Efficient Cubic B-spline Image interpolation on a GPU

F. Champagnat and Y. Le Sant

September 1, 2011

1 Abstract

Application of geometric transformation to images requires an interpolation step. When applied to image rotation, the presently most efficient GPU implementation for the cubic spline image interpolation still cost about 10 times as much as linear interpolation. This implementation involves two steps: a prefilter step performs a two-pass forward-backward recursive filter, then a cubic polynomial interpolation step is implemented thanks to a cascade of linear interpolations. This paper proposes a simpler and faster implementation for the prefilter – which is the most time consuming – in terms of a direct convolution. The overall cost for our cubic B-spline interpolation algorithm then reduces to only twice the cost of linear interpolation.

2 Introduction

Accurate interpolation is required in many image processing low level tasks such as image zooming and warping, registration and optical flow estimation [4]. Cubic B-spline (CBS) interpolation is known to provide results with superior accuracy compared to nearest neighbour or linear interpolation or even cubic convolution interpolation [9]. These empirical facts are backed by different theoretical arguments, in particular the fact that CBS is a good approximation to sinc interpolation [10].

The algorithm for CBS interpolation that enjoys the least numerical complexity is due to Thévenaz *et al.* [9], it involves two steps : prefilter the image using a separable infinite impulse response (IIR) filter, then perform cubic polynomial interpolation. We focus on the GPU implementation of this algorithm.

The state-of-the-art method for CBS interpolation on a GPU is due to Ruijters and Thévenaz [7]. It follows the original two-step logic and takes advantage of the massive parallelism of GPUs and of hardwired linear interpolation on GPUs. Noticeable gains of the GPU over the CPU counterpart are demonstrated in [7]. But it should be emphasized that the prefilter step is usually slower than the polynomial interpolation step.

The presented technique deals with the prefilter step and corrects this discrepancy providing a 10 times acceleration factor compared to Ruijters and Thévenaz prefilter [7].

Such an acceleration is obtained using a much simpler technique than Ruijters & Thévenaz, it consists in replacing the two-pass forward-backward recursive filter [7] by a more classical direct convolution with a finite impulse response (FIR) filter.

An original and simple analytical expression of the prefilter coefficients is provided. It shows exponential decay, therefore long range interaction of the exact prefilter are actually negligible. The order of truncation is tuned so as to give negligible difference with Ruijters & Thévenaz filter.

A rough complexity analysis shows that turning from the original recursive filter to a FIR convolution filter amounts to increase the flops per pixel by a factor of five. But in the meantime we switch from a "one line/column per thread" to a "one pixel per thread" logic which implies a better occupation of the processing unit. Another fact that explains the efficiency of our approach with respect to the recursive one is that it avoids the dual memory access involved by the forward-backward process.

The CUDA code for CBS image interpolation using the proposed prefilter implementation is freely available on the ONERA website, at the address:
http://www.onera.fr/dtim-en/gpu-for-image/index.php.

The paper is organised as follows: Section 3 gathers the main facts about image B-spline interpolation and current trends on implementation. Section 4 briefly discusses the challenge of these implementations for GPU architecture in particular the bottleneck of prefiltering. Section 5 presents our solution for efficient prefilter implementation. Section 6 is devoted to implementation details, runtime measurement experiments are gathered in Section 7.

3 B-spline interpolation in 1-D

B-spline interpolation provides an approximation of tunable accuracy to the sinc interpolator, their use has been popularized in image processing in a series of papers by Unser *et al.* [9][10] that exhibited their theoretical properties and proposed very efficient implementation techniques.

The zeroth order B-spline is a symmetrical box function of unit mass:

$$\beta^0(x) = \begin{cases} 1, & |x| < \frac{1}{2} \\ 0, & |x| > \frac{1}{2}. \end{cases} \tag{1}$$

The nth order B-spline is the $n+1$ fold convolution of β^0:

$$\beta^n = \beta^{n-1} * \beta^0. \tag{2}$$

It is a piecewise polynomial of degree n with a symmetrical bell shape [10]. The general form of the nth degree B-spline interpolation of a signal s is a sum of shifted B-splines weighted by coefficients $c(k)$:

$$s(x) = \sum_k c(k)\beta^n(x-k). \tag{3}$$

Increasing the order of the B-spline family yields converging approximation to the sinc interpolator [10].

B-spline interpolation techniques proceed in two steps:

1. compute spline coefficients $c(k)$;
2. perform polynomial interpolation using (3).

Since B-splines have a finite support, only a limited number of coefficients $c(k)$ are involved in the computation of $s(x)$.

The case $n = 0$ boils down to nearest neighbour interpolation whereas $n = 1$ identifies with linear interpolation and, in both cases, $c(k) = s(k)$: coefficients identify with signal samples. This is not so for orders $n \geq 2$, where computation of coefficients $c(k)$ involve resolution of the banded system

$$s(l) = \sum_k c(k)\beta^n(l-k), \tag{4}$$

with appropriate boundary conditions. This implies that $c(k)$ depends on all the data $s(l)$. Therefore, for $n > 2$ the B-spline interpolator $s(x)$ depends on all the data, not just the four nearest neighbours, unlike the popular cubic convolution interpolator [3].

Whereas the classical approach for computation of $c(k)$ involves inversion of a banded matrix [2], the presently most efficient CPU implementation for B-spline interpolation involves a two-pass forward-backward recursion on the signal samples [9].

4 Cubic spline interpolation on GPUs

We now focus on the case $n = 3$, *i.e.,* cubic B-spline image interpolation and corresponding GPU implementation. CBS image interpolation is performed by separable 1-D cubic spline interpolations applied on image rows then on image columns. The currently widely available CUDA implementation of cubic B-spline interpolation is due to Ruijters and coworkers [6][7], it strictly follows the previous two-step procedure.

As regards the polynomial interpolation step, Ruijters *et al.* [6] have improved on Sigg and Hadwigger [8] who where the first to propose a cascade of bilinear texture interpolation. Since texture interpolation is hardwired on a GPU, this step is the fastest. The prefilter step is more challenging for parallel implementation since it relies inherently on sequential recursive filtering. Recently, Ruijters & Thévenaz have proposed a parallel implementation of the prefilter on a "one thread per line/column" basis. Although the overall implementation is very competitive with the CPU implementation, especially for large image or 3-D volumes [7] it is still 10 times as much expensive than the bilinear interpolation for applications like image rotation. One practical alternative is to drop the prefilter step but this leads to oversmoothed interpolated images [7].

Our contribution enables to shrink this factor from 10 to 2 without sacrificing accuracy.

5 FIR prefilter approximation

Our technique is very simple: we provide in closed form the prefilter infinite coefficient series equivalent to the usual recursive form and then we show that truncation to the first 15 coefficients yields as accurate results than the recursive implementation of [7][9].

The most compact description of the prefilter $b(k)$ is in terms of its Z-transform [10]:

$$B(z) \triangleq \sum_k b(k) z^{-k} = \frac{6}{z + 4 + z^{-1}}. \tag{5}$$

Indeed, $b(k)$ has also the following simple expression

Proposition 1

$$b(k) = \sqrt{3} \left(\sqrt{3} - 2\right)^{|k|}. \tag{6}$$

Proof: Use Lemma 1 in Appendix with $a = \sqrt{3} - 2$ and normalize so that $B(1) = 1$. ■

The series of the first eigth coefficients of $b(k)$ is drawn on Fig. 1: in particular note the fast exponential decay. This means that a given prefilter output sample actually receives contributions mainly from nearby samples.

The proposed method consists to keep the $(2K+1)$ coefficients $b(k)$, $k = -K, \cdots, 0, \cdots, K$, then to normalize their sum to 1. This approach is denoted "GPU FIR#" where # is the number of prefilter coefficients.

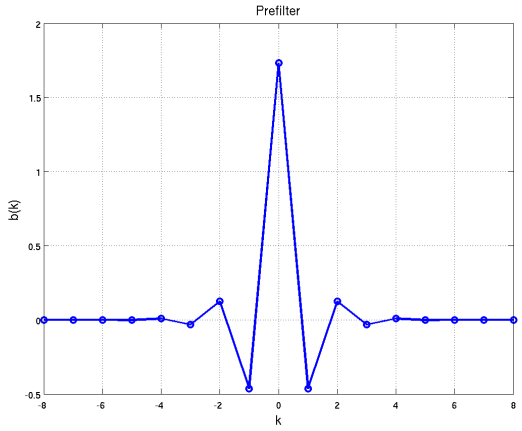

Figure 1: Coefficients of the prefilter b. Note the fast exponential decay.

We check the accuracy of this procedure using the following experiment borrowed from Unser [10] : perform 36 progressive 10° rotations on an image and then compare to the original image. This test is rather selective because each rotation involves an interpolation step and interpolation errors tend to accumulate [7].

This experiment is conducted on the same Lena image as [7]. After 36 rotations, the result is compared to the original Lena image and the gray-level RMS error is computed. Figure 2 presents the RMS error as a function of truncation order $(2K+1)$ it shows that the approximation converges for 11 coefficients.

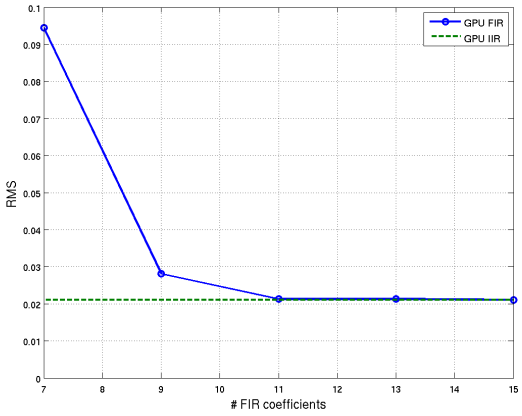

Figure 2: RMS error between a progressively rotated Lena image and the original Lena image. Solid curve : RMS error as a function of the number of FIR coefficients. Dashed curve: RMS error for the GPU IIR method by Thévenaz *et al.* [9], that corresponds to an infinite coefficient series.

In order to gain intuition on the main source of errors, Fig. 3 visualizes the images resulting from this procedure using different implementations. The recursive CPU implementation of the prefilter by Thévenaz *et al.* [9] is denoted CPU IIR, whereas the recursive GPU implementation of the prefilter by Ruijters & Thévenaz [6] is denoted GPU IIR.

Fig. 3 is best understood in conjunction with Fig. 4 that shows residual between the true

Figure 3: The Lena image is progressively rotated 36 times by 10° steps. Top left: CPU IIR implementation [9]. Top middle: Ruijters & Thévenaz GPU IIR implementation [7]. Top right: GPU FIR7. Bottom right: GPU FIR9. Bottom middle: GPU FIR11. Down right: GPU FIR15.

image and the result of the rotated images.

In particular, GPU FIR# images look sharper than CPU IIR and GPU IIR which are almost identical. Inspection of residual images on Fig. 4 reveals that the high frequency content of GPU FIR# is indeed erroneous. This holds in particular for the GPU FIR7 case of Fig. 3 where some noise is apparent in the flat background areas. GPU IIR images and GPU FIR15 images are nearly identical: the maximum discrepancy between both image is inferior to one gray level.

Indeed, we have been using a 7 coefficient approximation in our previous work on optical flow estimation [1] and the end-to-end discrepancy w.r.t. GPU IIR interpolation in this context is inferior to 0.03 pixel, which is far lower than the usual accuracy of optical flow methods. But considering the visual quality of GPU FIR7 images, we set truncation level to 15 coefficients for maximum compliance with the GPU IIR implementation.

6 GPU implementation

The proposed filter works as a separable convolution with 15-coefficients 1D-filters. As regards row filtering, our implementation is quite straightforward and amounts to the solution proposed in the CUDA SDK [5]. Each row is processed with blocks of 256 threads. The row is first copied in the shared memory (SM), and each thread computes one output pixel by summing the data already in SM multiplied by filter coefficients stored in the constant memory. There is an overlapping area of seven pixels between blocks and the first and the last seven threads only load data into the SM.

The implementation for column filtering is more complex and is illustrated in Fig. 5. The image is divided in blocks of 96 columns and 256 rows (these parameters have been optimized for TESLA C2050). There is one thread per column. The kernel starts loading in the SM the 15 rows required to compute the first output row of the block which is, say, row n of the image.

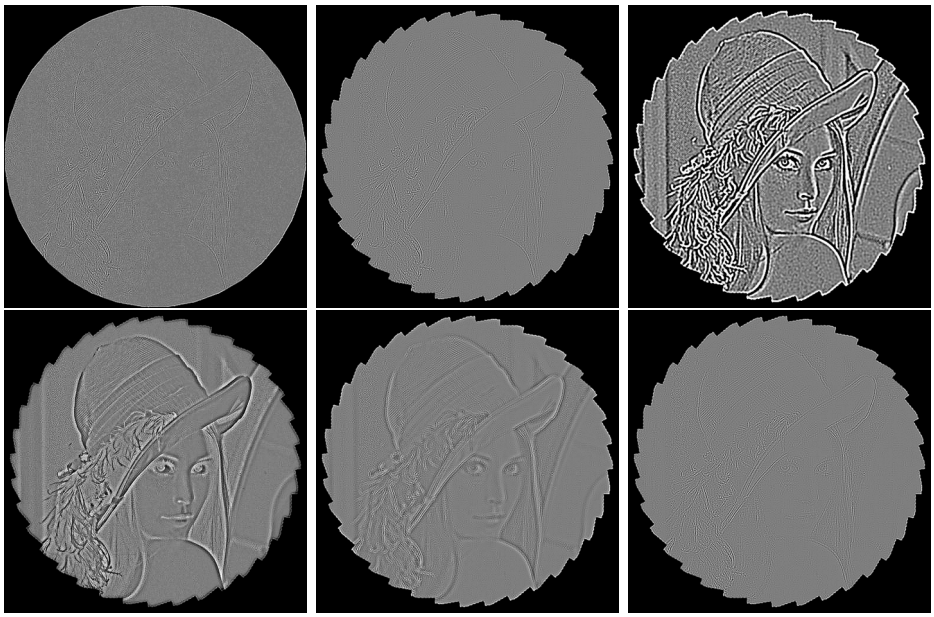

Figure 4: Residual between true image and progressively rotated images. Top left: CPU IIR implementation [9]. Top middle: Ruijters & Thévenaz GPU IIR implementation [7]. Top right: GPU FIR7. Bottom right: GPU FIR9. Bottom middle: GPU FIR11. Down right: GPU FIR15.

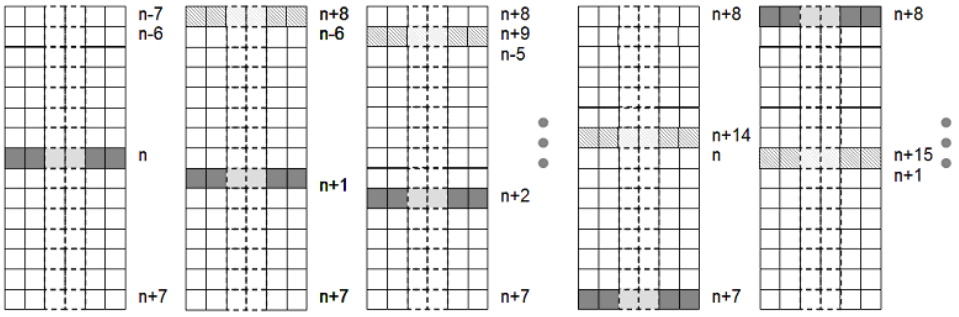

Figure 5: Shared memory management for the column filter. Each individual cell grid represents the 15 rows required at a given step to compute the current output row whose location is represented in dark gray. At each step, the replaced input row is shown in light gray and its vertical location rotates down inside the shared memory.

Table 1: Prefilter calculation times in ms. Comparison of GPU IIR and GPU FIR15 implementations of CBS interpolation.

GPU type	GTX 260M		TESLA C2050	
size (pix.)	1024^2	2048^2	1024^2	2048^2
GPU IIR (ms)	12.78	62.6	5.52	12.506
GPU FIR15 (ms)	2.9	11.15	0.35	1.2

Table 2: Row/column budget for a 2048^2 image on a Tesla C2050.

	GPU IIR	GPU FIR15
row	10.2	0.59
column	2.3	0.61
total	12.5	1.2

Then, the first line in the SM that stores the image row $n-7$ is no longer useful. It is replaced by row $n+8$ and the output line $n+1$ is computed. For the next output line $n+2$, the second line in the SM that stores the image row $n-6$ is then replaced by the image row $n+9$. This process is repeated until the end of the block. It has been devised in order to yield one coalesced loading per line. In conjunction to this optimized memory management, efficient permutation of the filter coefficients is performed in order to match the actual order of image rows in the SM.

The reward for this more complex column filter kernel is that row and column filtering run at about the same speed, as shown in the following experiments, see Table 2.

7 Experiments

Our implementation of cubic B-spline interpolation differs from that of Ruijters & Thévenaz [7] only in the prefilter step, so our experiments primarily assess the difference between both implementations of the prefilter step.

The original purpose of this work is GPGPU [1] so the first target is a Tesla C2050 Nvidia card. But B-spline interpolation concerns more generally visualization so we also performed additional tests on the less powerfull GTX 260M Nvidia Card driving our graphic display.

Table 1 gives the running times in ms for both target cards and two image sizes. The acceleration factor ranges from 4.4 for the GTX 260M with a 1 Mpix image to 10 for Tesla C2050 with a 4Mpix image. So the GPU FIR implementation tends to be more efficient for larger image sizes and more powerful GPUs. Note however that the block sizes have been optimized for Tesla C2050.

Another distinctive feature of the GPU FIR w.r.t. GPU IIR is the row/column behaviour. For GPU IIR, the row filter costs 5 times as much as the column filter. Conversely, row and column filters are well balanced for GPU FIR15.

Finally, we quantify the global prefilter plus polynomial interpolation task w.r.t. the basic bilinear alternative. The respective weight of prefilter vs polynomial interpolation depends on the ratio between output and input image sizes. For geometric transforms like rotation or quadratic warp, this ratio is around 1, this is the context where GPU FIR15 is the most competitive. For image zooming, the prefilter step does not depend on the zoom factor, but the cost of the polynomial interpolation step increases quadratically with the zoom factor so this last step

Table 3: CBS interpolation calculation times in ms. Comparison of GPU IIR and GPU FIR15 implementations of CBS interpolation w.r.t. bilinear interpolation.

	Rotation	×2 zoom	×4 zoom	×8 zoom	×16 zoom
bilinear	1.5	1.56	1.56	1.56	1.56
GPU FIR15	3.2	2.33	2.12	2.05	2.04
GPU IIR	14.5	5.16	2.8	2.2	2.08

progressively dominates.

This behaviour is illustrated in Table 3: the output image size is fixed to 2048^2, but the input image size depends on the transformation. Times have been measured on a Tesla C2050. If the advantage of the GPU FIR over GPU IIR reduces with the zoom factor note that the factor between GPU FIR and bilinear interpolation is always lower than 2, whereas this factor varies much more for GPU IIR. This stability is the reward for reducing the overhead represented by prefilter in cubic spline interpolation.

8 Concluding remarks

The proposed GPU FIR prefilter implementation runs between 4 to 10 times faster than the former GPU IIR prefilter implementation.

However, the FIR implementation involves roughly 5 times as much flops than the IIR implementation. This paradox can be explained by two main factors. The first drawback of the IIR filter is inherent to the forward-backward passes that imply dual access to global memory whereas FIR implementation requires a single access. But, the main drawback of the GPU IIR implementation is that the memory access cannot be coalesced with the row kernel. This behaviour is observed in Table 2: row filtering costs 5 times as much as column filtering for GPU IIR. This could be improved using the shared memory or even using a transposition that enables to proceed row as column. But the resulting column routine is not as efficient as ours because each thread processes a whole column, which means that there are not enough blocks per multi-processor to hide the memory access while switching to another block.

The success of our approach to cubic B-spline prefiltering relies on the better match of GPU architectures with FIR filters than recursive filters. We took advantage of the fact that an explicit expression for the cubic B-spline prefilter coefficients could be derived and that a limited number of them provides enough accuracy. Finally, one key feature is the distinct handle of row and column filters. Note that our procedure for column filtering can be generalized straightforwardly in the case of higher dimension volumes.

Appendix

Lemma 1 *Let $a \in \mathbb{C}$, $0 < |a| < 1$, then $a(k) = a^{|k|}$, $k \in \mathbb{Z}$ defines a stable filter whose Z-transform $A(z) \triangleq \sum_k a(k) z^{-k}$ reads*

$$A(z) = \frac{1 - a^2}{1 + a^2 - a(z + z^{-1})} \text{ for } |a| < |z| < |a|^{-1}. \tag{7}$$

Proof:

$$
\begin{aligned}
A(z) &\triangleq \sum_k a^{|k|} z^{-k} && (8)\\
&= \sum_{k\geq 0} a^k z^{-k} + \sum_{k\leq -1} a^{-k} z^{-k} && (9)\\
&= \sum_{k\geq 0} \left(az^{-1}\right)^k + \sum_{k\geq 1} (az)^k && (10)\\
&= \frac{1}{1-az^{-1}} + \frac{az}{1-az} && (11)\\
&= \frac{1-a^2}{1+a^2-a(z+z^{-1})}. && (12)
\end{aligned}
$$

■

References

[1] F. Champagnat, A. Plyer, G. Le Besnerais, B. Leclaire, S. Davoust, and Y. Le Sant. Fast and accurate PIV computation using highly parallel iterative correlation maximization. *Experiments in Fluids*, 50:1169–1182, 2011.

[2] Carl De Boor. *A Practical Guide to Splines*. Springer-Verlag, 1978.

[3] R. Keys. Cubic convolution interpolation for digital image processing. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 29:1153–1160, 1981.

[4] Guy Le Besnerais and Frédéric Champagnat. B-spline image model for energy minimization-based optical flow estimation. *IEEE Transactions on Image Processing*, 15(10):3201 – 3206, October 2006.

[5] V. Podlozhnyuk. *Image Convolution with CUDA*. Nvidia Corp, June 2007.

[6] Daniel Ruijters, Bart M. ter Haar Romeny, and Paul Suetens. Efficient GPU-based texture interpolation using uniform B-splines. *Journal of Graphics, GPU, & Game Tools*, 13:61–69, 2008.

[7] Daniel Ruijters and Philippe Thévenaz. GPU prefilter for accurate cubic B-spline interpolation. *The Computer Journal*, 2010.

[8] Christian Sigg and Markus Hadwiger. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, chapter Fast Third-Order Texture Filtering, pages 313–329. Addison-Wesley, 2005.

[9] P. Thévenaz, T. Blu, and M. Unser. Interpolation revisited. *IEEE Transactions on Medical Imaging*, 19(7):739–758, July 2000.

[10] M. Unser. Splines: A perfect fit for signal and image processing. *IEEE Signal Processing Magazine*, 16(6):22–38, November 1999.