same time for all filters	- Complexity depends on the size of the kernel. Can be slow for large or unseparable kernels.
+ No hardware restrictions on filters	- The size of the kernel is limited by the graphics hardware
+ Works with all frequency filters	- Suitable kernels do not exist for all filters
+ Slowdown due to multiple filters is smaller than with convolution	- Multiple filters introduce significant slowdown
- FFT is time consuming	+ Very fast for small kernels
- Many passes make FFT sensitive to precision, requires at least 16 bits per channel textures even for non-HDR applications	+ Precision is not critical, 8 bits per channel textures may be used for non-HDR applications
- RGB (RBGA) images require more complicated quaternion Fourier Transform or two standard FFTs that both introduce a slowdown	+ Works for RGBA without problems
- Correctly handles only images with dimensions that are powers of two, images of different sizes have to be extended and filled with zeros	+ Any image size is possible

Table 1: Pros and cons of the two approaches

Acknowledgements

This project has been partly supported by the Ministry of Education, Youth and Sports of the Czech Republic under research program No. Y04/98: 212300014.

References

[1] Bas, P., Le Bihan, N., Chassery, J. 2003. *Color Image Watermarking Using Quaternion Fourier Transform*. Laboratoire des Images et des Signaux INPG/CNRS.

[2] Frigo, M., Johnson, S. G. 2005 *FFTW*. Fast Fourier Transform library. http://fftw.org

[3] Krawczyk, G., Myszkovski, K., Seidel, H. 2005. Perceptual Effects in Real Time Tone Mapping. *Proceedings of the 21st spring conference on Computer graphics*, 195 - 202.

[4] Mantiuk, R., Myszkowski, K., Seidel, H. 2004. Visible Difference Predicator for High Dynamic Range Images. *IEEE International Conference on Systems, Man and Cybernetics*, 3, 2763 - 2769.

[5] Moreland, K., Angel, E. 2003. The FFT on a GPU *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware* session: Simulation and computation, 112 - 119.

[6] Pharr, M. et al. 2005 *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. NVIDIA, Addison-Wesley.

[7] Smith, S. W. 1997. *The Scientist and Engineer's Guide to Digital Signal Processing*. California Technical Publishing.

[8] Somanaweera, T., Liu, D. 2005. Medical Image Reconstruction with the FFT. In *Programming Techniques for High-Performance Graphics and General-Purpose Computation*, Addison-Wesley.

[9] Thompson, W., Shirley, P., Ferwerda, J. 2002. A Spatial Post-Processing Algorithm for Images of Night Scenes. In *Journal of Graphics Tools* 7, 1, 1-12.